Confidentiality Enforcement Using Dynamic Information Flow Analyses

by

Gurvan LE GUERNIC

M.S., Institut National des Sciences Appliquées de Rennes, France, 2002
M.S.E., Institut National des Sciences Appliquées de Rennes, France, 2002

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2007

Abstract

With the intensification of communication in information systems, interest in security has increased. The notion of *noninterference* is typically used as a baseline security policy to formalize confidentiality of secret information manipulated by a program. This notion, based on ideas from classical information theory, has first been introduced by Goguen and Meseguer (1982) as the absence of *strong dependency* (Cohen, 1977).

> "information is transmitted from a source to a destination only when variety in the source can be conveyed to the destination" Cohen (1977)

Building on the notion proposed by Goguen and Meseguer, a program is typically said to be *noninterfering* if the values of its public outputs do not depend on the values of its secret inputs. If that is not the case then there exist illegal information flows that allow an attacker, having knowledge about the source code of the program, to deduce information about the secret inputs from the public outputs of the execution.

In contrast to the vast majority of previous work on noninterference which are based on static analyses (especially type systems), this PhD thesis report considers dynamic monitoring of noninterference. A monitor enforcing noninterference is more complex than standard execution monitors.

> "the information carried by a particular message depends on the set it comes from. The information conveyed is not an intrinsic property of the individual message." Ashby (1956).

The work presented in this report is based on the combination of dynamic and static information flow analyses. The practicality of such an approach is demonstrated by the development of a monitor for concurrent programs including synchronization commands. This report also elaborates on the soundness with regard to noninterference and precision of such approaches.

Confidentiality Enforcement Using Dynamic Information Flow Analyses

by

Gurvan LE GUERNIC

M.S., Institut National des Sciences Appliquées de Rennes, France, 2002
M.S.E., Institut National des Sciences Appliquées de Rennes, France, 2002

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2007

Approved by:

Co-Major Professor	Co-Major Professor	Co-Major Professor
Anindya Banerjee	Thomas Jensen	David Schmidt

Abstract

With the intensification of communication in information systems, interest in security has increased. The notion of *noninterference* is typically used as a baseline security policy to formalize confidentiality of secret information manipulated by a program. This notion, based on ideas from classical information theory, has first been introduced by Goguen and Meseguer (1982) as the absence of *strong dependency* (Cohen, 1977).

> "information is transmitted from a source to a destination only when variety in the source can be conveyed to the destination" Cohen (1977)

Building on the notion proposed by Goguen and Meseguer, a program is typically said to be *noninterfering* if the values of its public outputs do not depend on the values of its secret inputs. If that is not the case then there exist illegal information flows that allow an attacker, having knowledge about the source code of the program, to deduce information about the secret inputs from the public outputs of the execution.

In contrast to the vast majority of previous work on noninterference which are based on static analyses (especially type systems), this PhD thesis report considers dynamic monitoring of noninterference. A monitor enforcing noninterference is more complex than standard execution monitors.

> "the information carried by a particular message depends on the set it comes from. The information conveyed is not an intrinsic property of the individual message." Ashby (1956).

The work presented in this report is based on the combination of dynamic and static information flow analyses. The practicality of such an approach is demonstrated by the development of a monitor for concurrent programs including synchronization commands. This report also elaborates on the soundness with regard to noninterference and precision of such approaches.

Contents

List of Figures

List of Tables

Acknowledgments

As an attempt to apologize, I want to start by giving thanks to all those I will forget in the following acknowledgments. To my regret, I know they will be too much of them, but that does not mean I do not recognize their impact on the fulfillment of this thesis.

Among those I will name, I owe a huge debt to my official and unofficial advisors who accepted to follow me in this unusual PhD. Anindya Banerjee deserves acknowledgment for introducing and guiding me in his field of study which also became mine. I am grateful to Thomas Jensen for backing up my choices from the early beginning of my research pursuit. I highly appreciate the help furnished by David Schmidt to make this joint-PhD possible, his deep understanding of PhD student supervision, and his ability to unknot other's difficulties. My advisors deserve the highest acknowledgment for having succeeded to put the right level of pressure in order to ensure a continuous progress of this thesis without breaking its momentum. Thanks for your help, guidance, and patience during my thesis.

This thesis also owes a lot to the comity members who accepted to review the work accomplished during my PhD. I am grateful to Gérard Boudol, Andreï Sabelfeld, Frédéric Cuppens and Claude Jard for the improvements they brought this thesis.

I am thankful to my colleagues at KSU and IRISA, especially those of the Lande team of course, for supporting me in all the meanings of the term. Special thanks go to those working at the IRISA's cafeteria for keeping a cheerful atmosphere in this haven for much needed breaks.

Special acknowledgment to Linda which enabled me to solve issues unrelated to my thesis but which could have put an end to it more surely than any theoretical difficulties. I owe you a lot.

I want to express warm thanks to my friends, which I will not name otherwise I will thank even more people in the first paragraph. They know who they are, and I am sure they know how important they are. I thank those who know what a PhD is for having followed the same path, as well as those who do not but learned a bit about it by supporting those who do. Thank you for having been there to discuss about the PhD student life when needed, and allowed me to escape of my PhD on other times.

Of course, nothing would have really been possible without my family. It is an overused statement, but so much true. I owe way more to my parents, brother, cousins, grand-mother, aunts and uncles than the considerable debt I have toward them just for this thesis. You are the strong and reliable foundations on which I can harmoniously grow.

And finally, I have some specific thoughts for two special girls, Elizabeth and Faith, which change my life for ever.

Chapter 1

Introduction

Julius Caesar's cipher (Kahn, 1996) is one of the earliest recorded mechanisms for the protection of secret information. Early mechanisms were mainly interested in ensuring the confidentiality of data while it is transferred. Their goal is to prevent an enemy, or attacker, to learn anything useful even if he acquires the protected data. Such mechanisms are based on cryptographic techniques (Menezes et al., 1996).

With the spread of information systems in the middle of the last century, secret data were confronted by a new threat. Information systems store more and more secret, or private, data. As before, the confidentiality of those data can be impaired while they are transmitted from one system to another. This is still mainly dealt with using cryptographic techniques. However, with information systems, an attacker can additionally attempt to gain knowledge about secret data by extracting them completely or partially from the system. Therefore, ensuring the confidentiality of secret data stored in and manipulated by information systems has always been a subject of interest and research.

There are two main approaches for the attacker. He might tamper with the authentication mechanism. If it succeeds, the attacker may be able to access the information system with the identity and permissions of someone having access to secret data. The other approach consists of deducing information about the secrets from the observable outputs of a system executing a precise process, malicious or not. Of course, any secret information in a system could be encrypted to prevent processes from leaking useful information. However, for efficiency reasons, this is not what is done. Moreover, secret information may be required by some processes. This information must then be provided unencrypted to those processes. This document is interested in thwarting the second approach, i.e. information leakage. Different techniques exist to prevent information leakage. Among them, the principal two are access control (Denning, 1982, Chapter 4) and, to a lesser degree, information flow control (Denning, 1982, Chapter 5).

Access Control is the most widely used technique to enforce confidentiality. It is also the first technique to have been developed. It relies on the assumption that a process can not leak information about data it does not have access to. Therefore, the main idea is to restrict access of processes to the smallest number of data needed for their fulfillment. This is an instance of the *least privilege* design principle (Saltzer and Schroeder, 1975).

There exist many different access control policies. The most well known and used type of access control policy is the *Discretionary Access Control* (DAC) (Brand, 1985; National Security Agency, 1995b, Chapter 2). In a DAC policy, access rights of every principal (process or user) on every piece of data (or object) are explicitly stated. Bell and LaPadula's security model (1973b) implements DAC security policy using a matrix. Each cell of the matrix explicitly states the access rights of a principal, associated with the row of the cell, on an object, associated with the column of the cell. In a DAC policy, it is often the case that the owner of an object — in the majority of cases, the principal who created it — is the entity in charge of deciding what are the access rights of other principals on that object. *Mandatory Access Control* (MAC) (Brand, 1985; National Security Agency, 1995a, Chapter 3) is the other widely used type of access control policy. In a MAC policy, a security level is associated to every principal and every piece of data. The security level associated to a principal is often called a *security clearance*. The level of data is often called a *security classification*. The decision of allowing a user to access an object in a given mode is taken by comparing the security clearance of the principal with the security classification of the object. In the majority of cases, the security levels are ordered (frequently forming a lattice). In that case, a principal is usually allowed to read an object only if its security level is higher than the object's one (called the *no read-up* rule), and allowed to modify an object only if its level is lower than the object's one (called the *no write-down* rule). In a MAC policy, security levels of newly created data are usually automatically assigned by the *Trusted Computing Base* (TCB) (Saltzer and Schroeder, 1975). Finally, another example of access control policy is the so called *Chinese Wall* (Brewer and Nash, 1989). With such a policy, the rights of a principal depend on data this principal has already accessed. The main goal of this policy is to prevent a principal to carry information from one set of data to another one which must be kept ignorant of the content of the first set.

Access control is quite efficient at enforcing the least privilege design principle. However, this is sometimes insufficient because, once the access has been granted, there is no more real control of the dissemination of data. A process may need to access confidential data in order to fulfill its goal. An owner may agree to allow this process to access confidential data for the fulfillment of the process purpose. However, it is likely that the owner does not want the process to spread the information it has been given access to. Unfortunately, this spreading of information often can not be practically and efficiently enforced using access control. For example, a tax computation service needs access to the financial information of the client. It also needs access to the service provider server, for example, to send a bill if the service cost is on a per use basis or simply to download the latest updates. Then what are the customer's guarantees that its financial data are not unduly sent to the service provider? To enforce such a policy, it is required to enforce some sort of information flow control. Additionally, many end-users do not have a sufficient knowledge of their personal information system to set it up adequately to protect their confidential data (Good and Krekelberg, 2003; Leyden, 2006; McCue, 2006; Page, 2007; Ranger, 2007). This is another reason why user-centric end-to-end information flow control mechanisms are of high interest.

1.1 Information Flow Control

The goal of information flow control is to ensure the sole existence of safe, or secure, information flows in a process, or more generally in an information system. Lampson (1973) defines three different categories of channels which can be used by unsafe information flows. The first one is the category of *legitimate channels*. Such channels are based on mechanisms intended for information transfer which are needed for the legitimate operation of the information system. For example, a web browser needs to have access to the Internet. However, an information flow control mechanism must prevent the web browser to create an undesired flow from confidential data on the computer to a particular web site. The second category concerns *storage channels*. Such channels make use of mechanisms which are not primarily intended to transfer data, but to store it. The information is first transferred from the source of the flow to a storage location and then, later, from the storage location to the destination of the flow. Such channels are used to attempt to fool the information flow control mechanism by delaying in time and space the realization of the undesired flow. The last category of channels contains the so called *covert channels*. Such channels use mechanisms which are not intended for the manipulation (transfer, computation or storage) of information. They are based on the encoding and transfer of information from the source of the flow into the side effects of legitimate mechanisms. Information contained in those side effects are then decoded and transferred to the destination of the flow. For example, such channels can use locks, intended for preventing race conditions, or even the computation time of a given process. It is in general impossible to construct an information flow control mechanism which is both sound and complete (Denning, 1982; Jones and Lipton, 1975, Sect. 5.2.1). The channels which must be overseen by an information flow control mechanism, and the constraints which must be put on them, depend on the desired equilibrium between the confidentiality of secret data and the efficiency and capabilities of the information system. There exist different properties defining different levels of secure information flows.

In 1973, Lampson introduced the notion of *confinement*. It is a general notion which states that a process or information system does not unduly leak confidential data. Lampson (1973) states that a process can be confined only if it is not able to maintain secret information longer than its own execution. Such processes are called *memoryless* (Fenton, 1974b; Gat and Saal, 1976). Additionally, it is required to call only secure programs or confined ones. Lipner (1975) explores the available techniques which can be used to confine a process.

Bell and LaPadula (1973b) defined a secure information flow property that they named the $\star$*-property*. It requires that every object a given principal (user or process) can read has a lower or equivalent security level than any object the same principal can write into. It is obvious to see that this property effectively prevents any unsafe flows through legitimate or storage channels. It can be enforced using access control mechanisms without too much difficulty. However it is also a very restrictive property. In order to enforce this property, a web browser, which requires “write” access to the Internet to send its requests, should be allowed access only to data with the lowest level of security. It may thus be denied access to a bookmarks file or even to a preferences, or settings, file.

Cohen (1977) introduces the notion of *strong dependency*. It borrows from classical information theory (Ashby, 1956) the idea that there exist an information flow from input i to output o in a process P whenever *variety* in i is conveyed to o by the execution of P.

> "information is transmitted from a source to a destination only when variety in the source can be conveyed to the destination" Cohen (1977, Sect.1)

In other words, for deterministic processes, o is strongly dependent on i if and only if there exist at least two executions of P whose inputs differ only in i and whose outputs differ in o. Goguen and Meseguer (1982) define their notion of *noninterference* as the absence of strong dependencies. They first define it as follows:

> "one group of users, using a certain set of commands, is $\underline{\text{noninterfering}}$ with another group of users if what the first group does with those commands has no effect on what the second group of users can see." Goguen and Meseguer (1982)

It states that the effects observable by the second group are not strongly dependent on the actions of the first group. This definition is refined more formally as follows:

> "commands in [the set] A, issued by users in G, are **noninterfering** with users in G' provided that any sequence of commands to the system, given by any users, produces the same effect for users in G' as the corresponding sequence with all commands in A by users in G deleted;" Goguen and Meseguer (1984)

Those definitions are more suited for information systems in which actions are generated in an unpredictable way and where it is the actions of a group which must be kept secret rather than data. When the secrets are data manipulated by a process, which follows a partially predictable sequence of actions, the notion of noninterference is usually stated as follows. A process, P, is said to be *noninterfering* if the values of its public (or low) outputs do not depend on the values of its secret (or high) inputs. Formally, noninterference of P is expressed as follows: given any two initial input states σ_1 and σ_2 that are indistinguishable with respect to low inputs, the executions of P started in states σ_1 and σ_2 are *low-indistinguishable*; i.e. there is no observable difference in the public outputs. In the simplest form of the *low-indistinguishable* definition, public outputs include only the final values of low variables. In a more general setting, the definition may additionally involve intentional aspects such as power consumption, computation times, etc. Some variants of the notion of noninterference have been proposed. For example, Giacobazzi and Mastroeni (2004) define a notion of *abstract noninterference*. It states that there is no strong dependency between some properties of the secret inputs and some properties of the public outputs; for example, their parity or sign. The properties on outputs are those that the attacker can distinguish. The properties on inputs are those that must be kept secret.

The work presented in this document is interested in the notion of noninterference of programs. The *attacker model* for noninterference analysis of programs is quite standard. The attacker is considered to have access to the program source code and to the values of public inputs and public outputs. Noninterference analyses must prevent the attacker to deduce any piece of information about the secret inputs of a given program by observing one or more of its executions. There have been lots of developments in the domain of static analyses of programs for noninterference (Sabelfeld and Myers, 2003). Some of them are reviewed in Section 2.1.

1.2 Thesis: Dynamic Information Flow Analyses Are Worthwhile

The interest for the control of information flows in processes appeared in the early 70's (Lampson, 1973; Lipner, 1975). At that time, some dynamic analyses for information flow were developed (Fenton, 1974b). The majority of them are described at the machine level. However, those efforts have not been pushed further, maybe because the level of confidentiality investigated was too strong and therefore the machines developed too restrictive. In the 80's (McHugh and Good, 1985), 90's (Volpano et al., 1996), and early 2000's decade (Barthe et al., 2007; Sabelfeld and Myers, 2003), efforts have been put on the development of static analyses for information flow control. Even if the problem of information flows in processes is difficult, those analyses achieve a good level of precision and their respective soundness is strongly proved. The middle of the 2000's decade has seen the resurgence of development of dynamic analyses for information flow (Vachharajani et al., 2004). Those analyses deal with quite expressive languages. However, as emphasized in Section 2.2, they lack formal proofs of their soundness. Such proofs are the strength of static information flow analyses. They are what allow a user to confidently run an analyzed program.

Static analyses are well fitted for information systems that are administered strongly. Programs to be installed on the system can be analyzed by the administration staff before being deployed on the system. However, with the fast spreading of information systems in our society, more and more users own small information systems which store personal data, execute programs coming from different sources and whose sole administrator is the user itself. For example, cell phones are such devices. Lots of them store the owner directory and calendar. There are even some developments to incorporate some credit card functionalities to such devices. Those same devices execute programs, such as games or communication softwares, which are downloaded from sources that the user does not fully trust. Thus comes the worry for users that the downloaded programs may leak some confidential information. It would be comforting for users to know that the executions of such programs are monitored by a mechanism that the user can safely trust.

This document defends the thesis that it is possible to develop and prove the soundness of precise dynamic information flow analyses for expressive languages. It presents and proves the soundness of two dynamic analyses having different levels of precision for a core language. It extends one of those works to a more expressive language, a concurrent language including synchronization commands. The remainder of this chapter gives an overview of the advantages, drawbacks and difficulties of dynamic information flow analyses.

1.2.1 Advantages of Dynamic Analyses

There are some advantages to using dynamic information flow analyses rather than static ones. First of all, dynamic analyses are more user-centric with regard to flow policy definitions than static analyses. End-users may have different definitions of what must be considered secret. Some may consider secret the content of their address book whereas others do not. They may also have different and changing requirements with regard to what can be done with secret data. Some end-users may consider that secret data must note be displayed on the screen of their mobile phone whenever they are in a crowd, where people may "have a look". Static analyses are usually run at compile time once and for all. With a static analysis, the potential flow policies to be respected must therefore be known and verified at compile time. However, it is not realistic to statically verify all potential flow policies. Some flexibility can be obtained by developing static analysis techniques for run-time security checks (Banerjee and Naumann, 2005; Tse and Zdancewic, 2007) and declassification (Matos and Boudol, 2007; Sabelfeld and Sands, 2007). On the other hand, dynamic analyses are run with every execution. Therefore, the ability for end-users to modify the flow policy before every run of a program comes for free with dynamic analyses.

Moreover static analyses concern a given program and thus all its executions. Therefore, if a single execution of a program leaks confidential information then the program is rejected, and thus all of its executions. A dynamic analysis, on the other hand, verifies a property for a single execution, independently of the behavior of other executions with regard to this property. Therefore, with a dynamic analysis, it is possible to run some safe executions of a program even if one of its executions can not be proved to respect the flow policy.

Finally, as a dynamic analysis verifies a given property during an execution, it can gain a more precise knowledge of the control flow of a program than a static analysis would. As an example, let us consider the following program where *h* is the only secret input and *l* the only other input (a public one).

```
if ( test1(l) ) then tmp := h else skip end;
if ( test2(l) ) then x := tmp else skip end;
output x
```

Without information on *test1* and *test2* (and often, even with), a static analysis would conclude that this program is unsafe because the secret input information could be carried to *x* through *tmp* and then to the output. However, if *test1* and *test2* are such that no value of *l* makes both predicates true, then any execution of the program is perfectly safe. In that case, the monitoring mechanism would allow any execution of this program. The reason is that, *l* being a public input, only executions following the same path as the current execution are taken care of by the monitoring mechanism. So, for such configurations where the branching conditions are not influenced by the secret inputs, a monitoring mechanism is at least as precise as any static analysis — and often more precise.

1.2.2 Drawbacks of Dynamic Analyses

Of course, there are some drawbacks to using dynamic analyses rather than static ones. The first obvious one is the decrease in execution speed. A static analysis is run once and for all. Dynamic analyses must be run for every execution. It is then a burden for every execution.

Additionally, the information flow monitor modifies the original semantics of the program. The effects of a program once monitored may not be the same as the intended one. However, this is a required behavior for monitors. The behavior of a program leaking confidential data *must* be altered by the monitor. Otherwise, the monitor would not do its job.

The difficulty to report security violations (Denning, 1982, Sect. 5.3.1) is a similar problem. A static analysis can return a complete report of the security violations to the authority executing the analysis. A dynamic analysis is not always able to report those violations. As it is applied at run time, reporting violations may by itself create an insecure information flow (Le Guernic and Jensen, 2005).

Finally, dynamic information flow analyses add new protection requirements. The monitoring mechanism itself must be protected from tampering by an attacker. A well crafted Trojan horse may be designed to modify some values on which relies the information flow monitor. Only the Trusted Computing Base (TCB) must be allowed to set and modify data used by the protection mechanism.

1.2.3 Difficulties of Dynamic Analyses

Monitoring information flow is more complex than monitoring standard properties (e.g. division by zero). Monitors which see executions as a sequence of executed actions, like standard execution monitors (Schneider, 2000), are not sufficient to enforce strong information flow properties. This particular point is supported by works of McLean and Schneider. McLean (1994) proved that information flow policies equivalent to noninterference are not *trace properties*.

> "The fact that possibilistic properties [, among which is noninterference,] are no properties of traces follows immediately from the fact that they are not preserved by trace subsetting." McLean (1994, Sect. 2.1)

Schneider (2000) concluded that execution monitors are limited to the enforcement of *trace properties*.

> "In Alpern and Schneider [1985] and the literature on linear-time concurrent program verification, a set of executions is called a *property* if set membership is determined by each element alone and not by other members of the set. Using that terminology, we conclude [. . .] that a security policy must be a property in order for that policy to have an enforcement mechanism in [the class of Execution Monitors]." Schneider (2000, Sect. 2)

In order to enforce strong constraints on the information flow (like noninterference), the monitoring mechanism must be aware of the commands that are *not* evaluated by a given execution (Vachharajani et al., 2004, Sect. 4.2.2). An information theoretic viewpoint why this is the case is given by Ashby.

> "the information carried by a particular message depends on the set it comes from. The information conveyed is not an intrinsic property of the individual message." Ashby (1956, § 7/5 page 124).

For example, executions of the following programs, (a) and (b), in an initial state where h is `false` are equivalent concerning executed commands.

(a) **if** h **then** x **:=**1 **else** **skip** **end**; **output** x

(b) **if** h **then** **skip** **else** **skip** **end**; **output** x

In contrast, (b) is obviously a noninterfering program, while (a) is not. The execution of (a), with a low-equivalent initial state where h is `true` and x is `0`, does not give the same final value for the low output x. Therefore, a monitor for information flow must take into account not only the current state of the program but also the execution paths *not taken* during execution.

Additionally, it is not sufficient for a noninterference monitor to efficiently track information flows. It must also adequately correct unsafe information flows. As explained in Section 2.2.4, lots of previous works on dynamic analysis of information flow lack an appropriate correction of unsafe information flows. Le Guernic and Jensen (2005) emphasize the fact that an inadequate correction mechanism can lead to the creation of a new covert channel. This channel can then be used by an attacker to gain information about some confidential data.

1.2.4 Organization of the Dissertation

The remainder of this document is organized as follows. Chapter 2 starts by reviewing some static analyses of information flow, thus giving an overview of the accomplishments in the domain of static analyses for noninterference-like properties. The chapter follows by giving a progress report of the accomplishment in the domain of dynamic information flow analyses. In conclusion, it discusses the main weaknesses of previous work on dynamic information flow analyses compared to static information flow analyses. The main flaw of the majority of previous accomplishments is the absence of formal proofs that the developed analysis is not only able to detect "bad flows" but also to monitor noninterference. In other words, they lack demonstration that the proposed mechanisms are well fitted for the dynamic correction of "bad flows". Chapter 3, whose content as been published previously (Le Guernic et al., 2006a), gives the formal definition of an automaton-based noninterference monitor able to dynamically correct flows from secret data to public outputs. It deals with a simple sequential language which is the core of any imperative language. A formal proof is given which states that any execution monitored by this mechanism is noninterfering. It is also proved that the monitoring mechanism is transparent for any program well-typed under a type system similar to the one of Volpano et al. (1996). Chapter 4, whose content as been published previously (Le Guernic, 2007a), demonstrates that the approach followed in Chapter 3 can be extended to more complex languages. This chapter extends the work of Chapter 3 to concurrent programs. The language also incorporates an expressive synchronization command. As in Chapter 3, soundness with regard to noninterference and transparency for a defined set of programs are proved. Chapter 5 comes back to the sequential language of Chapter 3. It presents a noninterference mechanism which takes into consideration more information about the context of the execution. It is then possible to increase the precision of the information flow analysis. As in the two previous chapters, soundness with regard to noninterference is proved. In addition, it is proved that the proposed dynamic analysis is more precise than the analysis of Chapter 3 (this analysis being already more precise than the type system of Volpano et al. (1996)). Chapter 6 concludes this dissertation.

Chapter 2

Confidentiality as Secure Information Flow

Following Denning's work (1976; 1977) on the certification of programs for secure information flows, a considerable number of secure information flow analyses have been developed. The majority of them, but not all, attempts to prove that a given software respects the program property of *noninterference* (Goguen and Meseguer, 1982). Many of those analyses, as the historical one by Volpano, Smith, and Irvine (1996), are based on a type system. Section 2.1 gives an overview of the achievements in the field of static analyses for secure information flow. For a wider insight on the subject, the reader is referred to the noteworthy survey by Sabelfeld and Myers (2003). On the other hand, Section 2.2 reviews the chief, at least to the author's knowledge, dynamic analyses for secure information flow. This type of analyses, which have been sporadically developed since the early work of Weissman (1969), attempts to enforce an execution property similar to *noninterference* by means of monitoring.

2.1 Static Information Flow Analyses

Starting with Bell and LaPadula (1973a,b), the research domain of static analyses for confidentiality has been greatly influenced by the work of Goguen and Meseguer (1982) defining the notion of noninterference and of Volpano et al. (1996) proposing a type system for which well typed programs respect the non-interference property. Therefore, a lot of research on static analyses for secure information flow address the problem of noninterference using type systems. This approach is studied in Section 2.1.1. However, as concisely summarized by Section 2.1.2, other techniques have also been used, for example abstract interpretation or constraint based analyses. As usual when attacking a new research field or applying a new technique to a field, early analyses have often been applied to simple programming languages. Yet, as shown by Section 2.1.4, the languages addressed by those analyses have since been extended to include modern features. Some of them can be qualified as “real world” languages. Not only the languages have been extended, as shown by Section 2.1.3, the property checked itself has been augmented in order, for example, to be termination-aware or to deal with nondeterminism. Despite the advantages inherent to static analyses, those analyses for confidentiality suffer from a main drawback intrinsic to their compile-time nature. Static analyses for secure information flow can not be practically applied by end-users, thus removing from their hand the total control over the security policy to be respected. In order to circumvent this problem, some static analyses taking into account dynamic information flow tests have been developed (Section 2.1.5). Section 2.1.6 concludes the review of static analyses for confidentiality.

2.1.1 Type-based Analyses

The work of Volpano, Smith, and Irvine in 1996 launched an interest in type systems (Pierce, 2002; Schmidt, 1994) for examining information flows. A type system for information flow is a set of rules which are used to check if a typing environment (γ) is compatible with a given program. A typing environment is a mapping from variable identifier to security type. Security types comes from a lattice ($\mathbb{L}, \leq$) of security levels ($\mathbb{L}$) including a bottom element ($\perp$ or L) and a top element ($\top$ or H). The rules of a sound type system for secure information flow are such that: if γ is compatible with the program $\mathtt{P}$ and $\gamma(x) < \gamma(y)$ then the final value of x after execution of $\mathtt{P}$ is not influenced by the initial value of y. Therefore, a program $\mathtt{P}$, whose input variables $i_1 \mathinner{..} i_n$ have security levels $\mathcal{L}(i_1) \mathinner{..} \mathcal{L}(i_n)$ and output variables $o_1 \mathinner{..} o_m$ have security levels $\mathcal{L}(o_1) \mathinner{..} \mathcal{L}(o_m)$, is considered *safe* if there exists a compatible typing environment γ such that, for every input i_k, $\gamma i_k = \mathcal{L}(i_k)$ and, for every output o_k, $\gamma o_k = \mathcal{L}(o_k)$. Type systems are intrinsically flow-insensitive as the security level associated to variables by γ are constant and not mutable as the values are. Additionally, a type system does not "compute" the security level associated to variables which are neither input nor output. Therefore, in order to automatically analyze programs having such variables type-based noninterference analyses must also include an inference mechanism able to "compute" a compatible security level for those variables.

Volpano et al. (1996) propose a type-base noninterference analysis for a deterministic sequential language. The grammar of this language is given in Figure 2.1 where x stands for variable identifiers, e stands for expressions without side effects and S stands for statements (a program being a statement). Their type system is proved to be sound with regard to the simplest definition of noninterference.

$$
\begin{array}{lcl}
S & ::= & x \mathbin{\mathbf{:=}} e \\
 & | & \mathbf{skip} \\
 & | & S \;;\; S \\
 & | & \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S\ \mathbf{end} \\
 & | & \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}
\end{array}
$$

Figure 2.1: Grammar of the language of (Volpano et al., 1996)

A type system for noninterference equivalent to the one in (Volpano et al., 1996) is given in Figure 2.2. The definition of typing environments (γ) is extended in order to also associate a security level to any expression e. The definition of "compatibility" is accordingly extended so that a typing environment γ is compatible with a program if it respects the conditions stated previously and additionally is such that γe is greater or equal to the least upper bound ($\sqcup$) of the security levels of the free variables in e. It is obvious that the different values assigned to the program counter when executing a software influence the outputs as those are dependent on the control flow. In order to take into account this influence, the program counter

(pc) can be seen as a variable which is reassigned at the beginning of every conditional and whose value is used in any assignment. The typing environment also associates a security level to pc which is used in the typing rules. Finally, the typing environment $\gamma[x \mapsto l]$ is equal to γ except for x which is associated the security level l.

$$\frac{(\gamma(e) \sqcup \gamma(pc)) \leq \gamma(x)}{\gamma \vdash x := e} \qquad \text{(T-ASSIGN)}$$

$$\frac{}{\gamma \vdash \textbf{skip}} \qquad \text{(T-SKIP)}$$

$$\frac{\gamma \vdash S_1 \qquad \gamma \vdash S_2}{\gamma \vdash S_1 \; ; \; S_2} \qquad \text{(T-SEQ)}$$

$$\frac{\gamma[pc \mapsto (\gamma(pc) \sqcup \gamma(e))] \vdash S_1 \qquad \gamma[pc \mapsto (\gamma(pc) \sqcup \gamma(e))] \vdash S_2}{\gamma \vdash \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}} \qquad \text{(T-IF)}$$

$$\frac{\gamma[pc \mapsto (\gamma(pc) \sqcup \gamma(e))] \vdash S}{\gamma \vdash \textbf{while } e \textbf{ do } S \textbf{ done}} \qquad \text{(T-WHILE)}$$

Figure 2.2: A type system equivalent to the one of (Volpano et al., 1996)

Consider the *typing judgment* (T-SKIP). The statement **skip** has no impact on the information flows of a program, at least when dealing with a definition which is not timing aware. As expected, (T-SKIP) states that any typing environment is compatible with the program **skip**. Similarly, the rule (T-SEQ) is easily understood. A typing environment is compatible with the program S_1 ; S_2 if it is compatible with the program S_1 and the program S_2. The typing judgment (T-IF) may be more difficult to understand. It states that a branching statement, whose condition is e, is compatible a typing environment γ if all branches are compatible with a typing environment where the program counter as a security level at least as high as the expression e (because e influences the control-flow under which those branches are executed) and as high as the security level associated to the program counter by γ (because previous control-flow is still active for the execution of the branches). The rule (T-WHILE) works similarly. Finally, (T-ASSIGN) states that assigning the value of an expression e to a variable x is compatible with the typing environment γ if the security level of x is at least as high as the one of e (this part take care of *direct* flows) and as high as the security level of the program counter, and hence as high as the information carried by the control-flow (this part take care of *indirect* flows, sometimes called implicit flows (Sabelfeld and Myers, 2003)).

Many other works involve type systems and secure information flow. To name a few, Heintze and Riecke (1998) developed a type system for a calculus, SLam, based on the λ-calculus (Barendregt, 1984). Pottier and Conchon (2000) describe a systematic way of producing a sound type system usable for noninterference checking without investing too much effort. As an example, they start from a lambda-calculus possessing a sound type system and extend it with labels. They prove that, in the operational semantics they build for this language, labels track adequately the information flows. Extending slightly the original types, they then prove, using the original soundness proof of the type system and a translation mechanism to the original language, that the information flows created by the evaluation of a labeled lambda-term is given by the type of the translated lambda-term. Finally, Abadi, Banerjee, Heintze, and Riecke (1999) noticed that a number of static analyses are based on a notion of dependency, for example secure information flow of course but also binding-time analysis, call tracking, and program slicing. Based on this observation, they propose a framework for type-based dependency analysis called *Dependency Core Calculus (DCC)*. To demonstrate the validity of their framework, they translate into *DCC* some type calculi, including some involved with secure information flow.

2.1.2 Static Analyses Using Other Mechanisms

Not all static analyses for secure information flow are expressed as a type system. For example, the technique used in the early work of Denning in 1976, even if highly similar to type systems, is not defined as such. In this work, Denning describes informally a mechanism in which variables and program statements have a security level. Those levels are ordered and form a security lattice. Denning gives a set of rules relating program structures and the appropriate security level of the program entity involved in those structures. She suggests to either statically assign a security level to variables and statements and then use the rules to check that those levels are appropriate, or assign security levels only to input variables and use directly the rules to compute appropriate security levels for the other program entity. This mechanism, presented in (Denning, 1976), is formalized in (Denning and Denning, 1977) as a "certification semantics".

Using *flow logic* (Nielson and Nielson, 1998), Clark et al. (2002) give a set of *acceptability rules*. Using those rules, it is possible to determine if $\widehat{\mathsf{R}}$ is an acceptable result of an information flow analysis of a given statement S. If the acceptability rules apply "$\widehat{\mathsf{R}} \models S$" then $\widehat{\mathsf{R}}$ is an over-approximation of the dependencies which may be created by the execution of S. The acceptability rules are not an analysis; however, Clark et al. prove the existence of a fix-point. It is then possible to use the rule as a static information flow analyses by fix-point computation. Other more standard and well studied formalisms have since been applied to the field of static secure information flow analysis.

Constructing static analyses for noninterference using abstract interpretation. Abstract interpretation (Cousot, 1996; Cousot and Cousot, 1977; Jones and Nielson, 1995) is a technique to derive a static analysis, sound by construction, from an instrumented semantics. For secure information flow, the semantics instrumentation consists usually in adding a label to values which reflects the set of inputs which have influenced the labeled value. The "sound by construction" feature of analyses derived by abstract interpretation is appealing. However, it remains that the soundness of the analysis relies on the soundness of the instrumented semantics. It is then still required to prove that the security labels computed by the instrumented semantics are sound. When it is not the case, for example in the work of Ørbæk (1995) (cf. Section 2.2.2), there is no more soundness guarantee for the derived analysis.

Mizuno and Schmidt (1992) present and prove the correctness of a “security flow control” algorithm for message based, modular systems. The compile-time analysis algorithm is an abstract interpretation of the denotational semantics (Schmidt, 1986; Stoy, 1977) of the language. The aim of the analysis is to accept programs for which no execution would create an information flow from a high level data to a low level one. They first develop a full path semantics ($\mathbf{C}_{full}$). This semantics evaluates in the same time the value of variables and their security level (for those with a dynamic security level). The evaluation generates equations and inequations of security levels which must be satisfiable for the program to be accepted. Then, they *stage* the semantics $\mathbf{C}_{full}$ and obtain two independent semantics: one for the evaluation of variables values ($\mathbf{C}_{exec}$), and one which approximates the security levels ($\mathbf{C}_{abs}$). They prove that $\mathbf{C}_{abs}$ gives a safe approximation of the results returned by $\mathbf{C}_{full}$.

Ørbæk (1995) provides two analyses, one of which is based on abstract interpretation, for the secure information flow problem of *integrity* (dual of *confidentiality*). Despite a mistake in its instrumented semantics (exposed on page 26), the proposed analyses are believed to be sound. Giacobazzi and Mastroeni (2004) use abstract interpretation to check or deduce information flow at a finer level. For example, that the parity of high input values will not interfere with the sign of any low output value. Using their technique, they are able to determine the most concrete (most precise) attacker for which a given program is secure or how much information flows from secret inputs to public outputs.

Constraint based static analyses are used for secure information flows verification. This type of analyses (Aiken, 1999; Nielson et al., 2005, Chapter 3) work in two steps:

1. first *generate* a set of constraints that must be true in order for a given program to respect a given property, or at least state some constraints that must be respected by any execution of the given program (using, for example, Hoare logic (Hoare, 1969));

2. then *solve* those constraints using theorem provers (Coq; PVS), model checkers (Bogor; Clarke et al., 2000) or dedicated constraint solving techniques.

Usually, for secure information flow analyses, the constraint solving problem concerns the assignment of security labels to variables such that at any execution step information flows to variables having a security level higher than or equal to the one of the origin of the flow. Another approach (Barthe et al., 2004), which does not use security labels, attempts to directly prove that, for any pair of executions, the values of public outputs are the same if the values of public inputs are also the same.

Guttman et al. (2005) want to verify that some particular information flows between specific endpoints go through trusted programs in a particular order. Those constraints are expressed using linear temporal logic (Kröger, 1987; Pnueli, 1977). Guttman et al. use model checking on an abstract model of the access control mechanism of SELinux (National Security Agency, 2007) to verify that a given configuration enforces the desired constraints on the information flows. Combining the type based approach with either theorem provers or model checkers, Terauchi and Aiken (2005) develop an analysis for termination insensitive noninterference. They first rephrase this property as, what they call, a *2-safety* problem. Such problems can be disproved by exhibiting only two executions for which a given property does not hold; whereas the original definition of noninterference is stated over all possible executions of a program. Then, after applying type based techniques to gain in precision, they use a theorem prover or a model checker on a *self-composition* (Barthe et al., 2004) of the original program. Naumann (2006) extends the approach to deal with heap objects and automatize some of the validation steps when using self-composition. Following a slightly different approach and using abstract interpretation techniques, Jacobs, Pieters, and Warnier (2005) combine a theorem prover (PVS) and some rewriting rules, derived from an abstract labeling function, to prove automatically if a program ensures confidentiality.

2.1.3 Extending the Secure Information Flow Property

Early work on secure information flow, such as (Volpano et al., 1996), used mainly a simple definition of this property for a sequential deterministic language. Such definitions are sometimes, depending on the observational capacity of the attacker, insufficient to certify that a given program respects a desired level of confidentiality. For that reason, some efforts have been put on extending the secure information flow property, and develop dedicated analyses, to take into consideration different qualities which may be required for the certification of some programs.

One of those qualities is *termination sensitivity*. Termination insensitive properties take into consideration only terminating executions. Therefore, termination insensitive definitions of secure information flow may be insufficient, depending on the level of confidentiality desired, to certify a program which is not guaranteed to terminate in any case. In order to answer this problem, some work involves a termination sensitive definition of secure information flow. For example, Volpano and Smith (1997a) describe a type system for an imperative language which prevents covert flows through the termination behavior of the program. The conclusion of their work is that guards of statements which can influence the termination behavior of a program (like loop statements) must depend on public data only in order for a program to be well typed; and then have secure information flows with regard to termination.

Another desirable feature for the definition of secure information flow is *nondeterminism* of the language on which the property applies. Early definitions of secure information flow (aka noninterference) are based on the assumption that the language is deterministic. It assumes that if the origin of an information flow has a constant value over some executions then the destination of this flow will have a constant value over those executions. This is not anymore the case with nondeterministic languages. Therefore, definitions of secure information flow which do not take into consideration nondeterminism have a high probability of rejecting a program which is perfectly safe but make use of nondeterministic features of the program. This concern gave raise to *possibilistic noninterference*. Regular noninterference requires that two executions started with the same public inputs have the exact same public outputs; whereas possibilistic noninterference requires that two executions started with the same public inputs can possibly have the same public outputs. Banâtre et al. (1994) propose a dependency analysis for a language incorporating a nondeterministic choice operator. Instead of using levels associated with variables and compute the final levels of outputs depending on the fixed levels of inputs, they determine at any step of the computation the initial values of which variables may influence the current value of every variable. It is then easy, knowing the security levels of inputs and outputs, to determine if the value of a public output can possibly be influenced by the value of a secret input. Joshi and Leino (2000) propose, for a nondeterministic language, a secure information flow approach based on a semantics characterization. Their definition is based on the semantics of two programs. The first one corresponds to the sequence composed of the program analyzed followed by an “havoc on h” function which *discards* (or resets) the values of secret variables. The second program is the sequence of the same “havoc on h” function followed by the first program. If both programs have the same semantics then the program analyzed (which is contained in the two programs compared) is secure with regard to confidentiality. As a final example, Sabelfeld and Sands (2001) give, for sequential nondeterministic programs, an extensional semantics-based formal specification of secure information flow constructed upon *partial equivalence relations* (Hunt and Sands, 1991).

Threads are also a feature which is not well handled by early definitions and analyses of secure information flow. The problem is that two independently-secure programs may not be secure anymore once run in parallel if they share some resources. Those resources can be, for example, shared variables or files on a hard drive. “Thread-unaware” analyses can not be used independently on both programs because there is no guarantee that shared resources will receive the same security levels during both analyses. Smith and Volpano adapted in 1998 their type system to a multi-threaded language. The schedulers taken into consideration are purely nondeterministic. Similarly, Boudol and Castellani (2001, 2002) develop a type system for multi-threaded programs and observe that any loop statement, whose guard is secret, must be followed by assignments to public variables only for the whole program to be considered secure.

Nondeterminism, either from a language construct or thread scheduling, creates a new type of covert flow. If possibilistic noninterference ensures that two executions started with the same public inputs can possibly have the same public outputs, it does not ensure that those outputs have the same occurrence probability in both executions. Therefore, by executing multiple times the same program which is possibilistic noninterfering, an attacker could potentially deduce the value of the secret inputs depending on the probability of the different possible public outputs. To deal with probabilistic attacks, the notion of *probabilistic noninterference* has been defined. It states that any possible public output has the same occurrence probability in any execution having the same public input. Volpano and Smith (1999) give a probabilistic noninterference type system for multi-threaded programs run with a probabilistic scheduler. They show that, to ensure probabilistic noninterference on a simple concurrent language, it is sufficient to force any conditional with secret guard to be evaluated atomically. Still using partial equivalence relations, Sabelfeld and Sands (2000) propose a scheduler-independent definition of probabilistic noninterference and use probabilistic-bisimulation-like relations to prove probabilistic noninterference.

Computation duration can also be a “vector” of insecure information flows. Take, for example, a program which branches on a secret value. If this secret value is true then the program does a simple computation. If the secret value is false then the program does a highly time-consuming computation. An attacker which is able to evaluate how long the computation lasted is then able to determine the value of the secret. In order to prevent such covert flows, the analysis used to certify a program must be *timing-aware*. Agat (2000) proposes an analysis, under the form of an instrumented type system, which is able to close some timing covert channels. His analysis is in fact a program transformation which succeeds only if the transformed program is "secure" with regard to a timing aware definition of secure information flow. In 2005, Hedin and Sands give a timing-aware end-to-end secure information flow analysis which is parametrized over the time model of the language and the algorithm used to check low-observational equivalence. The language, on which this analysis applies, is a subset of the Java bytecode which integrates objects and arrays but excludes exceptions and `jsr/ret` instructions.

The majority of analyses presented above analyze a program, multi-threaded or not, as a whole. That is as a complete unit in which every piece works together in order to fulfill a precise goal, and with no interaction with any other entity. In modern computing however the majority of programs are evaluated on a "computing base" running other, supposedly unrelated, programs as well. Some programs may even be distributed over many "computing bases". This gives rise to concerns about the observable impact a program may have on its environment (like owning the lock of a file), the level of secrecy of communication channels between the different parts of a program, or even the level of trust a data manipulated by a distributed program may have on the different "computing bases" running the program. Attempts to answer some of those concerns produced analyses for *distributed programs* and *system wide* analyses. Zdancewic, Zheng, Nystrom, and Myers attack in 2001 the problem of program distribution in the presence of mutual distrust of the "computing bases". The goal of this work is to develop an architecture which allows partitioning a program on different machines such that the secrets of any principal are preserved, except in case of failure of a machine trusted by the principal. Working on system wide analyses, Mantel and Sabelfeld (2001) propose a bisimulation-based timing-aware notion of security. This notion comes with a trace-based framework which integrates local computation security as well as security of their communications.

2.1.4 Extending the Language

Initial developments of new analyses for secure information flow usually take into consideration a core calculus. Its language is often limited to a simple sequential imperative language including conditionals, loops, and assignments. After an initial validation of the new techniques used in the analysis, it is frequent to see the development of works attempting to apply similar techniques to more and more expressive languages.

The first development, when the core calculus is not functional in essence, is the integration of *procedures* and *functions*. The inclusion of such structures requires the analysis to incorporate a notion of modularity or polymorphism (Cardelli and Wegner, 1985). Indeed, an efficient analysis can not assign, to data manipulated by a function, a fixed security level valid for every call site. Volpano and Smith (1997b) extend their type system of 1996 to first-order procedures. This is done using *let-polymorphism* (Milner, 1978). Heintze and Riecke (1998) develop a functional language, called *SLam calculus*, which supports functions as first-order values. They prove the soundness of the type system of SLam with regard to noninterference.

References, or *mutable states*, are also a feature which is problematic for information flow analyses. Indeed, with such a feature, the information contained in a container, or variable, can be modified without this container to be explicitly used. It makes it more difficult to associate a valid security level to variables.

Crary, Kliger, and Pfenning (2005) propose a security-oriented type system for a higher-order language with mutable state. They deal with mutation explicitly via the notion of *impurity*.

The two previous extensions are integrated in the notion of *objects*. This feature also adds the problem of dynamic call binding. It is more difficult to deduce at compile time which method will be evaluated by a given call, and then to track the information flows induced by this call. Barthe and Serpette (1999) develop and prove the correctness of an information flow analysis for an object calculus (Abadi and Cardelli, 1996, Chapter 8). Their analysis, which is purely syntactic, is based on a type system. Later, Banerjee and Naumann (2002) propose a type system guaranteeing secure information flow for a complete language similar to Java. In this work, a security level is associated to classes. Additionally, the type of a method reflects the impact of the method on the heap, from a security point of view.

Tracking the information flows in a program including *exceptions* is really difficult because it modifies the normal control flow of the program. However, there still exist some attempts at analyzing information flows in the presence of exceptions. Volpano and Smith (1997a) propose a simple security-oriented type system dealing with exceptions. However, this type system may be too restrictive. Myers (1999a) and Pottier and Simonet (2002b) propose more precise analyses, one of which is for an imperative language and the other one for a functional language.

The majority of static program analyses for secure information flow deal with a high level language. It is interesting to have such analyses for lower level languages (Barthe and Rezk, 2005). That way it is possible to certify a program without having access to the source code. The analysis of Hedin and Sands (2005) take into consideration a subset of Java bytecode. Dealing with all of Java bytecode, Genaim and Spoto (2005) propose a compositional information flow analysis. It encodes those flows using boolean functions. Barthe et al. (2006) connect, in a systematic way, works at source code level and bytecode level.

There also exist some real world languages. Jif (Myers et al., 2001) is an implementation of the real world language called JFlow (Myers, 1999a,b). This language is an extension of Java (Gosling et al., 2005). It enables the programmer to add information flow annotations which are statically checked. Those annotations are based on the *decentralized label model* of Myers and Liskov (1997, 1998). In this model, variables receive a label which describes allowed information flows among the principals of the program. Some dynamic tests are possible. This allows more flexibility when statically checking *noninterference* by a given program. After statically checking the annotations using a type system, the Jif compiler returns a standard Java program integrating the remaining dynamic tests. This program can then be compiled with a regular Java compiler. However, Myers (1999b) does not give a formal proof of correctness. Pottier and Simonet (2002a) discovered a couple of flaws in his type system. Yet, Jif has been used by others to develop secure prototypes of common softwares. For instance, Eswaran, Ongtang, and Hadka (2006a) implemented a secure web browser (Eswaran et al., 2006b) and McDaniel, Hicks, King, and Ahmadizadeh (2006a) built a secure email system called JPmail. However, in the opinion of Hicks, Ahmadizadeh, and McDaniel (2006a), developing applications in Jif is quite complex. They try to simplify it by proposing a policy store and a policy manager (Hicks et al., 2006b). They even develop an IDE for Jif called Jifclipse (McDaniel et al., 2006b).

Whereas JFlow is an imperative language, FlowCaml (Simonet, 2003a,b,c,d) is an extension of a functional language called Objective Caml (Leroy et al., 2004). It includes a constraint-based type system (and an inference engine) working with security labeled types (Pottier and Simonet, 2003a,b; Simonet, 2002a,b). This type system ensures noninterference for any well-typed program. There also exists an interpreter for FlowCaml. This interpreter dynamically types data, commands and functions which are successively evaluated. However, it types commands the same way the static analysis does, and thus, does not take into account any dynamic information. Therefore, it can not be called a dynamic analysis. For example, the interpreter merges the types of both branches of an **if**-statement, even if the value of the condition is known. In the following session:

```
# let x1 : !alice int = 42;;
val x1 : !alice int
# let x2 : !bob int = 11;;
val x2 : !bob int
# let c = true;;
val c : 'a bool
# if c then x1 else x2;;
-: [> !bob, !alice] int
```

the interpreter gives to the **if**-statement a security level which is an upper bound of the security levels `alice` and `bob` (`c` is considered to be public). However, as `c` is true, the commands evaluates to `x1` and so is only required to have a security level which is higher or equal to the security level `alice`.

2.1.5 Static Analyses Using Dynamic Tests

As stated by Vachharajani et al. (2004), the main drawback of static analyses, is that the specification of the security policy is not in the hand of the end user. A program is analyzed for a fixed security policy. If the security levels of the inputs and outputs are modified then the analysis must be run again. In order to circumvent this drawback, some static analyses have been developed to take into account some dynamic security tests. One branch of a conditional whose condition is a security test is then analyzed assuming the condition is true. The other branch is analyzed assuming the same condition is false.

Static Analyses Using General Purpose Security Tests

Stack inspection (Fournet and Gordon, 2002) is an access control mechanism of Java (Gong et al., 1997, 2003) and the .NET Common Language Runtime (Gough, 2001). This mechanism associates *permissions* to pieces of code. A method which owns a given permission can enable this particular permission. Enabled permissions can be checked at run time using the method `checkPermission` whose argument is a permission. This method returns true if and only if the permission given as argument is owned by the method *m* who called `checkPermission` and: this method *m* has enabled this permission, or the same permission check run by the caller of *m* would return true.

The static analysis developed by Banerjee and Naumann (2003, 2005) uses this access control mechanism as a dynamic security test. The language taken into account is a class-based object-oriented language integrating stack inspection, of course, and the main features of such languages, as dynamic binding and dynamically allocated mutable objects. The analysis is based on a type system. A method without any call to `checkPermission` has exactly one type. But methods with such calls have multiple types which depend on the permissions which must be enabled to execute some pieces of the method. When typing a caller method m which owns the set of permissions P, the types taken into consideration for methods called in m are those which require only permissions in P. They prove that well typed programs are secure with regard to information flows. A modular type inference mechanism for this type system has been proposed by Sun, Banerjee, and Naumann (2004). This work investigates and makes sense of a particular design pattern associated with an access control mechanism whose general impact on the security of programs is not easily understood.

Banerjee and Naumann (2004) have also adapted their previous work to the access control mechanism called *history-based access control* (Abadi and Fournet, 2003). The analysis is once again a type system ensuring noninterference of well typed programs. Similar to stack inspection, history-based access control allows to dynamically check that some permissions are enabled. However, stack inspection computes enabled permissions from the call chain (or call stack). Whereas, history-based access control computes enabled permissions from the call history, i.e. from all the methods which have previously been called either they have already returned or not. A permission is enabled if and only if the permission is owned by the currently executed method and, this method has enabled this permission or the same permission is enabled for the most recently called method. This access control mechanism is more secure with regard to confidentiality than stack inspection because it is more restrictive. However, it is itself a covert channel for information leakage. Therefore, the analysis must account for the access control mechanism itself, which is done by their type system.

Static Analyses Using Special Purpose Security Tests

The *decentralized label model* (Myers and Liskov, 1997, 1998), and therefore JFlow (Myers, 1999a) which is based on this model, also includes some dynamic features. This model associates a label to every variable. A label is composed of a set of atomic flow policies. Such a policy is a pair. The first element is the identifier of the principal who owns and states the policy. It is called the owner of the policy. The second element is a set composed of the identifiers of the principals which are allowed, by the owner of the policy, to read the data protected by this policy. This is the reader set of the atomic flow policy. From now on, the principal identifiers will simply be called principals (context will tell if we are talking about principals or principal identifiers). The readers designated by a given label l are those belonging to the intersection of the reader sets of all the atomic policies of l. In this model, principals are partially ordered, thus forming a principal hierarchy. If P_1 is higher than P_2 in the lattice, it is said that P_1 "acts for" P_2. P_1 is then allowed to do whatever P_2 is allowed. Principals who are effectively allowed to read a data whose label is l are those which can act for a reader designated by the label l. It is assumed that every principal can act for itself.

The first dynamic feature of the decentralized label model concerns labels. Variables can be of type `label`. Values of this type can be used as *near first-class* values or as a label for other values. However, variables of type `label` are immutable after initialization. They are thus only partially dynamic; yet, they allow some run time flexibility. For example, when using a file as input, it is possible to compute at run time the label given to the file descriptor by looking at the permissions registered by the operating system for the given file. However, the analysis is run at compile time and thus has no information on the label of the file descriptor. It is then likely that the analysis will not be able to certify the program. In order to give more information to the analysis, the programmer has to include some dynamic label tests under the form of "label cases" similar to the usual `switch` command. Such dynamic tests compare the label of the variable tested with a list of labels. To each label corresponds a statement. The statement which is executed is the one associated to the first label of a higher level than the one of the variable tested. Zheng and Myers (2004) develop a type system for a functional language having a similar feature. It is proved that well typed programs are noninterfering.

The other type of dynamic tests concerns the principal hierarchy. The language incorporates an "acts-for" conditional. The test of such a conditional enables the programmer to verify if one principal is allowed to act for another before executing a piece of code. This allows programs to be analyzed at compile time even if the complete run time principal hierarchy is not fully known. Tse and Zdancewic (2007) formalize this type of dynamic test for a λ-calculus. They develop and prove the soundness, with regard to noninterference, of a type system for their language. Hicks, Tse, Hicks, and Zdancewic (2005) propose a notion of *noninterference between updates*. This notion accounts for the modification of the principal hierarchy during the execution of a program. They also provide an analysis, based on a type system, which take into account this notion. The language, in which programs are written, allows tests of this principals hierarchy.

2.1.6 Conclusion

As seen in the current section, there have been lots of static analyses proposed for secure information flow. They cover a large spectrum of secure information flow properties going from the original termination insensitive definitions to, for example, timing aware analyses or system wide analyses. Expressiveness of the languages analyzed has also increased a lot since the early work of (Volpano et al., 1996).

One critique which could be made against those static analyses is that the majority of them are not flow sensitive. It implies a loss of precision. However, flow sensitivity could be quite easily added by transforming the analyzed programs into a Static Single Assignment form (SSA) (Cytron et al., 1991) prior to the static analysis. It works well for sequential programs, but it may not be really efficient for concurrent ones. A solution for concurrent programs may be to "sequentialize" them before applying the transformation to SSA form. However, type checking loses its appealing feature of modularity when applying this technique. The types given to the variables of a thread depend on other threads. If a single thread is modified then the whole program must be analyzed again. Moreover, the types computed are valid for a single thread interleaving, corresponding to the "sequentialization".

Another drawback inherent to static analyses, but which some analyses attempt to answer, is that they are not user-centric (Vachharajani et al., 2004). It means that they do not always allow policy modifications by end users. The run-time environment is not taken into account. However, as shown in Section 2.1.5, there are already ongoing works to solve this problem. Nonetheless, it remains that those analyses are run at compile time and thus out of reach of the majority of end-users. End-users still have to trust the origins or certifying entities of all the programs they use.

Finally, static analyses certify programs, in other words all of their executions. Therefore, the security level given to a variable must be valid for any possible execution of the program. For example, after executing the statement "**if** c **then** z **:=** x **else** z **:=** y **end**", the security level of z must reflect potential flows from c, x and y. z will then have a security level which is the least upper bound of those of c, x and y. However, for a precise execution and in addition to the flow from c, there is either a flow from x to z or from y to z; but not both in the same time. Therefore, if the security levels of x and y are quite far from each other in the security lattice, then it will drastically increase the security level of the data contained in z above what is strictly needed. Declassification (Sabelfeld and Sands, 2007) may be used to circumvent this problem. However it is not clear if it would be really practical. Another solution is to refine the notion of secure information flow to the level of executions and develop dynamic execution analyses instead of static program analyses.

2.2 Dynamic Information Flow Analyses

In parallel to static program analyses, dynamic execution analyses have been developed. However, less work in the domain of dynamic analyses has been done. Still, some promising dynamic analyses, or monitors, have been proposed. They are described either at the binary code (or machine) level, at the source code (or program) level and at the event (or system) level.

2.2.1 Analyses Described at the Binary Code Level

Describing a dynamic analysis at the binary level has a major advantage. It can be applied to any program without requiring the source code of the program. This explains the high interest for such analysis which are supported either by a hardware extension (Brown and Knight, 2001; Crandall and Chong, 2004; Suh et al., 2004), by a binary instrumentation (Newsome and Song, 2005; Qin et al., 2006) or by a mix of both (Vachharajani et al., 2004). Those analyses associate a security level, sometimes called *data mark*, to every storage location of the machine. Storage locations designate, for example, the registers, the program counter or every memory word.

The main drawback of description at this level is the difficulty to deal with implicit indirect flows (i.e. information flows created by pieces of code which *are not executed*). Therefore, several papers restrict themselves to tracking direct flows and, to some extent, explicit indirect flows (Crandall and Chong, 2004; Newsome and Song, 2005; Qin et al., 2006; Suh et al., 2004). This type of analysis is sometimes called *taint analysis*. It looks at the problem of security under the aspect of integrity and does not take care of information flowing indirectly through branching statements containing different assignments. Its aim is usually to prevent an attacker to gain control of a system by giving *spurious* inputs to a program which may be buggy but is not malicious. It is used to prevent attacks based, for example, on buffer overflow or format string vulnerabilities as those used by some worms or Trojan horses. Even if those analyses have some practical use, they are not fit to deal with confidentiality seen as secure information flow.

Yet, some binary code level dynamic analyses take into consideration implicit indirect flows (Brown and Knight, 2001; Fenton, 1974b; Saal and Gat, 1978; Vachharajani et al., 2004). Fenton (1974a,b) describes the addition of data marks to the abstract computer model of Minsky (1967). Data marks are fixed, except for the program counter's data mark which is computed dynamically. This means that storage locations can either contain only secret information or only public information. This feature increases the requirements on the size of the machine because, for example, it must contain a set of registers specifically dedicated to contain secret information and another distinct set of registers for public information. It also requires the existence of a distinct program (using different storage locations) for every possible configuration of security levels of inputs. Finally, it also requires the programmer or compiler to be aware of the security level of data manipulated at any time in order to be able to use storage locations with compatible data marks. Fenton (1974b) shows that, assuming public registers are accessible only at the end of the computation, its abstract machine is secure with the fixed data marks, but would not be with variable ones. Saal and Gat (1978) and Brown and Knight (2001) describe, without formally proving something similar to noninterference, an abstract machine in which some storage locations have a fixed data mark and others have a dynamically computed one.

The basic ideas behind the three works (Brown and Knight, 2001; Fenton, 1974b; Saal and Gat, 1978) are the same. When storing a value v to a fixed data mark storage location fl, the machine checks that the data mark of fl is higher or equal to the least upper bound of the data mark of v and the one of the program counter. If it is not the case then the storage operation is ignored, i.e. as if the operation was NOP. When storing a value v to a dynamic data mark storage location dl, the machine updates the data mark of dl to the least upper bound of the data mark of v and the one of the program counter. With just this mechanism, the program counter's data mark will monotonically increase at each conditional jump operation. It will then end up with the highest possible data mark, and the machine will not be able to compute anything useful. In order to allow the machine to safely decrease the data mark of the program counter and come back to a previous level, it is possible to push a pair, composed of the program counter's data mark and a return address, into a stack. Then, it is possible to pop this pair from the stack and set the program counter's data mark and value to those in the pair previously pushed into the stack. This feature allows a program to execute a procedure, whose behavior depends on a secret value, and still assign values to low level data mark locations after returning from the procedure call.

The machines of Fenton (1974b), Saal and Gat (1978) and Brown and Knight (2001) do not seem to take into consideration implicit indirect flows. So, how can they be secure from a point of view similar to noninterference? An implicit indirect flow from o to d exists whenever an assignment to d is not executed because of the value of o. As any flow, it causes insecurity whenever o has a higher level than d and d does not influence the low-observational behavior of the program the same way depending on the fact that the previous assignment has been executed or not. With the machines described previously, this type of flows does not exist on fixed data mark when o has a higher level than d because in that case, even if the assignment is executed, the value of d is not modified. So, the machine of Fenton (1974b), contrary to what Venkatakrishnan et al. (2006) state, securely handles implicit indirect flows. Problems arise when the destination of the flow has a dynamic data mark. It is then impossible to take into consideration the implicit indirect flow to increase *d*'s data mark to a level at least as high as the one of *o* because the machine does not see the operation causing this flow. To solve this problem, Saal and Gat (1978) propose to push into the stack the values and data marks of every dynamic data mark storage location whenever the program counter is pushed into the stack. Whenever the program counter is popped out of the stack, the values and data marks of every dynamic data mark storage location are reset to their values at the time the program counter was popped. Therefore, assignments, which have been executed or could have been executed since the last program counter push, do not have anymore influence on the dynamic data mark storage locations. It is then perfectly safe with regard to implicit indirect flows. However, this requires any useful output of a procedure to be stored in a fixed data mark storage location. This prevents easy reuse of procedures. It seems that in the work of Brown and Knight (2001) push and pop operations are not global over all dynamic data mark storage locations. It is possible to push and pop any dynamic data mark storage locations independently from others. This prevents a safe handling of implicit indirect flows. The following example emphasizes this problem. *x* is a dynamic data mark storage location. *h* is a fixed data mark storage location containing a secret, so its data mark is high. *pc* is the program counter. It is equal to the number of the line which is currently executed. After the execution of a line, *pc* is incremented by 1 and the line corresponding to the new value of *pc* is executed next. *PUSH* pushes its parameters into the stack. Its first parameter is a dynamic data mark storage location. *POP* pops a pair (*reg*, *val*) from the stack and sets *reg* to the value *val*. The data mark of *reg* is the least upper bound of the one of *val* and the one of the program counter when the pair was pushed into the stack. **OUTPUT** prints on the screen the value of its parameter if its data mark is low and the data mark of the program counter is high. Otherwise, it does not print the value. This behavior is the one expected for an output statement which is not allowed to display a secret value on a public output (the screen).

```
1   PUSH('pc', pc + 3)
2   if h then pc := pc + 1
3   x := 1
4   POP()
5   OUTPUT(x)
```

If *h* is false, then the value of *pc* at the end of the execution of line 2 is equal to 2. Therefore, the next line executed is the line number 3 and the assignment to *x* is executed. As the program counter's data is high after executing the conditional, *x*'s data mark is then high after executing the assignment. Then, line 4 is executed and the program's data mark is reset to low. Still, the output statement will print nothing because *x*'s data mark is high. However, if *h* is true then the value of *pc* at the end of the execution of line 2 is equal to 3. Therefore, the next line executed is the line number 4 and the assignment to *x* is not executed. Therefore, when executing the output statement, *x*'s data mark is still low and its value is printed on the screen. This simple example shows that it is possible to code a program whose low-observational behavior is influenced by a secret value when executed on the machine of Brown and Knight (2001).

Following a similar approach, McCamant and Ernst (2007) propose a *quantitative* dynamic information flow analysis for sequential programs. This analysis computes "an upper bound of the number of bits of information about the secret inputs present in the program's output". The definition of the number of bits leaked, based on the notion of entropy, is counter-intuitive in my opinion. It is nearer from an average number of bits leaked by any execution than an upper bound of the number of bits of secret information leaked by the analyzed execution. For example, let us study an execution of the following program. This program has one secret input (h) and one public input (l), each one coded on n bits. b is a temporary boolean coded on a single bit. *and*, *or* and *not* are the logical operations on single bit. For any variable v, v_i is the ith bit of the variable v.

```
i := 0; b := true;
while i < n do
  b = b and ((l_i and h_i) or not (l_i or h_i));
  i++;
done;
output b
```

For any execution of this program, the dynamic analysis of McCamant and Ernst computes an upper bound of 1 bit of information leaked. However, for any execution where h is equal to l, it is possible to deduce directly the precise value of the secret input h. Therefore, from such a single execution, an attacker learns the precise value of n bits of secret information; not only 1. In the same time, this first attempt at quantitative dynamic information flow analysis does not handle precisely implicit flows. Any program branching on a secret value is considered interfering, unless the branching statement occurred in an *enclosure region*. This construct (enclosure region) is similar to the stacking mechanism used by Saal and Gat (1978) and Brown and Knight (2001) in order to protect some computations on secret data. Variable values are saved when entering an enclosure region and restored when exiting this region except for designated variables. The values of those designated variables are considered secret when exiting the enclosure area. The static nature of this mechanism prevents a precise dynamic tracking of information flows. Even if this work is a promising first attempt at quantitative information flow *testing*, it can not be used as a noninterference monitor. The reason is that it does not *prevent* "bad" information flows. It just notifies the user of an approximation of the amount of information that has been leaked during the execution.

Vachharajani, Bridges, Chang, Rangan, Ottoni, Blome, Reis, Vachharajani, and August (2004) propose a mix of hardware extension and binary instrumentation. They propose an architectural framework, called RIFLE, to enforce confidentiality. As in previous work (Brown and Knight, 2001; Fenton, 1974b; Saal and Gat, 1978), the machine described extends every storage location, general purpose registers and memory words, with a data mark. Special purpose registers, now called data mark registers, containing data mark values are also added in order to be able to manipulate those data marks. The machine executes programs written in a special purpose instruction set. Compared to standard instruction sets, instructions are syntactically extended with the addition of a label. A label is a list of data mark registers. The semantics of instructions with regard to the values of general purpose registers and memory words is the same as with standard instruction sets. Additionally, the semantics of instructions is extended to update data marks in accordance with direct flows[1] and a given set of data marks. The semantics of an instruction modifying the value of the storage location dl with a value computed from the storage locations ol_i states that the new data mark of dl is the least upper bound of the data marks of ol_i and data marks contained into the label of the instruction. A special instruction is also added. This instruction stores into a data mark register the least upper bound of the data marks contained in some other data mark registers. The original binary code is instrumented by a binary translation. It replaces the standard instructions by their counterpart in the special purpose instruction set described above. It also encodes any indirect flows using the label of instructions. For every instruction i_b influencing the control flow of the program, the data mark of the register conditioning the behavior of i_b is stored in a data mark register r_m. This data mark is added to the label of every instruction which is control dependent on i_b. This instrumentation takes care of explicit indirect flows. In order to take into account implicit indirect flows, the data mark register r_b is added to the label of every instruction which potentially uses a value defined in an instruction control dependent on i_b. When an "insecure" output is detected, the machine terminates the execution of the program (Vachharajani et al., 2004, Sect. 4.1). A formal definition of "secure program", equivalent to noninterference, is given. However, Vachharajani et al. (2004) admit to fail to give a formal proof that an instrumented program running on their machine is always secure. Besides, as is explained in Section 2.2.4, the way the machine deals with insecure outputs create a new covert channel which is not taken into account by their framework. Therefore, RIFLE does not enforce their definition of secure programs.

2.2.2 Analyses Described at the Source Code Level

Describing a dynamic information flow, or dependency, analysis at the source code level is usually easier than describing it at the binary level. This is due to the fact that the control flow of the program is usually easier to understand at the source code level. However, as for dynamic analyses described at the binary level, there are few dynamic information flow analyses described at the source code level.

The majority of dynamic information flow analyses deal with a notion weaker than noninterference. Some of them deal with the notion of *taint*, which has been described in Section 2.2.1. For example, the language Perl includes a special mode called "Perl Taint Mode" (Birznieks, 1998; Wall et al., 2000). In this mode, the direct information flows originating with user inputs are tracked. It is done in order to prevent the execution of "bad" commands. However, this analysis does not take any indirect flows into consideration. Extending standard taint analyses, Xu, Bhatkar, and Sekar (2006) propose a taint analysis for C. Their analysis takes into account direct flows, as any taint analysis, but also some explicit indirect flows. However, because they claim it is not necessary for the type of attacks they want to prevent, their dynamic analysis does not take into consideration all explicit indirect flows or any implicit indirect flow. Recently, Lam and Chiueh (2006) proposed a framework for dynamic taint analyses which, however, does not take any type of indirect flow into consideration. The notion of *need* is similar to the notion of taint. A need analysis determines what parts of a "program" are needed, or required, to "evaluate" this program. For a

[1]Vachharajani et al. (2004) do not use the same naming conventions for flows. Flows, called *explicit flows* in their paper, are called *direct flows* in this document. Flows, called *implicit flows* in their paper, are called *indirect flows* in this document. This type of flows is partitioned into *explicit indirect flows* and *implicit indirect flows* in this document.

λ-term t_λ, a need analysis determines which sub-terms of t_λ are needed in order to be able to evaluate t_λ to normal form. Gandhe, Venkatesh, and Sanyal (1995) and Abadi, Lampson, and Lévy (1996) propose two dynamic need analyses based on labeled λ-calculi. However, such analyses do not have to track implicit indirect flows. Therefore, the dynamic analyses described at the source code level presented thus far are not powerful enough to be used for checking properties similar to noninterference.

Some instrumented semantics, used as the bases for the development of static analyses, are good candidates as dynamic information flow analyses. For example, Ørbæk (1995) gives an instrumented semantics of a first order language with pointers before extracting from it two static *trust* analyses. The notion of trust is equivalent to a stronger notion of taint. It aims at determining which output values can be trusted to be exact even if some input values are not trusted to be exact. The instrumented semantics is a standard semantics with the addition of label to values. The evaluation rules are accordingly modified in order for the labels to reflect the trustworthiness of their associated value. The instrumented denotational semantics can be seen as a dynamic analysis of `trust`. Ørbæk proves the correctness of its static analyses with regard to the labels computed by the instrumented semantics. However, he does not prove that the labels are correctly computed. His dynamic analysis takes into account explicit indirect flows, but not implicit indirect flows. In my opinion, a dynamic trust analysis should consider implicit indirect flows and be equivalent to a dynamic noninterference analysis. This opinion is motivated by the following program example: “`x := 0;` **while** `b` **do** `x :=1; b :=!b` **done**”. Consider an execution for which `b` is false, but is not trusted. The final value of `x` is `0`. The instrumented semantics considers the final value of `x` as trusted. The fact that the value of `b` is not trusted means that its exact value may be different. In other words, it is possible that for this precise execution the value of `b` should have been different. Therefore, the exact value of `b` is maybe `true` and, consequently, the exact final value of `x` may be `1`. Hence, when the value of `b` is `false` and is not trusted, the final value of `x` should not be trusted either. However, for any *implicit* indirect flow in an execution, there exists at least one execution for which this flow is an *explicit* indirect flow. And static analyses results must be valid for any possible executions of the analyzed program. Therefore, as the instrumented semantics takes into consideration explicit indirect flows, it is believed that the static analyses extracted from this semantics are correct.

Other approaches rely only on a program transformation. They will add variables in the program whose values reflect the security level of data contained in the original variables. Tests over those added variables are also inserted in order to dynamically enforce noninterference. It is then sufficient to prove that any transformed program is a noninterfering program to safely run the transformed version on any trusted machine. Venkatakrishnan, Xu, DuVarney, and Sekar (2006) propose such a transformation for a simple deterministic procedural language. As lots of dynamic analyses, it does not address information flow leaks resulting from timing, storage and termination channels. The transformation returns a program in which every original variables is associated a boolean variable which reflects its level of security. Each time an original variable is assigned the value of an expression e, its associated boolean variable is assigned a value reflecting the security level of the expression e and of the current program counter. The transformation takes in parameter a static analysis of *defined* variables. This analysis is used to insert assignments to the added boolean variables which reflect the effects of implicit indirect flows. The transformation also takes as parameter a security policy. It is used to determine at which program points dynamic tests must be inserted in order to enforce noninterference. The correction mechanism chosen is to stop the execution as soon as an output of a secret data on a public channel is detected. A security policy, defined as a typing environment, specifies which variables are considered as high inputs, low inputs or low outputs. Therefore, Venkatakrishnan et al.'s transformation is not user centric with regard to policy definition; i.e. the user can not decide at run time which inputs contain secret values and which outputs are publicly observable. With regard to theoretical results, they provide a semantics preservation theorem. However, this theorem is not really powerful as a monitor which would halt any execution does respect this theorem. Basically, the semantics preservation theorem states that a transformed program has the same semantics as the original one unless the monitor decides to modify its semantics. They also claim to prove that their transformation ensures confidentiality of secure data. However, their main theorem proves that their transformation conservatively tracks all types of information flows, but not that any execution of a transformed program respects their definition of noninterference. Therefore, their transformation can safely be used at compile time to test the noninterfering behavior of a program, but not at run time to enforce noninterference. Moreover, their transformation does not approximate information flows with the same precision in a branch which is executed and in a branch which is not executed. As explained by Le Guernic and Jensen (2005), this is a required feature in order to avoid creating a new covert channel and be able to dynamically enforce noninterference. This is also one of the main points of Chapter 5. As an example, consider the following program where *h* is a secret input, *l* is a public input, *x* is neither an input nor an output, and **output** writes its parameter on a public output channel:

```
x := l;
if h then
  if false then
    x := x
  else skip end;
else skip end;
output x
```

This program is obviously noninterfering. Venkatakrishnan et al. (2006) allows and even encourages optimizations of the static analysis given as parameter to the transformation in order to increase the overall precision of their dynamic analysis. The only constraint put on the analysis is that it must return a "conservative upper bound" of "all the variables that get assigned" in the piece of code analyzed. However, if the static analysis takes into account some simple information about dead code, then their transformation considers the final output as safe if *h* is false and unsafe if *h* is true. Therefore, their transformation will prevent the flow in the final output if and only if *h* is true. This behavior, outputting the value of *x* or doing something else, is publicly observable. Hence, their dynamic analysis transforms a noninterfering program into an interfering one. They later discuss possible optimizations. At least one of those optimizations (the usage optimization of the static analysis for branching statements) further increases the difference in the treatment of executed and non-executed branches. In fact, as explained in Section 2.2.4, because the dynamic correction mechanism chosen is unbalanced in the analysis of non-executed branches, correcting the aforementioned error is not sufficient to be able to enforce noninterference.

Similarly, Masri, Podgurski, and Leon (2005; 2004) present a dynamic information flow analysis for structured or unstructured programs. Their algorithm achieves a good level of precision for a quite complete language. However, their work focuses on dynamic slicing rather than on noninterference monitoring. It does not study deeply the dynamic correction of "bad" flows and lacks formal statements and proofs of the correctness of the correction mechanism. It suffers from misconception similar to those of the transformation of Venkatakrishnan et al. (2006). For example, the dynamic correction mechanism proposed is to stop the execution as soon as a potential flow from a secret data to a public *sink* is detected. Once again, as explained in Section 2.2.4, this is not an acceptable dynamic correction mechanism with regard to the analysis used for non-executed branches. It also claims to deal with multi-threaded languages, however it is unable to handle attacks similar the one exposed in Figure 4.6.

More recently, Shroff, Smith, and Thober (2007) proposed an interesting approach to handle *implicit* indirect flows (indirect flows whose assignments which create them are not executed, as opposed to *explicit* indirect flows). It relies on the simple idea that an implicit indirect flow can exist in an execution if and only if there exists another execution for which this indirect flow is explicit. Therefore, the proposed dynamic information flow analysis track direct flows and collects explicit indirect flows dynamically. The information collected about indirect flows is transferred from one execution to another using a cache mechanism. After an undetermined number of executions, the analysis will know about all indirect flows in the program and thus will then be sound with regard to the detection of all information flows. Using this approach they are able to handle a language including alias and method calls. However, as this fully dynamic analysis is not sound from the beginning it can not be used to *enforce* confidentiality, even if it can be used to detect "unsafe" information flows in a way similar to intrusion detection. Moreover, it is undecidable in general to know when enough executions have been run for the dynamic analysis to be sound with regard to noninterference detection. For example, it can be proved by reducing the Post Correspondence Problem (PCP) (Hopcroft et al., 2001; Post, 1946) to a noninterference analysis problem using the following program (the reduction algorithm).

```
if testPcpSolution(pcp, solution)
then x := solution else skip;
output x
```

In this program, *pcp* is an instance of PCP considered either as a public information. *solution* is a sequence of integers considered as a secret information. *testPcpSolution* is a function which returns true if and only if *solution* is a solution to the PCP instance *pcp*.

To circumvent this drawback, Shroff, Smith, and Thober propose to execute their dynamic analysis with a prior knowledge of all the indirect flows which is obtained by static analysis of the program. In such a case, handling alias and method calls becomes harder and the analysis has to be more conservative to remain sound. Moreover, as indirect flows are mainly handled in a static manner and only direct flows composed only of direct sub-flows are really handled dynamically, their analysis is not as flow-sensitive and path-sensitive as could be expected from its dynamic nature. For example, their analyses are unable to detect that any execution of the following program, where h is the only secret input, are noninterfering whenever the public input l is `false`.

```
if h then x := 1 else skip end;
if l then skip else x := 0 end;
output x
```

```
if h then
  if l then x := 1 else skip
else skip end;
output x
```

Additionally, alias being difficult to handle, their analysis must be quite conservative and is therefore not flow-sensitive for alias. It is, for example, unable to detect that the following program is perfectly safe. This program uses the standard notation for pointer instead of "deref" which is the notation used in their analysis.

```
if h then *x := 1 else skip end;
*x := 0;
output *x
```

Nevertheless, their proposed approach to handle indirect flows is interesting. It remains to improve the static analysis collecting indirect information flows and the way this information is used by the dynamic part of the analysis in order to benefit from the run-time nature of the overall analysis.

2.2.3 Analyses Described at the Event Level

At a higher level, some dynamic analyses of event or action sequences attempt to prevent information to flow to an unauthorized user in a large system serving multiple users. They analyze the sequence of events generated by the actions of the different users acting in parallel in the system under supervision. Those analyses are in charge of controlling user actions and prevent those actions that may break the secure information flow policies. Such analyses do not study program code, either source or binary. Those dynamic analyses take as inputs sequences of events carrying information at different security levels. The monitoring mechanism based on those analyses attempts to prevent occurrences of actions carrying information at a given security level to be observable at a lower security level. Those analyses have an interest by themselves. However, the security level associated to events is approximated by an external entity outside of the scope of the dynamic events analysis. Therefore by letting the event security levels be set by a dynamic program analysis and the interactions of those events be monitored by a dynamic event analysis, it is possible to completely monitor the information flows in a system executing different programs for different users.

One of the foundational dynamic information flow analysis of event sequences is due to Bell and LaPadula (1973a,b). They describe a mathematical model of computer system to study confidentiality when users — or subjects — read from and write to shared objects. The model includes a function giving security level to subjects and objects, a matrix of allowed accesses (given by a controller) and a relation showing the access really granted. Bell and LaPadula define what it means for a system to be secure in their model and what are the constraints on the system for it to be secure. They show that a system is secure with regard to confidentiality if it satisfies a given property called the $\star$-property. A system satisfies the $\star$-property if, for any subject, all the objects that it can write in have a higher level than the objects that it can read from. In other words, the security level of the information carried by a write action from a given user is approximated as the least upper bound of the security levels of objects that this user can read from. Then Bell and LaPadula establish some operation rules which answer to the access requests from the users depending on the state of the system and are $\star$-property preserving. It ends by giving some hints on a possible implementation of the model.

Earlier, Weissman (1969) describes a security control mechanism which dynamically computes the security level of newly created files. In this mechanism, a *security level history* is associated to each job running on the computer. This level monotonically increases as the job opens existing files. However, it is bound by the security level of the job itself. The security level history is used when labeling newly created files. In this mechanism, the security level of the information output by a given job is approximated as the least upper bound of the security level of previously accessed files. Extending this approach, Woodward (1987) presents its *floating labels* method. This method deals with the problem of over-classification of data in computer systems implementing the MAC security model. To improve on this problem, objects receive two types of security levels. One is the upper bound of the security level this object is allowed to contain. The other one is the security level of what it really contains. This latter level is dynamically computed using a mechanism similar to the security level history. Later, Haldar, Chandra, and Franz (2005) follow an approach similar to *security level history* to implement a system that adds dynamic computation of MAC levels of objects — individual units of run-time data storage — to an existing JVM. Whenever an object o_1 gets some information from another object o_2, its MAC level is increased so that it is at least as high as the MAC level of the object o_2. In his thesis, Rotenberg (1973) describes the hardware and software of the *Privacy Restriction Processor*, a machine similar to Weissman's work.

Nagatou and Watanabe (2006) propose a monitoring mechanism to detect and prevent unauthorized information flows through covert channels in a system serving multiple principals. For example, they try to prevent a principal to transmit sensitive data to a lower security level principal p_2 by deleting files for which p_2 has read access. The inputs to the monitoring mechanism are events. An event reflects an action that a principal is attempting. Each event is associated a security level which reflects the security level of the information carried by the action. It seems that the security level of an action is approximated by the clearance level of the user generating the action. A better precision could be achieved by a dynamic information flow analysis of the processes run by users. The definition of noninterference used is similar to Goguen and Meseguer's one. An action of security level l is noninterfering if and only if the system would have reacted the same way if all the previous actions whose security level is higher than l had not been executed. The monitoring mechanism is based on an emulator for every security level l which emulates the state of the system if only the actions of a security level lower or equal to l had been executed. An action of security level l is authorized if the system and the emulator for the level l react the same way to the action. As Vachharajani et al., Masri et al. and Venkatakrishnan et al., Nagatou and Watanabe makes the same erroneous assumption about dynamic correction of bad flows. As they assume the bandwidth of a covert channel to be narrow, they decide to simply terminate the receiver process of an unauthorized information flow. This process is the one generating the unauthorized action. However, this is a misleading assumption. If information flow from A to B is allowed but not the reverse, B can still send a secret integer value to A. For example, they can do so by applying the following protocol. A sequentially sends integers to B, starting at 0 and increasing the value by 1 each time. Once B receives an integer equal to the secret, it sends to A an acknowledgment. As the message sent by B to A depends on the secret, the monitor will then stop A. However, it is too late because A knows that the last value sent was the good one. In fact, with this protocol, the acknowledgment is sent using a covert channel which is not monitored. This covert channel is the behavior of the monitor itself.

2.2.4 Conclusion

There is less development of dynamic analyses for secure information flow than static ones. However, there are still some interesting proposals. They show that it is feasible to take into account implicit indirect flows in a dynamic analysis. In the majority of cases, this is done by incorporating the results of a simple static analysis either applied at run time or at compile time. It is then possible to dynamically track the information flows with a good level of precision. Compared to static analyses, it is easier with dynamic analyses to let the end-user decide the information flow policies to be enforced (Vachharajani et al., 2004). The information collected at run time enables the dynamic analyses to increase their precision (Masri et al., 2004). It is also possible with dynamic analyses to safely execute some noninterfering executions of a program which can not be proved to be noninterfering (Venkatakrishnan et al., 2006).

However, the majority of works on monitoring of program information flows fail to prove that they actually *enforce* an information flow property similar to noninterference. In fact, except for the noticeable exceptions of works by Fenton (1974b) and Shroff et al. (2007), the majority of them fail to achieve their goal. The reasons are common erroneous assumptions:

- that the only difficulty in information flow monitoring is the detection of the information flows,
- that an execution which is noninterfering in a standard environment is also noninterfering in the presence of any information flow monitoring mechanism,
- that the dynamic correction mechanism chosen has no impact on the information flows.

There are two important things to keep in mind when developing a secure information flow monitor. First, the precision of the dynamic analysis *must* be the same for any execution whose public inputs are the same. The correction of unauthorized flows has a publicly observable impact on the behavior of the program. Therefore, if the precision of the analysis is not the same for any execution whose public inputs are the same then the publicly observable behavior of the monitor is influenced by the secret inputs of the program. The second thing to keep in mind is that, when the control flow has been influenced by a secret input, the correction mechanism must have the same publicly observable behavior for an unauthorized flow which is executed than for an unauthorized flow which is not executed.

This last point is an error committed by the majority of works which use program interruption as dynamic correction mechanism of unauthorized flows, as the works by Masri et al. (2004); Vachharajani et al. (2004)

and Venkatakrishnan et al. (2006). The following program is a typical example for which such monitors are unable to enforce noninterference. In this example, h is a secret input, l is a public input and **output** outputs its parameter to a public channel.

```
if h then
  output h
else skip end;
output l
```

With a monitor which terminates a program just before the execution of a command which creates an unauthorized flow, this program does nothing if h is true and outputs the value of l if h is false. Therefore, the publicly observable behavior of a monitored execution of this program depends on the value of its secret input. This demonstrates that such monitors are unable to *enforce* a property similar to noninterference. To do so, it should terminate the program even if h is false, or ignore the output statement of h even if h is true.

Dynamic information flow analyses are a promising and interesting approach to the enforcement of confidentiality. This approach has its own advantages compared to static analyses. However, lots of previous information flow monitors fall short to achieve their goals. This demonstrates that it is a task more complicated than it seems at first. This is further emphasized by the fact that the peer-reviewed paper of Venkatakrishnan et al. (2006)fails to achieve its goals that it claims to prove. In order to be trusted by end-users and be able to compete with strongly proved static analyses, the development of secure information flow monitors must be backed up by a strong proof that the mechanism indeed *enforces* secure information flows. The remainder of this thesis proposes three monitors which differ in their precision or in the language features taken into consideration. All the monitors are proved to accurately enforce a given secure information flow property.

Chapter 3

Noninterference Monitoring

In contrast to static checking of noninterference, this chapter considers dynamic, automaton-based, monitoring of noninterference for a single execution of a sequential program. The monitoring mechanism is based on a combination of dynamic and static analyses. During program execution, abstractions of program events are sent to the automaton, which uses the abstractions to track information flows and to control the execution by forbidding or editing dangerous actions. The mechanism proposed is proved to be sound, any monitored execution is noninterfering; and to preserve some *safe* executions, including all executions of well typed programs (in the security type system of Volpano, Smith and Irvine).

The content of this chapter has been published in (Le Guernic et al., 2006a,b).

3.1 Introduction

The work presented in this chapter aims at protecting the values of secret data manipulated by programs from being revealed by the publicly observable behavior of those programs. We study a simple imperative sequential language with loops and outputs. The work proposed in this chapter can be extended to more complex languages: the main requirement is to have a precise enough knowledge of the control flow graph of the program. Chapter 4 extends it to a concurrent setting.

Before defining the monitoring mechanism and stating some of its properties, we present the language studied and outline the approach used.

3.1.1 The Language: Syntax and Semantics

The grammar of the studied language is given in Figure 3.1. $\mathbb{X}$ is the domain of variables and x stands for any element of this domain. The only constraints put on expressions (e) are that their evaluation must terminate, be deterministic and without side effects. Statements (S) are either sequences ($S\ ;\ S$), conditionals and while loops (B), or atomic actions (A). The output statement, "**output** e", is a generic statement used to represent any kind of public (or low) output, e.g., the action of printing the value of expression e on the terminal, producing a sound, or laying out a new window on the desktop. Only public outputs (i.e. outputs that are visible by standard users) are coded with the output statement; secret outputs are simply ignored. For example, sending a message m on a public network is represented by "**output** m", but sending an encrypted message n on a public network is abstracted by "**output** c", where c is a constant which emphasizes the fact that the content of an encrypted message cannot be revealed. Finally, sending a message on a private network, to which standard users do not have access, does not appear in the code of the programs studied.

$$
\begin{array}{rcl}
A & ::= & x \mathbf{:=} e \\
 & | & \mathbf{skip} \\
 & | & \mathbf{output}\ e \\
B & ::= & \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S\ \mathbf{end} \\
 & | & \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done} \\
S & ::= & S\ ;\ S\ \ |\ \ B\ \ |\ \ A
\end{array}
$$

Figure 3.1: Grammar of the language

For simplicity reasons, non-terminating executions are not in the scope of this work. Chapter 4 extends the work presented here by taking into account non-termination, among other features. This limitation allows the use of big-step operational semantics (also called *natural semantics* (Kahn, 1987)). This choice simplifies the proofs of the main theorems. The standard semantics of the language (Figure 3.2) is described using evaluation rules, written:

$$\sigma \vdash S \overset{o}{\Rightarrow} \sigma'$$

This reads as follows: statement S executed in state σ yields state σ' and *output sequence* o. Let $\mathbb{D}$ be the semantic domain of values. The domain of program states ($\mathbb{X} \to \mathbb{D}$) is extended to expressions, so that $\sigma(e)$ is the value of the expression e in state σ. An output sequence is a word in $\mathbb{D}^\star$. It is either an empty sequence (written ϵ), a single value (for example, $\sigma(e)$) or the concatenation of two other sequences (written $o_1\ o_2$).

3.1.2 The Monitoring Principles

Noninterference formalizes that there is no information flow from secret (or high) inputs to public (or low) outputs. For a given program $\mathtt{P}$, let $\mathcal{S}(\mathtt{P}) \subseteq \mathbb{X}$ be the set of variables whose initial values are the secret inputs. The only public output is the output sequence resulting from the execution. Contrary to the majority of works on noninterference, the values of the variables in the program state are never *directly* accessible (even at the end of the execution). Consequently, the values of the variables in the program state are never considered public outputs. We made this choice in order to map more closely the real behavior of an

$$\sigma \vdash x := e \overset{\epsilon}{\Rightarrow} \sigma[x \mapsto \sigma(e)] \qquad \text{(E}_O\text{-ASSIGN)}$$

$$\sigma \vdash \textbf{output}\ e \xRightarrow{\sigma(e)} \sigma \qquad \text{(E}_O\text{-OUTPUT)}$$

$$\sigma \vdash \textbf{skip} \overset{\epsilon}{\Rightarrow} \sigma \qquad \text{(E}_O\text{-SKIP)}$$

$$\frac{\sigma \vdash S^{\mathrm{h}} \overset{o^{\mathrm{h}}}{\Longrightarrow} \sigma' \qquad \sigma' \vdash S^{\mathrm{t}} \overset{o^{\mathrm{t}}}{\Longrightarrow} \sigma''}{\sigma \vdash S^{\mathrm{h}}\ ;\ S^{\mathrm{t}} \xRightarrow{o^{\mathrm{h}}\ o^{\mathrm{t}}} \sigma''} \qquad \text{(E}_O\text{-SEQ)}$$

$$\frac{\sigma(e) = v \qquad \sigma \vdash S^{v} \overset{o}{\Rightarrow} \sigma'}{\sigma \vdash \textbf{if}\ e\ \textbf{then}\ S^{\mathtt{true}}\ \textbf{else}\ S^{\mathtt{false}}\ \textbf{end} \overset{o}{\Rightarrow} \sigma'} \qquad \text{(E}_O\text{-IF)}$$

$$\frac{\begin{array}{c}\sigma(e) = \mathtt{true} \\ \sigma \vdash S^{1}\ ;\ \textbf{while}\ e\ \textbf{do}\ S^{1}\ \textbf{done} \overset{o}{\Rightarrow} \sigma'\end{array}}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ S^{1}\ \textbf{done} \overset{o}{\Rightarrow} \sigma'} \qquad \text{(E}_O\text{-WHILE}_{\mathtt{true}}\text{)}$$

$$\frac{\sigma(e) = \mathtt{false}}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ S^{1}\ \textbf{done} \overset{\epsilon}{\Rightarrow} \sigma} \qquad \text{(E}_O\text{-WHILE}_{\mathtt{false}}\text{)}$$

Figure 3.2: Standard semantics outputting the values of low-outputs

information system. It is rarely the case that the program states are observable by the low level users of an information system. Those users rather have access to the public outputs of the information system. All, and only, the values which should be visible to low users must be displayed using an output statement at the position inside the program where they are visible. At any point of the program where a value should be accessible by low users, an output statement for this value must be inserted. If the desired behavior is that low users have access to the values of low variables at the end of the execution, the value of those low variables must be output at the end of the program. This approach is equivalent to having a single public variable whose value is always accessible to low level users, not just at the end of the program. Every output statement would then correspond to an assignment to this public variable.

The main monitoring mechanism principle is based on information transmission notions of classical information theory (Ashby, 1956). Cohen states it as follows:

> "information can be transmitted from *a* to *b* over execution of *H* [(a sequence of actions)] if, by suitably varying the initial value of *a* (exploring the variety in *a*), the resulting value in *b* after *H*'s execution will also vary (showing that the variety is conveyed to *b*)." (Cohen, 1977, Sect. 3).

Hence, for preventing information flows from secret inputs to public outputs, the monitoring mechanism must ensure that variety — which can be seen as the property of variability or mutability — of the initial values of the variables in the set $\mathcal{S}(\mathtt{P})$ is not conveyed to the output sequence. This means that:

the monitoring mechanism, which works on a single execution, must ensure that even if the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ were different, the output sequence would be identical.

Hence, it will enforce that variety in $\mathcal{S}(\mathtt{P})$ is not conveyed to the output sequence.

The monitoring automaton has two jobs. The first is to track "variety," that is, to track entities (program variables, program counter, ...) having different values when the initial values of variables in $\mathcal{S}(\mathtt{P})$ are different. Its second job is to prevent conveying of variety to the output sequence, that is, to ensure that the output sequence would be identical for any execution having the same public inputs (the initial values of the variables not in $\mathcal{S}(\mathtt{P})$). To complete the first job, the states of the monitoring automaton are pairs. The first element of this pair is a set of variables. At any step of the computation, it contains all the variables that have "variety" (i.e., have a different value if the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ are different). The second element of the pair is a word in $\{\top, \bot\}^\star$ which tracks "variety" in the context of the execution (the value of the program counter). It can be seen as a stack. At the beginning of every conditional, the automaton pushes into the stack an element which reflects the "variety" in the expression of the conditional. At the end of every branching statement, the automaton pops one element from the stack. The second job (avoiding transfer of variety to the output sequence) is accomplished by authorizing, denying, or editing output statements depending on the current state of the monitoring automaton.

A common limitation. Note that the automaton does not hold information about actual values of the variables — it merely knows whether a variable *may* or *may not* have "variety". This feature prevents the automaton from detecting some "safe" executions. For example, consider the following program with two inputs, h (secret) and l (public).

```
x := l;
if h
  then x := 1
  else x := x / 2
end;
output x
```

Executions for which l is 2 are safe: whatever value h holds, any such execution outputs 1. However, these executions will be rejected by the automaton since it does not track the concrete values of variables. The variety in h is not conveyed to the output sequence, and there is no flow from h to the output. To discover this fact, however, the automaton requires information about the real values of variables. This is not the case with the work proposed here, as well as in the majority of works on noninterference. Therefore, the above example is out of reach of the proposed monitoring mechanism.

An uncommon benefit. The following is an example of program which the monitor can handle more precisely than static analyses. As above, the program has only two inputs, h (secret) and l (public), and publicly outputs the value of x at the end of the execution.

```
if l
  then x := 1
  else x := h
end;
output x
```

This program has some unsafe executions, those for which l is `false`, and some safe executions, those for which l is `true`. For unsafe executions, the monitoring mechanism presented in this chapter does not modify the values of the variables. At the end of any unsafe execution x is equal to h. So, it leaves the internal state of the program (the mapping form variables to values) as it should be. However, it outputs a default value instead of the value of x. In that manner, it prevents unsafe executions to reveal private data. On safe executions of this program, the monitor does not alter the behavior of the program (i.e. output 1). Thus, it leaves the safe executions unaltered.

Section 3.4.1 exhibits the potential of this monitoring mechanism. In order to formally demonstrate it, the next section gives a formal definition of the monitoring automaton and of the monitoring semantics.

3.2 Definition of the Monitoring Mechanism

The monitoring mechanism is divided into two main elements. The first is an automaton similar to *edit automata* (Ligatti et al., 2005). Inputs to the automaton are abstractions of the actions accomplished during an execution. The automaton tracks information flow and authorizes, forbids or edits the actions of the monitored execution to enforce noninterference. The second element of the monitoring mechanism is a semantics of monitored executions that merges together the behavior of the monitoring automaton and that of the standard output semantics in Figure 3.2. This section first describes the monitoring automaton and then the semantics of monitored executions.

3.2.1 The Automaton

The transition function of the automaton used to monitor an execution is independent of the program monitored. However, the set of states of the automaton, as well as its starting state, depends on the program monitored.

Let $A^\star$ be the set of all strings over the alphabet A. For any program $\mathtt{P}$, whose variables belong to $\mathbb{X}$ and the set of secret input variables is $\mathcal{S}(\mathtt{P})$ ($\mathcal{S}(\mathtt{P}) \subseteq \mathbb{X}$), the automaton enforcing noninterference is the tuple $\mathcal{A}(\mathtt{P}) = (Q, \Phi, \Psi, \delta, q_0)$ where:

- Q is a set of states ($Q \subset 2^{\mathbb{X}} \times \{\top, \bot\}^\star$),
- Φ is the input alphabet, constituted of abstractions of a subset of program events, specified below,
- Ψ is the output alphabet, constituted of execution controlling commands, specified below,
- δ is a transition function $(Q \times \Phi) \longrightarrow (\Psi \times Q)$,

- q_0, an element of Q, is the start state ($q_0 = (\mathcal{S}(\mathtt{P}), \epsilon)$).

Automaton states. An automaton state is a pair (V, w). $V \subseteq \mathbb{X}$ contains all the variables whose current value *may* have been influenced by the initial values of the variables in $\mathcal{S}(\mathtt{P})$; w, which belongs to $\{\top, \bot\}^\star$, can be seen as a stack that tracks variety in the context of the execution. In our approach, the context consists only in the program counter's value. If $\top$ occurs in w, then the statement executed belongs to a conditional whose test may have been influenced by the initial values of $\mathcal{S}(\mathtt{P})$. Hence the statement may not have been executed for a different choice of initial values of $\mathcal{S}(\mathtt{P})$.

For any given program, the number of states of this automaton is finite. As the number of variables used by a given program is finite, V is a finite set. Assuming that there are a bounded number of function calls, the length of w can be bounded. The maximum length of this word is equal to the maximum depth of conditionals. For example, with a program having no functions and a single conditional, the second element of a state is a word whose maximum length is 1. Hence, for any program $\mathtt{P}$ with a bounded number of function calls, the number of states of $\mathcal{A}(\mathtt{P})$ is finite.

Automaton inputs. The input alphabet of the automaton (Φ) consists of abstractions of events that occur during an execution. The input alphabet is composed of the following:

"`branch` e**"** is generated each time a branching statement has to be evaluated. e is the expression which (or whose value) determines the branch which is executed. For example, before the evaluation of the statement "**if** $x > \mathtt{10}$ **then** S_1 **else** S_2 **end**", the input `branch` $x > \mathtt{10}$ is sent to the monitoring automaton.

"`exit`**"** is generated each time a branching statement has been evaluated. For example, after the evaluation of the statement "**if** $x > 10$ **then** S_1 **else** S_2 **end**", the input `exit` is sent to the monitoring automaton.

"`not` S**"** is generated each time a piece of code, S, is not evaluated. It is sent just after the execution of the piece of code which has been executed instead of S. For example, the statement "**if** x **then** S_1 **else** S_2 **end**", with x being `true`, generates the automaton input `not` S_2 just after the execution of S_1.

"A**"** is any atomic action of the language (assignment, skip or output statement). Any such action is sent to the automaton for validation before its execution. The automaton will either allow or deny — i.e. skip — the execution of the action, or ask for another action to be executed instead of A.

For simplicity, we assume that the input events are generated dynamically by the interpreter that implements the monitoring semantics. However, in order to improve efficiency, a compiler can easily embed such events into a program's compiled code. Every conditional is preceded by a "`branch` e" event and followed by an "`exit`" event. Every branch of a conditional is followed by a "`not` S" event for every other branch S of the conditional. In fact, as the automaton applies a static analysis on S, the compiler would rather include an event incorporating the result of the static analysis on S. The execution of every action (assignment, skip or output statement) is conditioned by the answer of the security automaton.

Automaton outputs. Outputs are sent back from the automaton to the monitoring semantics in order to control the execution. The output alphabet (Ψ) is composed of the following:

"*ACK*" is used as answer for any input which is not an atomic action of the language. The automaton acknowledges the reception of information useful for tracking potential information flows but which does not require an intervention of the automaton.

"*OK*" is used whenever the monitoring automaton authorizes the execution of an atomic action.

"*NO*" is used whenever the monitoring automaton forbids the execution of an atomic action.

"*A*" is any atomic action of the language. This is the answer of the monitoring automaton whenever another action, A, has to be executed instead of the current one.

Automaton transitions. Figure 3.3 specifies the transition function of the automaton. A transition is written:

$$(q, \phi) \xrightarrow{\psi} q'$$

It reads as follows: in the state q, on reception of the input ϕ, the automaton moves to state q' and outputs ψ. Let $\mathcal{V}(e)$ be the set of variables occurring in e. For example, $\mathcal{V}(x+y)$ returns the set $\{x, y\}$. Let $\mathit{modified}(S)$ be the set of all variables whose value may be modified by an execution of S. This function is used in order to take into account the implicit indirect flows created when a branch is not executed. A formal definition of this function follows:

$$\begin{aligned}
&\mathit{modified}(x := e) = \{x\}\\
&\mathit{modified}(\mathbf{output}\ e) = \mathit{modified}(\mathbf{skip}) = \emptyset\\
&\mathit{modified}(S_1\ ;\ S_2) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2)\\
&\mathit{modified}(\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2)\\
&\mathit{modified}(\mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}) = \mathit{modified}(S)
\end{aligned}$$

Figure 3.3 shows that the automaton forbids (NO) or edits (**output** θ) only executions of output statements. For other inputs, it merely tracks, in the set V, the variables that may contain secret information (have *variety*), and it tracks, in w, the *variety* of the branching conditions.

Inputs "`branch` e" are generated at entry point of conditionals. On reception of such inputs in state (V, w), the automaton checks if the value of the branching condition (e) might be influenced by the initial values of $\mathcal{S}(\mathtt{P})$. To do so, it computes the intersection of the variables appearing in e with the set V. If the intersection is not empty, then the value of e *may be* influenced by the initial values of $\mathcal{S}(\mathtt{P})$. If this is the case then the automaton pushes $\top$ at the end of w; otherwise it pushes $\bot$. In either case, the automaton acknowledges the reception of the input by outputting ACK.

Whenever execution exits a branch — which is a component of a conditional c — the input "`exit`" is sent to the automaton. The last letter of w is then removed. As any such input matches a previous input "`branch` e" — generated by the same conditional c — which adds a letter to the end of w, the effect of "`exit`" is to restore the context to its state before the conditional was processed.

$$((V, w),\ \texttt{branch}\ e) \xrightarrow{ACK} (V, w\top) \qquad \text{iff } \mathcal{V}(e) \cap V \neq \emptyset \qquad \text{(T-BRANCH-high)}$$

$$((V, w),\ \texttt{branch}\ e) \xrightarrow{ACK} (V, w\bot) \qquad \text{iff } \mathcal{V}(e) \cap V = \emptyset \qquad \text{(T-BRANCH-low)}$$

$$((V, wa),\ \texttt{exit}) \xrightarrow{ACK} (V, w) \qquad \text{(T-EXIT)}$$

$$((V, w),\ \texttt{not}\ S) \xrightarrow{ACK} (V \cup \mathit{modified}(S), w) \qquad \text{iff } w \notin \{\bot\}^\star \qquad \text{(T-NOT-high)}$$

$$((V, w),\ \texttt{not}\ S) \xrightarrow{ACK} (V, w) \qquad \text{iff } w \in \{\bot\}^\star \qquad \text{(T-NOT-low)}$$

$$((V, w),\ \mathbf{skip}) \xrightarrow{OK} (V, w) \qquad \text{(T-SKIP)}$$

$$((V, w),\ x \mathrel{\mathbf{:=}} e) \xrightarrow{OK} (V \cup \{x\}, w) \qquad \text{iff } w \notin \{\bot\}^\star \text{ or } \mathcal{V}(e) \cap V \neq \emptyset \qquad \text{(T-ASSIGN-sec)}$$

$$((V, w),\ x \mathrel{\mathbf{:=}} e) \xrightarrow{OK} (V \setminus \{x\}, w) \qquad \text{iff } w \in \{\bot\}^\star \text{ and } \mathcal{V}(e) \cap V = \emptyset \qquad \text{(T-ASSIGN-pub)}$$

$$((V, w),\ \mathbf{output}\ e) \xrightarrow{OK} (V, w) \qquad \text{iff } w \in \{\bot\}^\star \text{ and } \mathcal{V}(e) \cap V = \emptyset \qquad \text{(T-PRINT-ok)}$$

$$((V, w),\ \mathbf{output}\ e) \xrightarrow{\mathbf{output}\ \theta} (V, w) \qquad \text{iff } w \in \{\bot\}^\star \text{ and } \mathcal{V}(e) \cap V \neq \emptyset \qquad \text{(T-PRINT-def)}$$

$$((V, w),\ \mathbf{output}\ e) \xrightarrow{NO} (V, w) \qquad \text{iff } w \notin \{\bot\}^\star \qquad \text{(T-PRINT-no)}$$

Figure 3.3: Transition function of monitoring automata

Inputs "`not` S" are generated at exit point of conditionals. It means that, due to the value of a previous branching condition, statement S has not been executed. This detects *implicit indirect flows*. On input "`not` S" in state (V, w), the automaton verifies whether S may have been executed with differing values for $\mathcal{S}(\mathtt{P})$. This is the case if the context of execution carries *variety* (i.e. if w does not belong to $\{\bot\}^\star$). Let (V', w') be the new state of the automaton. If the context carries *variety* then V' is the union of V with the set of variables whose values may be modified by an execution of S. Otherwise, nothing is done.

Atomic actions (assignment, skip or output) are sent to the automaton for validation before their execution. The atomic action **skip** is considered safe because the noninterference definition considered in this work is not time sensitive. Hence the automaton always authorizes its execution by outputting OK.

When executing an assignment (x **:=** e), two types of flows are created. The first is a direct flow from e to x. The second flow is an explicit indirect flow from the context of execution (i.e., the program counter) to x. For example, the execution of the assignment in "**if** b **then** x **:=** y **else skip end**" creates such a flow from b to x. Both forms of flows are *always* created when an assignment is executed. What is important is to check if secret information is carried by one of the flows (i.e., if variety in $\mathcal{S}(\mathtt{P})$ is conveyed by one of the flows). Hence, on input x **:=** e, the automaton checks if the value of the *origin* of one of those two flows is influenced by the initial values of $\mathcal{S}(\mathtt{P})$. For instance, if b is true in "**if** b **then** x **:=** y **else skip end**", y is the origin of a direct flow to x and b is the origin of an explicit indirect flow to x. The origin of the explicit indirect flow is influenced by $\mathcal{S}(\mathtt{P})$ only if w contains $\top$, meaning that the value of the condition of a previous (but still active) conditional was potentially influenced by the initial values of $\mathcal{S}(\mathtt{P})$. The origin of the direct flow is influenced by $\mathcal{S}(\mathtt{P})$ only if $\mathcal{V}(e)$ and V are not disjoint. If the value of e is influenced by the initial values of $\mathcal{S}(\mathtt{P})$ then at least one of the variables appearing in e has been influenced by $\mathcal{S}(\mathtt{P})$. Such variables are members of V. Let (V', w') be the new automaton state after the transition. If the origin of either the direct flow or the explicit indirect flow is influenced by the initial values of $\mathcal{S}(\mathtt{P})$, then x (the variable modified) is added to V: $V' = V \cup \{x\}$. On the other hand, if none of the origins are influenced by the initial values of $\mathcal{S}(\mathtt{P})$, then x receives a new value which is not influenced by $\mathcal{S}(\mathtt{P})$. In that case, V' equals $V \setminus \{x\}$. This makes the mechanism flow-sensitive.

The rules for the automata input, "**output** e," prevent bad flows through two different channels. The first one is the actual content of what is output. In a public context, ($w \in \{\bot\}^\star$), if the program tries to output a secret (i.e., the intersection of V and the variables in e is not empty), then the value of the output is replaced by a default value θ. This value can be a message informing the user that, for security reasons, the output has been denied. To do so, the automaton outputs a new output statement to execute in place of the current one. The second channel is the behavior of the program itself. This channel exists because, depending on the path followed, some outputs may or may not be executed. Hence, if the automaton detects that this output may not be executed with different values for $\mathcal{S}(\mathtt{P})$ (the context carries *variety*) then *any* output must be forbidden; and the automaton outputs NO.

3.2.2 The Semantics

The automaton alone does not enforce noninterference. It needs to receive inputs which abstract the events occurring during the execution. The outputs of the automaton must also be taken into consideration to control the execution of the program in order to enforce noninterference. The semantics linking the standard output semantics in Figure 3.2 with the monitoring automaton is given in Figure 3.4.

This semantics is described using evaluation rules written:

$$(q, \sigma) \Vdash S \overset{o}{\Rightarrow} (q', \sigma')$$

This reads: statement S executed in automaton state q and program state σ yields automaton state q', program state σ', and output sequence o. There are three rules for atomic actions: **skip**, x **:=** e and **output** e. There is one rule for each possible automaton answer to the action executed. Either the automaton authorizes the execution (OK), denies the execution (NO), or replaces the action by another one. The rules use the standard semantics (Figure 3.2) when an action must be executed. In the case where the execution is denied, the evaluation omits the current action (as if the action was "**skip**"). For the case where, on reception of input A, the monitoring automaton returns A', the monitoring semantics executes A' instead of A. Note from Figure 3.3, that A' can only be the action that outputs the default value (**output** θ).

For conditionals, the evaluation begins by sending to the automaton the input "`branch` e" where e is the test of the conditional. Then, the branch designated by e is executed (in the case of a while statement whose test is `false`, the branch executed is "**skip**"). The execution follows by sending the automaton input "`not` S" where S is the branch not executed (in case of a while statement whose test is `true`, what happens is equivalent to sending the automaton input "`not` **skip**"). Finally, the input "`exit`" is sent to the automaton and the execution proceeds as usual. In the case of a while statement with a condition equals to `true`, the execution proceeds by executing the while statement once again.

3.3 Example of monitored execution

Table 3.1 is an example of monitored execution. The monitored program is given in column "Program P". It has only two inputs: h, a private (or high) input, and l, a public (or low) input. The execution monitored is the one for which h equals `true` and l equals `22`. $\mathcal{S}(\mathtt{P})$, the set of secret inputs of P, is $\{h\}$. So the initial state of the automaton is $(\{h\}, \epsilon)$. Column "input" contains the inputs which are sent to the automaton. The two following columns contain the result of the automaton transition function: "output" contains the output sent back to the semantics, and "new state" shows the *new* internal state of the automaton *after* the transition. Finally, the last column shows the actions which are really fulfilled by the monitored execution.

In this example, there are only two alterations of the execution (on lines 5 and 8). The first occurs when the program attempts to output a value influenced by $\mathcal{S}(\mathtt{P})$ – **output** y. At this point of the execution, the value contained in y has been influenced by the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$. This is known because y belongs to the first element of the automaton state *before* execution of line 5. Consequently, the automaton disallows the output of this value. However, the fact of outputting something in itself is safe because the context of execution (the program counter) has not been influenced by $\mathcal{S}(\mathtt{P})$ (the second element of the automaton state belongs to $\{\bot\}^\star$). Hence, the automaton replaces the current action by an output action whose value is a default one (therefore not influenced by $\mathcal{S}(\mathtt{P})$). This value lets the user know that an output action has been denied for security reasons.

$$\frac{(q, A) \xrightarrow{OK} q' \qquad \sigma \vdash A \overset{o}{\Rightarrow} \sigma'}{(q, \sigma) \Vdash A \overset{o}{\Rightarrow} (q', \sigma')} \qquad (\mathrm{E}_{M(s)}\text{-}OK)$$

$$\frac{(q, A) \xrightarrow{A'} q' \qquad \sigma \vdash A' \overset{o}{\Rightarrow} \sigma'}{(q, \sigma) \Vdash A \overset{o}{\Rightarrow} (q', \sigma')} \qquad (\mathrm{E}_{M(s)}\text{-EDIT})$$

$$\frac{(q, A) \xrightarrow{NO} q}{(q, \sigma) \Vdash A \overset{\epsilon}{\Rightarrow} (q, \sigma)} \qquad (\mathrm{E}_{M(s)}\text{-}NO)$$

$$\frac{(q, \sigma) \Vdash S_1 \overset{o_1}{\Longrightarrow} (q_1, \sigma_1) \qquad (q_1, \sigma_1) \Vdash S_2 \overset{o_2}{\Longrightarrow} (q_2, \sigma_2)}{(q, \sigma) \Vdash S_1 \,;\, S_2 \overset{o_1\, o_2}{\Longrightarrow} (q_2, \sigma_2)} \qquad (\mathrm{E}_{M(s)}\text{-SEQ})$$

$$\frac{\begin{array}{c} \sigma(e) = v \qquad (q, \texttt{branch}\ e) \xrightarrow{ACK} q_1 \\ (q_1, \sigma) \Vdash S_v \overset{o}{\Rightarrow} (q_2, \sigma_1) \\ (q_2, \texttt{not}\ S_{\neg v}) \xrightarrow{ACK} q_3 \qquad (q_3, \texttt{exit}) \xrightarrow{ACK} q_4 \end{array}}{(q, \sigma) \Vdash \textbf{if}\ e\ \textbf{then}\ S_{\texttt{true}}\ \textbf{else}\ S_{\texttt{false}}\ \textbf{end} \overset{o}{\Rightarrow} (q_4, \sigma_1)} \qquad (\mathrm{E}_{M(s)}\text{-IF})$$

$$\frac{\begin{array}{c} \sigma(e) = \texttt{true} \qquad (q, \texttt{branch}\ e) \xrightarrow{ACK} q_1 \\ (q_1, \sigma) \Vdash S \overset{o_l}{\Rightarrow} (q_2, \sigma_1) \qquad (q_2, \texttt{exit}) \xrightarrow{ACK} q_3 \\ (q_3, \sigma_1) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o_w}{\Longrightarrow} (q_4, \sigma_2) \end{array}}{(q, \sigma) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o_l\, o_w}{\Longrightarrow} (q_4, \sigma_2)} \qquad (\mathrm{E}_{M(s)}\text{-WHILE}_{true})$$

$$\frac{\begin{array}{c} \sigma(e) = \texttt{false} \qquad (q, \texttt{branch}\ e) \xrightarrow{ACK} q_1 \\ (q_1, \texttt{not}\ S) \xrightarrow{ACK} q_2 \qquad (q_2, \texttt{exit}) \xrightarrow{ACK} q_3 \end{array}}{(q, \sigma) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{\epsilon}{\Rightarrow} (q_3, \sigma)} \qquad (\mathrm{E}_{M(s)}\text{-WHILE}_{false})$$

Figure 3.4: Semantics of monitored executions

On line 8, the program tries to output something while the current context of execution has been influenced by $\mathcal{S}(\mathtt{P})$. Hence, if the output occurs, the sequence generated by the execution is influenced by some secret values. Therefore the automaton denies any output; it does not even give another action to execute in place of the current one. The semantics does as if the action was "**skip**".

Static analyses reject the whole program. Notice that some executions of this program are interfering. Therefore, any sound static analysis for noninterference rejects this program. Users have then the choice to

	Program	Automaton:			Actions
	P	input	output	new state	executed
1	x := l + 3;	x := l + 3	*OK*	$(\{h\}, \epsilon)$	x := l + 3
2	**if** (x > 10) **then**	branch x > 10	*ACK*	$(\{h\}, \bot)$	
3	y := h;	y := h	*OK*	$(\{h,y\}, \bot)$	y := h
4	**output** x;	**output** x	*OK*	$(\{h,y\}, \bot)$	**output** x
5	**output** y;	**output** y	**output** θ	$(\{h,y\}, \bot)$	**output** θ
6	**if** (h) **then**	branch h	*ACK*	$(\{h,y\}, \bot\top)$	
7	z := 0;	z := 0	*OK*	$(\{h,y,z\}, \bot\top)$	z := 0
8	**output** x	**output** x	*NO*	$(\{h,y,z\}, \bot\top)$	
9	**else** x := 1	not x := 1	*ACK*	$(\{h,y,z,x\}, \bot\top)$	
10	**end**	exit	*ACK*	$(\{h,y,z,x\}, \bot)$	
11	**else skip**	not **skip**	*ACK*	$(\{h,y,z,x\}, \bot)$	
12	**end**	exit	*ACK*	$(\{h,y,z,x\}, \epsilon)$	

Table 3.1: Example of the automaton evolution during an execution.

throw away the program or run it without any guarantees regarding the security of their private data — more accurately, with a hint that the confidentiality of their private data will not be respected.

Using the monitoring mechanism proposed in this chapter, the user can run the program safely, with regard to the confidentiality of its private data. Section 3.4.1 proves that the confidentiality of its private data will be respected by any monitored executions. The user will obtain a normal output whenever l is lower or equal to 7 and, in other cases, a degraded but safe output.

3.4 Properties of the Monitoring Mechanism

The preceding sections gave a formal definition of the monitoring mechanism and an example of its behavior. We now consider the "efficiency" of the monitoring mechanism with regard to the standard notions of soundness and completeness — or transparency, for monitors.

A first theorem proves that any monitored execution is a noninterfering execution. This proves the soundness of the monitoring mechanism.

The problem of noninterference is undecidable. It is then impossible to achieve completeness. However, a second theorem proves that the monitor is transparent — does not alter observable behavior — for a non-trivial set of executions.

Complete proofs of these theorems can be found in Appendix A.

3.4.1 Soundness

The soundness property of the monitoring mechanism is based on the notion of noninterference between the secret inputs and the output sequence of an execution. An execution is considered *safe* — knowing that the program's source is public — if and only if it does not convey the variety in its secret inputs to the output sequence, that is, if the secret inputs have no influence on the execution's outputs.

Definition 3.4.1 (Output of a monitored execution: $[\![\mathtt{P}]\!]\sigma$).
For all programs $\mathtt{P}$*, with set of secret inputs* $\mathcal{S}(\mathtt{P})$*, and value store* σ*,* $[\![\mathtt{P}]\!]\sigma$ *is the output sequence obtained via the monitored execution of* $\mathtt{P}$ *in the initial state* $((\mathcal{S}(\mathtt{P}), \epsilon), \sigma)$*. This definition is formally stated as follows:*

$$[\![P]\!]\sigma = o \textit{ if and only if } \exists\, q', \sigma' : ((\mathcal{S}(P), \epsilon), \sigma) \Vdash P \overset{o}{\Longrightarrow} (q', \sigma')$$

Let $\overset{X}{=}$ be the equivalence relation between value stores defined as follows: $\sigma_1 \overset{X}{=} \sigma_2$ if and only if σ_1 and σ_2 are indistinguishable for X, i.e., σ_1 and σ_2 associate the same value to every variable in X. Let X^c be the complement of set X in $\mathbb{X}$ ($X^c \cup X = \mathbb{X} \ \wedge\ X^c \cap X = \emptyset$). The following theorem states that every monitored execution is *safe*, i.e. it is noninterfering.

Theorem 3.4.1 (Soundness: monitored executions are noninterfering).
For all programs $\mathtt{P}$*, whose set of secret inputs is* $\mathcal{S}(\mathtt{P})$*, and value stores* σ_1 *and* σ_2*,*

$$(\ \sigma_1 \overset{\mathcal{S}(P)^c}{=} \sigma_2 \ \wedge\ [\![P]\!]\sigma_i \textit{ defined for } i \in \{1,2\}\ ) \ \Rightarrow\ [\![P]\!]\sigma_1 = [\![P]\!]\sigma_2$$

Proof sketch. The proof — which follows directly from lemma A.1.6 on page 134 — goes by induction on the derivation tree of $[\![\mathtt{P}]\!]\sigma_1$. It relies on the fact that, after any "step" in the evaluation of $[\![\mathtt{P}]\!]\sigma_1$ and "equivalent step" in the evaluation of $[\![\mathtt{P}]\!]\sigma_2$, the automaton states of both executions are equal and the value stores are indistinguishable for the complement of the first element of the automaton states.

3.4.2 Partial Transparency

Transparency is a common notion in the domain of monitoring. A transparent monitor is one that has no observable effect from the user's point of view. Using the monitor presented in this chapter, any terminating monitored execution is non-interfering. However, to achieve this goal, the monitor sometimes modifies the output sequence of the execution. The sequence of outputs resulting from the execution of program $\mathtt{P}$ with initial state σ might differ according to the semantics used, the standard one (Figure 3.2) or the monitoring semantics (Figure 3.4). Theorem 3.4.2 defines a subset of the set of noninterfering executions on which the monitoring mechanism has no impact. It shows that the mechanism proposed in this chapter preserves the output sequence of any execution of a program which is well typed under a security type system similar to the one of Volpano et al. (1996). In other words, the monitor is transparent for any program which is well typed under the type system of Volpano et al. (1996).

Figure 3.5 shows the security type system. It is the same one as that of Volpano et al. (1996) except for a small modification of the typing environment and the addition of a rule for output statements (which are not in the language of (Volpano et al., 1996)). The typing environment, γ, prescribes types for identifiers and is extended to handle expressions. $\gamma(e)$ is the type of the expression e in the typing environment γ. The lattice of types used has only two elements and is defined using the reflexive relation $\leq$ ($L \leq H$). L is the type for public data and H the type for private data. A program $\mathtt{P}$ is well typed if it can be typed under a typing environment γ in which every secret input is typed secret (i.e. $\forall x \in \mathcal{S}(\mathtt{P}), \gamma(x) = H$).

$$\frac{\gamma(e) = \tau' \qquad \tau' \leq \tau}{\gamma \vdash e : \tau} \qquad \text{(T-EXP)}$$

$$\frac{\gamma(x) = \tau' \qquad \gamma \vdash e : \tau' \qquad \tau \leq \tau'}{\gamma \vdash x \mathbin{\mathbf{:=}} e : \tau\ \text{cmd}} \qquad \text{(T-ASSIGN)}$$

$$\frac{}{\gamma \vdash \textbf{skip} : \tau\ \text{cmd}} \qquad \text{(T-SKIP)}$$

$$\frac{\gamma \vdash e : L}{\gamma \vdash \textbf{output}\ e : L\ \text{cmd}} \qquad \text{(T-PRINT)}$$

$$\frac{\gamma \vdash S_1 : \tau\ \text{cmd} \qquad \gamma \vdash S_2 : \tau\ \text{cmd}}{\gamma \vdash S_1\ ;\ S_2 : \tau\ \text{cmd}} \qquad \text{(T-SEQ)}$$

$$\frac{\gamma \vdash e : \tau' \qquad \gamma \vdash S_1 : \tau'\ \text{cmd} \qquad \gamma \vdash S_2 : \tau'\ \text{cmd} \qquad \tau \leq \tau'}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end} : \tau\ \text{cmd}} \qquad \text{(T-IF)}$$

$$\frac{\gamma \vdash e : \tau' \qquad \gamma \vdash S : \tau'\ \text{cmd} \qquad \tau \leq \tau'}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} : \tau\ \text{cmd}} \qquad \text{(T-WHILE)}$$

Figure 3.5: The type system used for comparison

The problem of noninterference is neither dynamically nor statically decidable. Consequently, the monitoring mechanism proposed in this chapter is not complete or transparent. However, Theorem 3.4.2 states that the monitoring mechanism does not alter executions of well typed programs. To show that the inclusion is strict, consider the following program: `x := h; x :=0;` **output** `x`. `h` is the only secret input. Every execution is noninterfering. But as the type system is flow insensitive, this program is ill-typed. However, the monitoring mechanism does not interfere with the outputs of this program while still guaranteeing that any monitored execution is noninterfering.

Theorem 3.4.2 (Pseudo-transparency: monitoring preserves *type-safe* programs).
For all programs P with secret inputs $\mathcal{S}(P)$, typing environments γ with variables belonging to $\mathcal{S}(P)$ typed secret, types τ, and value stores σ and σ',

$$\left.\begin{array}{r} \gamma \vdash P : \tau\ cmd \\ \sigma \vdash P \overset{o}{\Rightarrow} \sigma' \end{array}\right\} \Rightarrow [\![P]\!]\sigma = o$$

Proof sketch. The proof — which follows directly from lemma A.2.5 — goes by induction on the derivation tree of the unmonitored evaluation of P. It relies on the fact that the unmonitored evaluation of any well typed command is matched by an equivalent monitored evaluation.

3.5 Conclusion

This chapter addresses the security problem of confidentiality from the point of view of noninterference. It presents a monitoring mechanism enforcing noninterference of any execution. This monitoring mechanism is based on a semantics that communicates with a security automaton. During the execution, the semantics generates automaton inputs abstracting the events that occur. The automaton tracks the flows of information between the secret inputs and the current value of the program's variables. It also validates the execution of atomic actions (mainly outputs) to ensure confidentiality of the secret inputs.

Because the proposed monitoring mechanism enforces noninterference, it significantly differs from standard monitors (Schneider, 2000; Viswanathan, 2000). Usually, monitors are only aware of statements which *are* really executed. With the mechanism proposed, when exiting a conditional, the branch which has *not* been executed is analyzed. This takes into account implicit indirect flows between the test of a conditional and those variables whose values would be modified by the execution of the branch which is not executed. As noticed — but not elaborated on — by Vachharajani et al. (Vachharajani et al., 2004, Sect. 4.2.2), this feature is required in order to enforce noninterference.

Section 3.4.1 shows that any monitored execution is noninterfering. Thus a user having access to the low outputs of a monitored execution is unable to deduce anything about the values of the secret inputs. In addition, the monitoring mechanism is proved not to alter executions of a program which is well typed under a type system similar to the one of Volpano, Smith and Irvine (Volpano et al., 1996).

Chapter 4 addresses the extension of the monitoring mechanism to a concurrent setting that includes a synchronization command. The goal is to achieve more precision than a type system equivalent to, e.g., the concurrent one due to Smith and Volpano (Smith and Volpano, 1998).

Chapter 4

Noninterference Monitoring of Concurrent Programs

Chapter 3 presented an automaton-based noninterference monitoring mechanism for sequential programs. The current chapter extends this work to a concurrent setting. Monitored programs consist of a set of threads running in parallel. Such threads run programs similar to those of Chapter 3 with the inclusion of a synchronization command. The monitoring mechanism is still based on a security automaton and on a combination of dynamic and static analyses. As in Chapter 3, the monitoring semantics sends abstractions of program events to the automaton, which uses the abstractions to track information flows and to control the execution by forbidding or editing dangerous actions. All monitored executions are proved to be noninterfering (soundness) and executions of programs that are well typed in a security type system similar to the one of Smith and Volpano (1998) are proved to be unaltered by the monitor (partial transparency).

The content of this chapter has been published in (Le Guernic, 2007a,b).

4.1 Introduction

Similar to Chapter 3, the monitoring mechanism presented in this chapter aims at protecting the confidentiality of private data manipulated by any program. As previously, the notion of confidentiality is based on the property of noninterference between the values of private inputs and the sequence output by the program during its execution. Whereas the previous chapter deals with deterministic sequential programs, the monitor presented in this chapter deals with nondeterminism and concurrency. Programs, in this chapter, are pools of sequential programs described using a language equivalent to the one of Chapter 3 except for the addition of a synchronization command similar to the one used in (O'Hearn, 2004). The other main difference lies in the semantics used: Chapter 3 uses a big-step (or natural (Kahn, 1987)) semantics, and therefore it takes into account only terminating executions. Use of a big-step semantics is inconvenient when dealing with thread interleaving and nontermination. Hence, this chapter uses a small-step semantics. This modification allows the work presented in this chapter to additionally take into account nonterminating executions.

Trying to enforce dynamically the confidentiality of private data when dealing with concurrency and synchronization is very difficult. Synchronization commands may prevent some output sequences to occur. Therefore, as demonstrated by the example 4.5 on page 58, when the execution of a synchronization command is conditioned by a private data, the value of this private data is revealed whenever the program outputs a sequence which can not occur if the synchronization command is executed. More insidious, a valid mechanism for sequential programs protecting private data stored in variables may not be usable in a concurrent setting; even if it is still efficient at its work with concurrent programs. When handled inaccurately, the mechanism protecting private data stored in global variables can create a new covert channel revealing the value of other private data. An attacker making use of this subtlety is given in Figure 4.6(b) on page 59. Nondeterminism in thread interleaving also brings its share of difficulties. In Chapter 3, programs were deterministic and therefore always output the same sequence of public values if executed with the same inputs. This is not the case in the multi-threaded setting. This has an impact on the definition of noninterference which must be used. The reason is that even noninterfering programs do not output the same sequence whenever they are executed with the same public inputs. Hence, the monitor will not ensure that any execution started with the same public inputs generates the same public output sequence. Instead, it ensures that, for any execution started with the same public inputs, there exists a thread interleaving which generates a public output sequence, or a prefix of it, equal to the sequence of public outputs generated so far by the currently monitored execution. Therefore an attacker can not deduce any information about private input. This is due to the fact that, with the scheduler used for a monitored execution, an attacker does not know which precise thread interleaving has been used.

This chapter follows a presentation similar to Chapter 3. It starts by presenting the language used: its syntax (Section 4.1.1) and semantics (Section 4.1.2). It also introduces, in Section 4.1.3, the principles used by the monitoring mechanism presented in this chapter. Section 4.2 defines formally the security automaton which is at the heart of the monitoring mechanism. The following section characterizes the monitoring semantics which links the monitoring automaton previously presented with the standard semantics given in Section 4.1.2. Then, comes an example in Section 4.4, which examines the evolution of the monitoring automaton during an execution. Finally, before concluding in Section 4.6, two main properties of the monitor, soundness and partial transparency, are proved in Section 4.5.

4.1.1 The Language: Syntax

A concurrent program is a pool of threads (Θ). Before execution, each thread contains a sequential program. Such a pool is formally defined as a partial function from integers to sequential programs: $\Theta(i)$ is the i^{th} sequential program of the pool. The grammar of the sequential programs is given in Figure 4.1. In order to gain simplicity while describing the semantics, the grammar is split into four different blocks. In this grammar, ⟨*ident*⟩ stands for a variable name (or identifier). As in any programming language, variables have an associated value which can be updated by an assignment. In the concurrent setting, each variable has also an associated unique lock. Variable locks can be acquired and then released by threads. Whenever a thread has acquired a lock but not released it yet, this thread is said to *own* this lock. A given lock can be owned by at most one thread at any time. ⟨*expr*⟩ is an expression of values and variable names. The expressions taken into consideration in this language are such that their evaluation always terminates, is deterministic – their evaluation in a given program state always result in the same value — and is free of side effects — their evaluation has no influence on the program state.

$$
\begin{array}{rcl}
\langle \mathit{action} \rangle & ::= & \langle \mathit{ident} \rangle \ \mathbf{:=} \ \langle \mathit{expr} \rangle \\
 & | & \mathbf{output} \ \langle \mathit{expr} \rangle \\
 & | & \mathbf{skip} \\
\langle \mathit{control} \rangle & ::= & \mathbf{if} \ \langle \mathit{expr} \rangle \ \mathbf{then} \ \langle \mathit{prog} \rangle \ \mathbf{else} \ \langle \mathit{prog} \rangle \ \mathbf{end} \\
 & | & \mathbf{while} \ \langle \mathit{expr} \rangle \ \mathbf{do} \ \langle \mathit{prog} \rangle \ \mathbf{done} \\
 & | & \mathbf{with} \ \langle \mathit{ident\text{-}set} \rangle \ \mathbf{when} \ \langle \mathit{expr} \rangle \ \mathbf{do} \ \langle \mathit{prog} \rangle \ \mathbf{done} \\
\langle \mathit{com} \rangle & ::= & \langle \mathit{action} \rangle \quad | \quad \langle \mathit{control} \rangle \\
\langle \mathit{prog} \rangle & ::= & \langle \mathit{prog} \rangle \ ; \ \langle \mathit{prog} \rangle \quad | \quad \langle \mathit{com} \rangle
\end{array}
$$

Figure 4.1: Grammar of the language for sequential programs in the concurrent setting

A sequential program (⟨*prog*⟩) is either a sequence of sequential programs or a command (⟨*com*⟩). Actions (⟨*action*⟩) and control commands (⟨*control*⟩) are the only two types of commands. An action is either an assignment of the value of an expression (⟨*expr*⟩) to a variable (⟨*ident*⟩), an output of the value of an expression, or a skip (which does nothing). A control command is either a conditional executing one program — out of two — depending on the value of an expression, a loop executing a program as long as the value of a given expression remains `true`, or a synchronization command (**with** ⟨*ident-set*⟩ **when** ⟨*expr*⟩ **do** ⟨*prog*⟩ **done**). This command is executed only if no other thread owns one of the locks of the given set of variables (⟨*ident-set*⟩) and the value of the expression (⟨*expr*⟩) is `true`. Otherwise, the thread executing this synchronization command is blocked. Once the command is executed, it is replaced by the program it encompasses (⟨*prog*⟩) and its thread acquires the locks of ⟨*ident-set*⟩.

The syntax and semantics of the synchronization command comes from previous works by Hoare (1972) and Brookes (2004). It has been chosen for its ability to encode easily lots of different synchronization constructions. It allows the monitoring mechanism to deal with only one synchronization construction while keeping the language expressive with regard to synchronization. Figure 4.2 shows how to encode Java's "synchronized" command and standard semaphores using the synchronization command in this chapter.

The synchronization command complicates the work of the monitor. As Figure 4.2 indicates, it is not infrequent for the expression in a synchronization command to be simply the constant "`true`". In such cases, the monitor can deal with synchronization commands more accurately. This explains why the special case,

$$\begin{array}{rcl}\text{synchronized}(x)\{P\} & \equiv & \textbf{with } x \textbf{ when } \texttt{true} \textbf{ do } P \textbf{ done} \\ \text{P}(s) & \equiv & \textbf{with } s \textbf{ when } s > 0 \textbf{ do } s \textbf{ := } s - 1 \textbf{ done} \\ \text{V}(s) & \equiv & \textbf{with } s \textbf{ when } \texttt{true} \textbf{ do } s \textbf{ := } s + 1 \textbf{ done}\end{array}$$

Figure 4.2: Encoding of different synchronization commands

where the expression is the constant "`true`", is taken into account in the monitoring mechanism exposed in this chapter (cf the definition of *stoppable*(**with** $\bar{x}$ **when** e **do** S **done**) on page 63).

4.1.2 The Language: Standard Semantics

During the execution, a thread does not necessarily contain a sequential program ($\langle prog \rangle$) as defined in Figure 4.1. In fact, a thread under execution contains an *execution statement* ($\langle exec\text{-}stat \rangle$), which is either empty ($\emptyset$), a usual sequential program ($\langle prog \rangle$), an *execution statement* followed by a sequential program ($\langle exec\text{-}stat \rangle$; $\langle prog \rangle$), or a *locked statement*, written $\odot\langle ident\text{-}set \rangle[\langle exec\text{-}stat \rangle]$ and carrying additional information about the locks owned by this particular thread. The grammar of *execution statements* is given in Figure 4.3 and is based on the grammar of sequential programs (Figure 4.1). A *locked statement* ($\odot\bar{x}[P]$)

$$\begin{array}{rcl}\langle exec\text{-}stat \rangle & ::= & \emptyset \\ & | & \langle prog \rangle \\ & | & \langle exec\text{-}stat \rangle \ ; \ \langle prog \rangle \\ & | & \odot\langle ident\text{-}set \rangle[\langle exec\text{-}stat \rangle]\end{array}$$

Figure 4.3: Grammar of sequential programs under execution

is obtained after the execution of a synchronization command (**with** $\bar{x}$ **when** e **do** P **done**). When a thread $\Theta(i)$ executes a synchronization command, it acquires the locks of the variables given in parameter ($\bar{x}$). This information is registered in the program state to forbid other threads from acquiring those locks as long as the thread $\Theta(i)$ has not released them. However, the thread $\Theta(i)$ is still allowed to execute synchronization commands on those locks — i.e. acquire again the same locks. The information needed to allow a thread to acquire again the same lock is not registered in the program state. The locks a given thread is allowed to acquire again are registered in its sequential program itself by the substitution of the synchronization command just evaluated by a locked statement. This locked statement contains the variables whose locks have just been acquired and the program to be executed while holding those locks.

A statement under execution (an *execution statement*) is a set — potentially empty — of nested *locked statements* followed by a usual sequential program. For this reason, we see that the language generated by the BNF rules of Figure 4.3 for execution statements ($\langle exec\text{-}stat \rangle$) is, with $n \geq 0$:

$$\Big(\ \odot\langle ident\text{-}set \rangle[\ \Big)^n \ \underbrace{\Big(\langle prog \rangle \ + \ \emptyset\Big)}_{\text{NESP}} \ \Big(\ \Big(; \langle prog \rangle\Big)^? \ ] \ \Big)^n \ \Big(; \langle prog \rangle\Big)^?$$

NESP is the position in the execution statement where the next step of execution of the thread will take place.

Concurrent program semantics. A concurrent program under execution is a pool of execution statements. One step of execution of a concurrent program consists of one execution step for one of the execution statements of the thread pool. There is no constraint on which thread has to be evaluated. The scheduler used for the thread interleaving is a nondeterministic one. After each step of execution, the next thread to execute is nondeterministically selected among those which can take an execution step.

Let ς and ς' denote execution states, and S'_ι denote an execution statement. The semantics of unmonitored executions of concurrent programs (pool of threads) is given by the following rule:

$$\frac{\iota \in \mathrm{dom}(\Theta) \qquad \langle\!| \varsigma \vdash \Theta(\iota) |\!\rangle \xrightarrow{o}_{O} \langle\!| \varsigma' \vdash S'_\iota |\!\rangle}{\langle\!| \varsigma \vdash \Theta |\!\rangle \xrightarrow{o}_{O} \langle\!| \varsigma' \vdash \Theta[\iota \mapsto S'_\iota] |\!\rangle} \qquad (\text{E}_O\text{-CONCUR})$$

As explained above, to take one execution step of a thread pool Θ in state ς, the scheduler selects nondeterministically a thread ι whose execution statement ($\Theta(\iota)$) can take one execution step. One execution step of this execution statement, in state ς, yields execution statement S'_ι, execution state ς' and output sequence o. Therefore, one execution step of the thread pool Θ can, if ι is selected by the scheduler, yield the output sequence o, execution state ς' and the thread pool in which the ι^{th} thread's statement is S'_ι and the other execution statements are the same as in Θ.

Execution statement semantics. The semantics of concurrent programs is based on the semantics of *execution statements* which is described in Figure 4.4. This latter semantics is based on rules written in the format:

$$\langle\!| \varsigma \vdash S |\!\rangle \xrightarrow{o}_{O} \langle\!| \varsigma' \vdash S' |\!\rangle$$

The rules mean that, in the execution state ς, statement S evaluates to statement S' yielding state ς' and output sequence o. Let $\mathbb{X}$ be the domain of variable identifiers and $\mathbb{D}$ be the semantic domain of values. An execution state ς is a pair (σ, λ) composed of a value store σ ($\mathbb{X} \to \mathbb{D}$) and a lock set λ ($2^{\mathbb{X}}$). σ maps variable identifiers to their respective current value. The definition of value stores is extended to expressions, so that $\sigma(e)$ is the value of the expression e in a program state whose value store is σ. λ is a set of variable identifiers. It corresponds to the set of variables whose lock is currently owned by a thread. An output sequence is a word in $\mathbb{D}^\star$. It is either an empty sequence (written ϵ), a single value (for example, $\sigma(e)$) or the concatenation of two other sequences (written $o_1\ o_2$).

As shown by the rule (E$_O$-WITH), if the prerequisites allow it, the execution of a synchronization command yields an *execution statement*, $\odot\bar{x}[\mathtt{P}]$, where $\bar{x}$ is a set of variable identifiers. This rule states that if the conditions hold, a synchronization command is replaced by the program P it includes. The current thread will now own the locks of the variables belonging to $\bar{x}$.

Example of thread pool execution. Let Θ be the thread pool described in Table 4.1 on page 54. With any initial execution state (σ, λ), where λ is empty, any of Θ's threads can take one execution step.

$$\langle\!|\, (\sigma,\lambda) \vdash x := e \,|\!\rangle \xrightarrow{\epsilon}_O \langle\!|\, (\sigma[x \mapsto \sigma(e)],\lambda) \vdash \emptyset \,|\!\rangle \qquad (\mathrm{E}_O\text{-ASSIGN})$$

$$\langle\!|\, (\sigma,\lambda) \vdash \textbf{output } e \,|\!\rangle \xrightarrow{\sigma(e)}_O \langle\!|\, (\sigma,\lambda) \vdash \emptyset \,|\!\rangle \qquad (\mathrm{E}_O\text{-OUTPUT})$$

$$\langle\!|\, \varsigma \vdash \textbf{skip} \,|\!\rangle \xrightarrow{\epsilon}_O \langle\!|\, \varsigma \vdash \emptyset \,|\!\rangle \qquad (\mathrm{E}_O\text{-SKIP})$$

$$\frac{\sigma(e) = v}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{if } e \textbf{ then } S^{\texttt{true}} \textbf{ else } S^{\texttt{false}} \textbf{ end} \,|\!\rangle \xrightarrow{\epsilon}_O \langle\!|\, (\sigma,\lambda) \vdash S^{v} \,|\!\rangle} \qquad (\mathrm{E}_O\text{-IF})$$

$$\frac{\sigma(e) = \texttt{true}}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{while } e \textbf{ do } S^{\mathrm{l}} \textbf{ done} \,|\!\rangle \xrightarrow{\epsilon}_O \langle\!|\, (\sigma,\lambda) \vdash S^{\mathrm{l}} \,;\, \textbf{while } e \textbf{ do } S^{\mathrm{l}} \textbf{ done} \,|\!\rangle} \qquad (\mathrm{E}_O\text{-WHILE}_{\texttt{true}})$$

$$\frac{\sigma(e) = \texttt{false}}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{while } e \textbf{ do } S^{\mathrm{l}} \textbf{ done} \,|\!\rangle \xrightarrow{\epsilon}_O \langle\!|\, (\sigma,\lambda) \vdash \emptyset \,|\!\rangle} \qquad (\mathrm{E}_O\text{-WHILE}_{\texttt{false}})$$

$$\frac{\bar{x} \cap \lambda = \emptyset \qquad \sigma(e) = \texttt{true}}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{with } \bar{x} \textbf{ when } e \textbf{ do } S^{\mathrm{s}} \textbf{ done} \,|\!\rangle \xrightarrow{\epsilon}_O \langle\!|\, (\sigma,\lambda \cup \bar{x}) \vdash \odot\bar{x}[S^{\mathrm{s}}] \,|\!\rangle} \qquad (\mathrm{E}_O\text{-WITH})$$

$$\frac{\langle\!|\, \varsigma \vdash S^{\mathrm{h}} \,|\!\rangle \xrightarrow{o}_O \langle\!|\, \varsigma' \vdash \emptyset \,|\!\rangle}{\langle\!|\, \varsigma \vdash S^{\mathrm{h}} \,;\, S^{\mathrm{t}} \,|\!\rangle \xrightarrow{o}_O \langle\!|\, \varsigma' \vdash S^{\mathrm{t}} \,|\!\rangle} \qquad (\mathrm{E}_O\text{-SEQ}_{\mathrm{action}})$$

$$\frac{\langle\!|\, \varsigma \vdash S^{\mathrm{h}} \,|\!\rangle \xrightarrow{o}_O \langle\!|\, \varsigma' \vdash S^{\mathrm{h}'} \,|\!\rangle \qquad S^{\mathrm{h}'} \neq \emptyset}{\langle\!|\, \varsigma \vdash S^{\mathrm{h}} \,;\, S^{\mathrm{t}} \,|\!\rangle \xrightarrow{o}_O \langle\!|\, \varsigma' \vdash S^{\mathrm{h}'} \,;\, S^{\mathrm{t}} \,|\!\rangle} \qquad (\mathrm{E}_O\text{-SEQ}_{\mathrm{branch}})$$

$$\frac{\langle\!|\, (\sigma,\lambda - \bar{x}) \vdash S^{\mathrm{s}} \,|\!\rangle \xrightarrow{o}_O \langle\!|\, (\sigma',\lambda') \vdash \emptyset \,|\!\rangle}{\langle\!|\, (\sigma,\lambda) \vdash \odot\bar{x}[S^{\mathrm{s}}] \,|\!\rangle \xrightarrow{o}_O \langle\!|\, (\sigma',\lambda') \vdash \emptyset \,|\!\rangle} \qquad (\mathrm{E}_O\text{-LOCKED}_{\mathrm{action}})$$

$$\frac{\langle\!|\, (\sigma,\lambda - \bar{x}) \vdash S^{\mathrm{s}} \,|\!\rangle \xrightarrow{o}_O \langle\!|\, (\sigma',\lambda') \vdash S^{\mathrm{s}'} \,|\!\rangle \qquad S^{\mathrm{s}'} \neq \emptyset}{\langle\!|\, (\sigma,\lambda) \vdash \odot\bar{x}[S^{\mathrm{s}}] \,|\!\rangle \xrightarrow{o}_O \langle\!|\, (\sigma',\lambda' \cup \bar{x}) \vdash \odot\bar{x}[S^{\mathrm{s}'}] \,|\!\rangle} \qquad (\mathrm{E}_O\text{-LOCKED}_{\mathrm{seq}})$$

Figure 4.4: Output semantics of sequential programs in the concurrent setting

Thread	Execution statement
1	**with** x, y **when** $\texttt{true}$ **do** $x := x + 1$; $y := y + 1$ **done**
2	**with** x, y **when** $\texttt{true}$ **do** $x := 2 \times x$; $y := 2 \times y$ **done**
3	**if** b **then** $x := \mathbb{0}$; $y := \mathbb{0}$ **else skip end** ; **output** x ; **output** y

Table 4.1: Thread pool execution (initial thread pool)

Assuming the scheduler selects the first thread, one execution step of Θ yields the thread pool Θ_1 (described in Table 4.2). The new execution state is (σ_1, λ_1) where $\sigma = \sigma_1$ and $\lambda_1 = \{x, y\}$. As at least one of x and y belongs to λ_1, the second thread can not take an execution step. Therefore, there can be no concurrent access of the two first threads on the pair (x, y). For the next step, the scheduler can select only the first or third thread.

Thread	Execution statement
1	$\odot x, y[x := x + 1 \;;\; y := y + 1]$
2	**with** x, y **when** $\texttt{true}$ **do** $x := 2 \times x$; $y := 2 \times y$ **done**
3	**if** b **then** $x := \mathbb{0}$; $y := \mathbb{0}$ **else skip end** ; **output** x ; **output** y

Table 4.2: Thread pool execution (after the execution sequence [1])

Assuming the execution take one step in thread 3 (with $\sigma(b) = \texttt{true}$) and then one other step in thread 1, the new thread pool is Θ_3 (described in Table 4.3). The new execution state is (σ_3, λ_3) where $\sigma_3 = \sigma_1[x \mapsto \sigma_1(x) + 1]$ and $\lambda_3 = \lambda_1$. As at least one of x and y belongs to λ_1, the second thread can not take an execution step. However, the third thread can take an execution step, and then modify the pair (x, y) concurrently with the first thread.

Thread	Execution statement
1	$\odot x, y[y := y + 1]$
2	**with** x, y **when** $\texttt{true}$ **do** $x := 2 \times x$; $y := 2 \times y$ **done**
3	$x := \mathbb{0}$; $y := \mathbb{0}$; **output** x ; **output** y

Table 4.3: Thread pool execution (after the execution sequence [1, 3, 1])

4.1.3 Monitoring Principles

The mechanism developed in this chapter is a monitor aiming at enforcing the confidentiality of private data manipulated by any concurrent program. In order to achieve this goal, the monitoring mechanism is designed to enforce a stronger property based on the notion of noninterference. The informal definition of this notion (Goguen and Meseguer, 1982) states that a program is "safe" if its private inputs have no influence on the observable behavior of the program. This definition is stated at the level of the whole program (or at the level of all its executions as a whole). On the other hand, a monitor acts at the level of one execution and not of all executions of a program. Consequently, it is necessary to refine the notion of noninterference in order for it to be applicable to a single execution.

The remainder of the current section starts by giving some definitions which are then used to define formally the notion of noninterference for executions. Subsequently, it provides a short description of the way the monitor works. Finally, by commenting upon two examples of concurrent programs, it illustrates some of the difficulties of monitoring noninterference and the solutions adopted.

An *initialized program* is a program whose inputs are set. More formally, as described by Definition 4.1.1, an initialized program is a pair composed of a program, P, and a *program state*, X. The set of program states includes the set of *execution states* defined on page 52 and the set of *monitored execution states* defined on page 67. By extension, an *initialization of a program* P is an initialized program whose first element is P. The execution of an initialized program (P, X) is the execution of P with the initial state X. With a deterministic semantics, the output sequence generated by the execution of an initialized program is unique. Generally, in a concurrent setting, this is not the case: the output sequence generated during the execution of an initialized program depends on the nondeterministic thread interleaving. In this chapter, the observable behavior of an execution is the output sequence generated. By extension, the observable behavior of an initialized program in a deterministic setting is the output sequence generated by its execution. However, in a nondeterministic concurrent setting, the output sequence generated by the execution of an initialized program depends on the thread interleaving which occurred during the execution. The definition of the observable behavior of an initialized program in a nondeterministic setting must then take into account all those possible output sequences. Additionally, as it is also desired to take into account non-terminating executions, our definition of observable behavior must also take into consideration the observable behavior of an initialized program at any step of its execution, not only the final one. Definition 4.1.2 states that the observable behavior of an initialized concurrent program (Θ, X) is the set of all prefixes of any output sequence which can be obtained by an execution of Θ in the initial state X.

Definition 4.1.1 (Initialized program).
An initialized program *is a pair (P, X) composed of a program,* P*, and a program state, X.*

Definition 4.1.2 (Observable behavior: $\mathcal{O}[\![\Theta]\!]_s X$).
Let $\rightarrow_s$ denote a small-step semantics. For all concurrent programs Θ and program states X, the observable behavior with the semantics $\rightarrow_s$ of the initialized program (Θ, X), written $\mathcal{O}[\![\Theta]\!]_s X$, is the set:

$$\{\, o \mid \exists\, X', \Theta' \;:\; \langle\!|\, X \vdash \Theta \,|\!\rangle \xrightarrow{o}{}_s^{\star} \langle\!|\, X' \vdash \Theta' \,|\!\rangle \,\}$$

What is a noninterfering concurrent execution? In order to prove the monitor's soundness with regard to noninterference in section 4.5.1, it is required first to define precisely "noninterference" in our concurrent setting. A program P is said to be noninterfering if and only if its private inputs have no influence on the observable behavior of the program. Noninterference for a program P is usually characterized as follows:

the observable behaviors of any two initializations of P, which differ only in the values of private inputs, are the same.

The monitoring mechanism does not determine if a program is or is not noninterfering, but rather if a precise execution is or is not noninterfering. It then has to determine if, by looking at the current output sequence, it is possible to differentiate the initialized program executed from initializations of the same program with the same public inputs but potentially different private inputs. This is possible only if the output sequence, the observable behavior, of the current execution — which follows a precise thread interleaving — of the program P can not be produced, for any thread interleaving, from another initial state X_2 having the same public inputs; in other words, only if there exists an initial state X_2, having the same public inputs as the initial state of the current execution, such that the output sequence of the current execution does not belong to the observable behavior of the initialization with X_2 of the same program ($\mathcal{O}[\![\Theta]\!]_s X_2$). Definition 4.1.4 characterizes a noninterfering execution as an execution whose output sequence belongs to the observable behavior of any initialization of the same program with an initial state low equivalent (Definition 4.1.3) to the initial state of the current execution.

Definition 4.1.3 (Low Equivalent States).
Two states X_1, respectively X_2, containing the value stores σ_1, respectively σ_2, are low equivalent *with regards to a set of variables V, written $X_1 \overset{V}{=} X_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2:*

$$X_1 \overset{V}{=} X_2 \quad \Longleftrightarrow \quad \forall x \in V : \sigma_1(x) = \sigma_2(x)$$

Definition 4.1.4 (Noninterfering Execution).
Let V^c be the complement of V in the set $\mathbb{X}$ and $\rightarrow_s$ denote a small-step semantics. For all concurrent programs Θ, program states X_1 and output sequences o, an execution with the semantics $\rightarrow_s$ of the initialized program (Θ, X_1) generating the output sequence o is noninterfering, written $ni(\Theta, s, X_1, o)$, if and only if:

$$\forall X_2 : \; X_1 \overset{\mathcal{S}(\Theta)^c}{=} X_2 \;\Rightarrow\; o \in \mathcal{O}[\![\Theta]\!]_s X_2$$

Notice that the scheduler used for the thread interleaving is nondeterministic. The attacker cannot assume any knowledge prior to the execution about the precise thread interleaving; as it could do with a Round-Robin scheduler. After each step of execution, the next thread to execute is nondeterministically selected among those which can take an execution step. Such a scheduler avoids timing covert channels which are based on the hypothesis that the attacker is able to guess accurately in which order the scheduler will execute the different threads. When dealing with schedulers which are deterministic, the solutions proposed in (Russo et al., 2006) are applicable.

The notion of noninterference used in this chapter relates to the notion of possibilistic noninterference (Sutherland, 1986) and not the notion of probabilistic noninterference (Volpano and Smith, 1999). Relating to static analyses, a program is noninterfering if and only if *all its executions* are noninterfering. Compared to Chapter 3, the definition of noninterference used in this chapter takes into consideration nontermination, which is not the case of the work presented in the preceding chapter. Let P be the following program where h is a secret input: **while** h **do skip done** ; **output** $\mathbb{0}$. From the point of view of Chapter 3, every terminating execution of `P` is safe as any such execution outputs the same sequence. However, using Definition 4.1.4, any terminating execution of `P` is considered interfering as any such executions outputs `0` and only executions for which h is `false` do the same.

How does the monitor enforce noninterference? As in Chapter 3, the monitoring mechanism is separated into two parts. The first element, described in Section 4.3, is a special semantics which delegates the main job to a security automaton described in Section 4.2. The purpose of the special semantics is to select and abstract the important events, with regard to noninterference, which occur during the execution. The abstractions of those events are then sent to the security automaton. For each input received, the automaton sends back to the semantics an output. Depending on those outputs received, the special semantics modifies the normal execution of the program. The security automaton is in charge of keeping track of the variables whose value carries *variety* — i.e. their value is influenced by the values of the private inputs — and keeping track of *variety* in the context of execution — i.e. the program counter — of each thread independently.

With regard to synchronization commands, the solution adopted follows the standard solution found in the literature (Sabelfeld, 2001). Synchronizations inside the branch of a conditional whose branching condition is influenced by the private inputs are forbidden. However, it is impossible to stop the execution when encountering a synchronization command inside a conditional whose branching condition carries variety. Doing so would create a new covert channel leaking information to low-level users if a public output was supposed to occur later in the execution. One solution would consist in ignoring synchronization commands in a context carrying variety. The drawback of this solution is to suppress synchronizations that the programmer deemed it wise to include. The solution adopted in the monitor presented in this chapter is to force any synchronization to occur outside any conditional whose branching expression carries variety. Whenever the monitoring mechanism has to evaluate a conditional whose test is influenced by the secret inputs, it executes the locking part of any synchronization command appearing in any branch of the conditional.

The monitoring mechanism by the example. As with the monitor described in Chapter 3, the security automaton is unaware of the precise values of variables. Consequently, it sometimes concludes that variety is carried to a variable when it is not the case. A typical example is when a variable is assigned the same value in all branches of a conditional whose condition carries variety. In such a case, the value of the assigned variable is always the same at the end of the conditional. However, capturing this particular situation is outside the scope of the proposed monitor and of the majority of works on noninterference.

Synchronization commands are interference prone. Figure 4.5 contains the code of a concurrent program. It is composed of two threads, has a single private input (h) and uses an internal variable v which is never output. The first thread (Figure 4.5(a)) takes the lock of the variable v and then outputs "a" followed by "b" before releasing the lock. The second thread (Figure 4.5(b)) outputs "c"; then, if the private input h is `true`, it tries to acquire the lock of v and then releases it; finally, it outputs "d". What can be deduced by a low-level user if it sees the output sequence "a c d b"? From the code of thread $\Theta(1)$, it is possible to deduce that the first thread owns the lock of v at least from the time the program outputs "a" until it outputs "b".

```
with v when true do
  output "a";
  ...
  output "b"
done
```

(a) Thread Θ(1)

```
output "c";
if h
  then
    with v when true do
      ...
    done
  else skip
end;
output "d"
```

(b) Thread Θ(2)

Figure 4.5: Example of interference due to a lock

Consequently, as “c” and “d” are output between “a” and “b”, thread Θ(2) has evaluated every command between the two outputs of “c” and “d” without needing to acquire the lock of v. It is then easy, by looking at the code of Θ(2), to deduce that the private input h is `false`. The “bad” flow from h to the output sequence is due to the synchronization commands on v.

For some output sequences, as “a c d b”, it is possible to deduce that the synchronization command in Θ(2) has not been executed, which in turn allows to deduce the value of h. It is worthwhile to notice that *not evaluating* the synchronization command in Θ(2) — i.e. h is `false`— leads to a “bad” flow, but that this is not the case for *the evaluation* of the synchronization command in Θ(2) — i.e. h is `true`. If h is `true`, it is impossible to deduce it from the output sequence. Indeed, whatever output sequence is generated when h is `true`, there exists a thread interleaving when h is `false` generating the same output sequence.

To prevent the “bad” flow presented above, it is not a good idea to simply stop the execution whenever a synchronization command has to be executed inside a conditional whose condition carries variety. In the example presented, the synchronization command in Θ(2) is *not* executed. Stopping the program each time the program has to evaluate a conditional whose condition carries variety and which contains a synchronization command is not appealing either. And, it does not seem wise to ignore any synchronization command inside a conditional whose condition carries variety. There may be a good reason for having one synchronization command in the previous example, avoiding a concurrent write access to the variable v for example. The solution adopted by the monitoring mechanism presented in this chapter consists in acquiring all the locks of any synchronization command appearing in any branch of a conditional whose condition carries variety before evaluating such a conditional. This mechanism prevents the program from outputting “a c d b” even if h is `false`.

Be cagey with newsmongers. Figure 4.6(a) contains the code of a Very Important Program (VIP) which has a single private input (h) and is run concurrently with the program in Figure 4.6(b). This latter program is a newsmonger which outputs indefinitely and as fast as it can some of the variables manipulated by the VIP (x and y). By itself, the VIP is not leaking any secret information, the main reason being that it does not output anything. This program only sets x to 0, then y to 0 and finally resets both to 1.

However, depending on the value of its private input, it resets first x or y to 1. The interference comes

```
x := 0; y := 0;
if h then
  x := 1; y := 1
else
  y := 1; x := 1
end;
```

(a) VIP

```
while true do
  output x;
  output y;
done
```

(b) Newsmonger

Figure 4.6: Example of interference due to concurrent access

from the newsmonger. If the newsmonger is lucky enough, and the VIP unlucky enough, the scheduler will let the newsmonger take at least two steps, so outputting x and y, while the VIP is in the middle of resetting those variables to 1. It is then easy, depending on which one of x and y has been reset to 1 first, to deduce the value of the private input h. The newsmonger is able to steal the value of the private input of an unmonitored execution of the VIP. Less intuitively, it is also possible, if the scheduler gives a high priority to the newsmonger, to deduce the value of h if the execution is supervised by a monitor which takes into account indirect flows created by an assignment only when this assignment is evaluated. The reason is that, in between the two assignments resetting x and y to 1, those two variables do not have the same security level. Consequently, such a "bad" monitor does not act the same way for the commands "**output** *x*" and "**output** *y*" if only one of x and y has been reset to 1.

The monitoring mechanism proposed in this chapter takes into account an indirect flow as soon as the conditional which is the source of this flow is executed. For any execution of the VIP, this means that x and y get a high security level at the same time — when the conditional branching on h is first evaluated. Therefore, the monitor has the same behavior for the two commands "**output** *x*" and "**output** *y*" — output a default value instead of the value of x or y. The newsmonger is then unable to steal the VIP's secret.

The remainder of the chapter is organized as follows. Section 4.4 describes precisely the behavior of the monitoring mechanism on an example. Before that, the next two sections give a formal definition of the monitoring mechanism: first of the security automaton used, and then of the special semantics communicating with this automaton.

4.2 The Monitoring Automaton

This section describes the security automaton used in the monitoring mechanism. It is in charge of tracking the information flows and controlling the execution in order to enforce noninterference. The semantics described in section 4.3 sends abstractions of the execution events to this automaton. In turn, the monitoring automaton sends back directions to the semantics to control the execution, thus *enforcing* noninterference.

Notations. For any set $\mathbb{S}$, let $2^{\mathbb{S}}$ be the power set of $\mathbb{S}$. For any set $\mathbb{A}$, let $\mathbb{A}^{\star}$ be the set of all strings over the alphabet $\mathbb{A}$. For any sequential program $\mathtt{P}$ whose variables belongs to $\mathbb{X}$, let the set of private input variables be $\mathcal{S}(\mathtt{P})$ ($\mathcal{S}(\mathtt{P}) \subseteq \mathbb{X}$).

The monitoring automaton is formally defined using multisets (Blizard, 1988; Syropoulos, 2001). A multiset, sometimes called a bag, is a pair (A, m) where A, called *underlying set of elements*, is a set and m $(A \rightarrow \mathbb{N})$ maps elements from A to their number of occurrences, called *multiplicity*. For example, if $A = \{a, b, c\}$, $m(a) = 2$, $m(b) = 0$ and $m(c) = 1$ then (A, m) defines the multiset $\{a, a, c\}$. In the following, if m $(\mathbb{M} \rightarrow \mathbb{N})$ denotes a multiset — which is known from the context — then it is the multiset $(\mathbb{M}, m)$. Consequently, m must be total over the *underlying set of elements* of the multiset. However, for simplicity, even if x does not belongs to the domain of the underlying function m, written $\mathrm{dom}(m)$, $m(x)$ is considered to be defined and equal to 0. The empty multiset, written $\emptyset$, is the multiset whose underlying function has an empty domain.

Operations on multisets are defined as follows. For any multisets m, n and f:

$$\begin{array}{rll}
m \subseteq n & \text{iff} & \forall x : m(x) \leq n(x) \\
m = n & \text{iff} & \forall x : m(x) = n(x) \\
(m \uplus n) = f & \text{iff} & \forall x : f(x) = m(x) + n(x) \\
(m \cup n) = f & \text{iff} & \forall x : f(x) = \max(m(x), n(x)) \\
(m \cap n) = f & \text{iff} & \forall x : f(x) = \min(m(x), n(x)) \\
(m \setminus n) = f & \text{iff} & \forall x : f(x) = \max(0, m(x) - n(x))
\end{array}$$

If one of those operations involve a multiset and a set $\mathbb{S}$, this set is considered as the multiset whose *underlying set of elements* is $\mathbb{S}$ and whose elements have a multiplicity of 1.

Formal definition of the monitoring automaton for sequential programs. The preceding sections presented the monitoring mechanism as if it is using a single monitoring automaton. However, the monitoring automaton for a concurrent program is defined by first defining monitoring automata for sequential programs. A concurrent program is a pool of threads, each of which contains a single sequential program. Each time the "automaton" of the concurrent program Θ receives an input ϕ, generated by an evaluation step in a sequential program $\Theta(i)$, a state for the monitoring automaton of $\Theta(i)$ is extracted from the state of the monitoring automaton of Θ. Then, the monitoring automaton for sequential programs takes a transition on input ϕ generating output ψ. Finally, the state of the monitoring automaton of Θ is updated using the new state returned by the transition of the monitoring automaton for sequential programs and the output ψ is sent back to the monitoring semantics.

For any program $\mathtt{P}$, let $\mathbb{X}$ be the set of variables of $\mathtt{P}$. The automaton enforcing noninterference for the sequential program $\mathtt{P}$ is the tuple $\mathcal{A}(\mathtt{P}) = (Q, \Phi, \Psi, \delta, q_0)$ where:

- Q is a set of states ($Q = 2^{\mathbb{X}} \times (\mathbb{X} \rightarrow \mathbb{N}) \times 2^{\mathbb{X}} \times \{\top, \bot\}^{\star}$);
- Φ is the input alphabet, constituted of abstractions of a subset of program events, specified below;

- Ψ is the output alphabet, constituted of execution controlling commands, specified below;
- δ is a transition function $(Q \times \Phi) \longrightarrow (\Psi \times Q)$;
- q_0, an element of Q, is the start state ($q_0 = (\mathcal{S}(\mathtt{P}), \emptyset, \emptyset, \epsilon)$).

The automaton states. An automaton state is a quadruple (V, W, L, w) composed of two sets (V and L) of variables belonging to $\mathbb{X}$, a multiset (W) of variables belonging to $\mathbb{X}$ and a word (w) belonging to a language whose alphabet is $\{\top, \bot\}$. At any step of the execution, V contains all the variables which *may* carry variety — i.e. whose values *may* have been influenced by the initial values of the secret input variables ($\mathcal{S}(\mathtt{P})$). L contains the variables whose lock *may* be owned by a thread at an "equivalent" step of an execution of the same program started with the same public inputs but potentially different private inputs. L is the part of the automaton state which protects secret data against attacks similar to the one described in the example about synchronization commands on page 57. W contains at least once each variable which is assigned in a branch of a still active conditional whose condition carries variety. For example, at any step of the execution of line 3 or 5 of the program in Figure 4.6(a), W contains $\mathtt{x}$ and $\mathtt{y}$. W is the part of the automaton state which protects secret data against attacks similar to the one described in the newsmonger's example on page 58. w, called a *branching context*, tracks variety in the context of the execution with regard to previous branching commands. The context consists in the level of *variety* the conditions of the previous, but still active, branching commands. If w contains $\top$, then the statement executed was initially a branch of a conditional whose test may have been influenced by the initial values of $\mathcal{S}(\mathtt{P})$. Hence this statement may not be executed with a different choice of initial values for the private input variables ($\mathcal{S}(\mathtt{P})$).

Automaton inputs. The input alphabet of the automaton (Φ) contains abstractions of the events, interesting for noninterference monitoring, that can occur during an execution. The alphabet Φ consists of the following:

"$\mathtt{branch}(\lambda, e, P^{\mathbf{e}}, P^{\mathbf{u}})$" is generated each time a conditional is evaluated. λ is the set of variables whose lock is currently owned by some thread, e is the expression whose value determines the branch which is executed, P^{e} is the branch which will be executed, and P^{u} is the branch which will not be executed. For example, with $((\sigma, \lambda), q)$ being the monitored execution state, before evaluation of "**if** $\mathtt{x} > \mathtt{10}$ **then** S_1 **else** S_2 **end**", the input "$\mathtt{branch}(\lambda, \mathtt{x} > \mathtt{10}, S_1, S_2)$" is sent to the monitoring automaton if x is greater than 10.

"$\mathtt{merge}(P^{\mathbf{e}}, P^{\mathbf{u}})$" is generated each time a conditional has been fully evaluated. P^{e} is the branch which has been executed, and P^{u} is the branch which has not been executed. After evaluation of "**if** $\mathtt{x} >$ 10 **then** S_1 **else** S_2 **end**", for example, the input "$\mathtt{merge}(S_1, S_2)$" is sent to the monitoring automaton.

"$\mathtt{sync}(\bar{x}, e)$" is generated before any synchronization command. For example, before evaluation of "**with** x, b **when** b **do** S^{s} **d** the input "$\mathtt{sync}(\{x, b\}, b)$" is generated.

any atomic action of the language (assignment, skip or output statement). Any action which has to be evaluated is first sent to the automaton for validation before its execution.

Automaton outputs. The automaton outputs control the behavior of the monitoring semantics. They are sent back from the automaton to the monitoring semantics in order to control the execution. The output alphabet (Ψ) is composed of the following:

"OK" is used whenever the monitoring automaton allows an execution step that it could have altered or denied.

"NO" is used whenever the monitoring automaton forbids the execution of an action. This action is simply skipped by the monitored execution.

any atomic action of the language. This is the answer of the monitoring automaton whenever another action than the current one has to be executed.

Compared to Chapter 3, the output "ACK" has disappeared. The reason is that the automaton can block the execution on any non-action commands. In Chapter 3, some inputs do not require intervention of the automaton on the execution. Their only goal is to help the automaton track useful information. There is not anymore such automaton inputs for this chapter.

Automaton transitions. Figure 4.7 specifies the transition function of the automaton. A transition rule is written:

$$(q, \phi) \xrightarrow{\psi} q'$$

It reads as follows: in the state q, on reception of the input ϕ, the automaton moves to state q' and outputs ψ. Let $FV(e)$ be the set of variables occurring in e. For example, $FV(x + y)$ returns the set $\{x, y\}$.

Let $modified(S)$ be the set of all variables whose value may be modified by an execution of S. This function is used in order to take into account the implicit indirect flows created when a conditional whose condition carries variety is evaluated. The formal definition of this function follows:

$$\begin{array}{lcl}
modified(x := e) & = & \{x\} \\
modified(\mathbf{output}\ e) & = & \emptyset \\
modified(\mathbf{skip}) & = & \emptyset \\
modified(S_1\ ;\ S_2) & = & modified(S_1) \cup modified(S_2) \\
modified(\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}) & = & modified(S_1) \cup modified(S_2) \\
modified(\mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}) & = & modified(S) \\
modified(\mathbf{with}\ \bar{x}\ \mathbf{when}\ e\ \mathbf{do}\ S\ \mathbf{done}) & = & modified(S)
\end{array}$$

Let $NLocks(S)$ be the set of all variables whose lock may required in order to execute S. This function is used by the monitoring automaton in order to "virtually" acquire all the locks which may be needed for the complete evaluation of a conditional whose condition carries variety. Those locks are not acquired by the current thread. They will be only when their corresponding synchronization command will be executed. However, the monitoring automaton registers them in its state. Before allowing the evaluation of a conditional whose condition carries variety, the automaton checks that there is no needed lock which is already registered in the automaton state for another thread. This is done in order to make sure that the evaluation of the branch designated by the condition — which carries variety — can not be stopped because of a needed lock which would be owned by another thread. The formal definition of $NLocks(S)$ follows:

$$\begin{array}{lcl}
NLocks(x := e) & = & \emptyset \\
NLocks(\textbf{output}\ e) & = & \emptyset \\
NLocks(\textbf{skip}) & = & \emptyset \\
NLocks(S_1\ ;\ S_2) & = & NLocks(S_1) \cup NLocks(S_2) \\
NLocks(\textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) & = & NLocks(S_1) \cup NLocks(S_2) \\
NLocks(\textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}) & = & NLocks(S) \\
NLocks(\textbf{with}\ \bar{x}\ \textbf{when}\ e\ \textbf{do}\ S\ \textbf{done}) & = & \bar{x} \cup NLocks(S)
\end{array}$$

Finally, let $stoppable(S)$ be true if the execution of S *may* be "stopped": either by looping, thus preventing the execution of the remaining of the thread, or waiting on the expression of a synchronization command to become *true*. This function is used by the automaton when finishing the complete evaluation of some conditionals whose condition carries variety. This is done in order to prevent an attacker to learn some secret information by observing from the source code that, in under some conditions, one of the branches of a conditional, whose condition carries variety, can not terminate. $stoppable(S)$ is formally defined as follows, where `true`and `false`are constants and not values:

$$\begin{array}{lcl}
stoppable(x := e) & \text{iff} & false \\
stoppable(\textbf{output}\ e) & \text{iff} & false \\
stoppable(\textbf{skip}) & \text{iff} & false \\
stoppable(S_1\ ;\ S_2) & \text{iff} & stoppable(S_1) \vee stoppable(S_2) \\
stoppable(\textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) & \text{iff} & stoppable(S_1) \vee stoppable(S_2) \\
stoppable(\textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}) & \text{iff} & e \neq \texttt{false} \\
stoppable(\textbf{with}\ \bar{x}\ \textbf{when}\ e\ \textbf{do}\ S\ \textbf{done}) & \text{iff} & e \neq \texttt{true} \vee stoppable(S)
\end{array}$$

It would be possible to increase the precision of this analysis for loop and synchronization statements by trying to evaluate their expressions rather than simply compare them with the `true`and `false`constants. This could be done using techniques similar to those presented in Chapter 5. However, this would increase the cost and complexity of the overall dynamic analysis. The remainder of this chapter will proceed with this purely syntactic analysis.

Figure 4.7 shows that the automaton forbids (NO) or edits (**output** θ) only executions of output statements. For other inputs, it either allows the evaluation of the statement — a transition exists for the current input in the current state — or forces the monitoring semantics to take an execution step for another thread — there is no transition for the current input in the current state. Whenever a transition occurs, the automaton tracks, in the set V, the variables that may contain secret information — have *variety*. Additionally, it registers, in W, the variables whose variety may evolve differently due to a conditional whose condition carries variety, It also tracks, in L, the variables whose lock status may carry variety. Finally, it tracks, in w, the *variety* of the branching conditions.

$$((V, W, L, w),\ \texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V, W, L, w\bot) \quad \text{iff} \begin{cases} \quad FV(e) \cap V = \emptyset \\ \vee \quad w \notin \{\bot\}^{\star} \end{cases} \quad \text{(T-BRANCH-low)}$$

$$((V, W, L, w),\ \texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V \cup \tilde{v}, W \uplus \tilde{v}, L \cup \tilde{l}, w\top) \text{ with } \begin{cases} \tilde{v} = \mathit{modified}(P^{\mathrm{e}}) \cup \mathit{modified}(P^{\mathrm{u}}) \\ \tilde{l} = \mathit{NLocks}(P^{\mathrm{e}}) \cup \mathit{NLocks}(P^{\mathrm{u}}) \end{cases} \quad \text{iff} \begin{cases} \quad FV(e) \cap V \neq \emptyset \\ \wedge \quad w \in \{\bot\}^{\star} \\ \wedge \quad \tilde{l} \cap (\lambda \cup L) = \emptyset \end{cases} \quad \text{(T-BRANCH-high)}$$

$$((V, W, L, w\bot),\ \texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V, W, L, w) \quad \text{(T-MERGE-low)}$$

$$((V, W, L, w\top),\ \texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V, W \setminus \tilde{v}, L \setminus \tilde{l}, w) \text{ with } \begin{cases} \tilde{v} = \mathit{modified}(P^{\mathrm{e}}) \cup \mathit{modified}(P^{\mathrm{u}}) \\ \tilde{l} = \mathit{NLocks}(P^{\mathrm{e}}) \cup \mathit{NLocks}(P^{\mathrm{u}}) \end{cases} \quad \text{iff} \begin{cases} \quad \neg \mathit{stoppable}(P^{\mathrm{e}}) \\ \wedge \quad \neg \mathit{stoppable}(P^{\mathrm{u}}) \end{cases} \quad \text{(T-MERGE-high)}$$

$$((V, W, L, w),\ \texttt{sync}(\bar{x}, e)) \xrightarrow{OK} (V, W, L, w) \quad \text{iff} \begin{cases} \quad FV(e) \cap V = \emptyset \\ \wedge \quad (\quad w \notin \{\bot\}^{\star} \\ \quad\quad \vee \quad \bar{x} \cap L = \emptyset \quad) \end{cases} \quad \text{(T-SYNC)}$$

$$(q,\ \mathbf{skip}) \xrightarrow{OK} q \quad \text{(T-SKIP)}$$

$$((V, W, L, w),\ x \mathrel{\mathbf{:=}} e) \xrightarrow{OK} (V \cup \{x\}, W, L, w) \quad \text{iff} \begin{cases} \quad FV(e) \cap V \neq \emptyset \\ \vee \quad x \in W \end{cases} \quad \text{(T-ASSIGN-sec)}$$

$$((V, W, L, w),\ x \mathrel{\mathbf{:=}} e) \xrightarrow{OK} (V \setminus \{x\}, W, L, w) \quad \text{iff} \begin{cases} \quad FV(e) \cap V = \emptyset \\ \wedge \quad x \notin W \end{cases} \quad \text{(T-ASSIGN-pub)}$$

$$((V, W, L, w),\ \mathbf{output}\ e) \xrightarrow{OK} (V, W, L, w) \quad \text{iff } w \in \{\bot\}^{\star} \text{ and } FV(e) \cap V = \emptyset \quad \text{(T-OUTPUT-ok)}$$

$$((V, W, L, w),\ \mathbf{output}\ e) \xrightarrow{\mathbf{output}\ \theta} (V, W, L, w) \quad \text{iff } w \in \{\bot\}^{\star} \text{ and } FV(e) \cap V \neq \emptyset \quad \text{(T-OUTPUT-def)}$$

$$((V, W, L, w),\ \mathbf{output}\ e) \xrightarrow{NO} (V, W, L, w) \quad \text{iff } w \notin \{\bot\}^{\star} \quad \text{(T-OUTPUT-no)}$$

Figure 4.7: Transition function of the monitoring automata

Inputs "$\texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})$" are generated at entry points of conditionals. On reception of such inputs in state $((V, W, L, w))$, if the conditional to be evaluated belongs to a branch of a conditional whose condition carries variety (w does not belongs to $\{\bot\}^{\star}$), the automaton simply pushes $\bot$ to the end of w. Otherwise, the automaton checks if the branching condition (e) carries variety, i.e. if its value might be influenced by the initial values of $\mathcal{S}(\mathtt{P})$. To do so, it computes the intersection of the variables appearing in e with the set V. If the intersection is empty, then the condition of the branching statement does not carry variety. In that case, the automaton simply pushes $\bot$ at the end of w. If the intersection is not empty, then the value of *e may be* influenced by the initial values of $\mathcal{S}(\mathtt{P})$. If this is the case then the automaton checks if the execution can proceed without being stopped by a lock that it can not acquire. To do so, it verifies that there is no needed lock which is already owned by another thread or has already been booked previously for the execution of a conditional of another thread ($\tilde{l} \cap (\lambda \cup L) = \emptyset$). If all needed locks are available, the automaton pushes $\top$ to the end of w, registers the needed locks in the new automaton state and deals with indirect flows. This is done by adding all variables which may be assigned to in one of the branches of the conditional into the set of variables potentially carrying variety (V) and into the multiset of variables whose order of assignment may carry variety (W).

Whenever execution exits a branch — which is a member of a conditional c — the input "$\texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})$" is sent to the automaton. On reception of such input, the automaton checks if other private inputs may have induced the execution of the conditional to be stopped. This is the case if the condition of the conditional c carries variety — w ends with $\top$ — and the function *stoppable*() concludes that at least one of the branches of c can get stuck. If that is not the case, the automaton allows the evaluation step and the last letter of w is removed. As any such input matches a previous input "$\texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})$" — generated by the same conditional c — which adds a letter to the end of w, the effect of "$\texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})$" restores the context to its state before the conditional was processed. Additionally, if c is the first conditional whose condition carries variety in this sequential program, the variables added into the multiset W for protection while executing c are removed from the multiset.

Before executing any synchronization command, the monitoring semantics sends to the automaton an input of type "$\texttt{sync}(\bar{x}, e)$". The automaton allows the evaluation step only if the expression in the synchronization command does not carry variety and either the needed locks are not reserved ($\bar{x} \cap L = \emptyset$) or the automaton has already booked those locks for the current thread — this is the case if the synchronization command belongs to a conditional whose test carries variety.

Atomic actions (assignment, skip or output) are sent to the automaton for validation before their execution. The atomic action **skip** is considered safe because the noninterference definition considered in this work is not time sensitive. Hence the automaton always authorizes its execution by outputting OK.

When executing an assignment (x **:=** e), two types of flows are created. The first is a direct flow from e to x. The second flow is an explicit indirect flow from the context of execution to x. For example, the execution of the assignment in "**if** b **then** x **:=** y **else skip end**" creates such a flow from b to x. Both forms of flows are *always* created when an assignment is executed. Explicit indirect flows are indirect flows and so are already taken care for by the processing of inputs of type "$\texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})$". Therefore, on such input, the automaton deals only with direct flows. Hence, on input x **:=** e, the automaton checks if the value of e carries variety. This is the case only if $FV(e)$ and V are not disjoint. If the value of e is influenced by the initial values of $\mathcal{S}(\mathtt{P})$ then at least one of the variables appearing in e has been influenced by $\mathcal{S}(\mathtt{P})$. Such variables are members of V. Let (V', W', L', w') be the new automaton state after the transition. If the origin of the direct flow is influenced by the initial values of $\mathcal{S}(\mathtt{P})$, then x (the variable modified) is added to V: $V' = V \cup \{x\}$. Otherwise x receives a new value which is not influenced by $\mathcal{S}(\mathtt{P})$. In that case, V' equals $V \setminus \{x\}$. This makes the mechanism flow-sensitive. Other parts of the automaton state do not change: $W' = W$, $L' = L$ and $w' = w$. There is an exception to this rule. If there exists an indirect flow from the condition carrying variety of a still active conditional then the variable assigned to must not be removed from the set of variables carrying variety. If such an indirect flow exists then the assigned variable is still under the protection acquired when the corresponding "$\mathtt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})$" input has been processed ($x \in W$). In that case, the variable is left in the set V.

The rules for the automata input, "**output** e," prevent bad flows through two different channels. The first channel is the actual content of what is output. In a public context, ($w \in \{\bot\}^\star$), if the program tries to output a secret (i.e., the intersection of V and the variables in e is not empty), then the value of the output is replaced by a default value. This value can be a message informing the user that, for security reasons, the output has been denied. To do so, the automaton outputs a new output statement to execute in place of the current one. The second channel is the generation of an output by itself. This channel exists because, depending on the path followed, some outputs may or may not be executed. Hence, if the automaton detects that this output may not be executed with different values for $\mathcal{S}(\mathtt{P})$ (the context carries *variety*) then *any* output must be forbidden; and the automaton outputs NO.

The automaton states for concurrent programs. As explained above, the monitoring automaton for a concurrent program is based on the monitoring automata for the sequential programs within it. Monitoring automaton states for concurrent programs are similar to the automata states for sequential programs. An automaton state is a quadruple ($\mathtt{Q} = (V, W, L, \overline{w})$) which belongs to the following set:

$$2^{\mathbb{X}} \times (\mathbb{X} \to \mathbb{N}) \times 2^{\mathbb{X}} \times (\mathbb{N} \to \{\top, \bot\}^\star)$$

V, W and L are directly inherited from automata states for sequential programs. $\overline{w}$ is a function mapping thread identifiers to their respective branching context (as defined for sequential programs).

The functions used to extract, respectively update, the automaton state for a given thread from, respectively into, the automaton state of the concurrent program are defined below.

$$\text{extractStates}((V, W, L, \overline{w}), \iota) = (V, W, L, \overline{w}(\iota)) \tag{4.1}$$

$$\text{updateStates}((V, W, L, \overline{w}), \iota, (V', W', L', w')) = (V', W', L', \overline{w}[\iota \mapsto w']) \tag{4.2}$$

The current section defines formally the monitoring automata. Among other things, it defines an input alphabet and an output alphabet which serve to communicate with the monitoring semantics. This semantics, described in the next section, is in charge of the concrete evaluation of the concurrent program under the supervision of the monitoring automaton.

4.3 The Monitoring Semantics

During a monitored execution, a thread contains a *monitored statement* ($\langle$*monit-stat*$\rangle$) which is similar to the *execution statements* described in Figure 4.3. The addition of *branched statements* ($\otimes(\langle prog\rangle, \langle prog\rangle)[\langle monit\text{-}stat\rangle]$) is the only difference. The statement $\otimes(P^e, P^a)[S]$ states that statement S is a partial execution of P^e — i.e., S is the result of the application of some execution steps to P^e — and that P^e is the executed branch of a conditional whose unexecuted branch is P^a. The rule ($\mathrm{E}_{\mathcal{M}(s)}$-IF) of Figure 4.9 gives a good intuition of the meaning of this statement. The grammar of *monitored statements* is given in Figure 4.8 and is based on the grammar of sequential programs (Figure 4.1).

$$\begin{array}{rcl}
\langle monit\text{-}stat\rangle & ::= & \emptyset \\
& | & \langle prog\rangle \\
& | & \langle monit\text{-}stat\rangle \; ; \; \langle prog\rangle \\
& | & \odot\langle ident\text{-}set\rangle[\langle monit\text{-}stat\rangle] \\
& | & \otimes(\langle prog\rangle, \langle prog\rangle)[\langle monit\text{-}stat\rangle]
\end{array}$$

Figure 4.8: Grammar of sequential programs under monitored execution

A statement under execution (an *execution statement*) is a set — potentially empty — of interwoven *locked statements* followed by a usual sequential program. For this reason, we see that the language generated by the BNF rules of Figure 4.3 for execution statements ($\langle$*exec-stat*$\rangle$) is, with $n \geq 0$:

A statement under monitored execution is a set — potentially empty — of interwoven *locked statements* and *branched statements* followed by a usual sequential program. For this reason, we see that the language generated by the BNF rules of Figure 4.8 for *monitored statements* is, with $n \geq 0$:

$$\Big(\odot\langle ident\text{-}set\rangle[\quad + \quad \otimes(\langle prog\rangle, \langle prog\rangle)[\Big)^n \quad \underbrace{\Big(\langle prog\rangle \quad + \quad \emptyset\Big)}_{\text{NESP}} \quad \Big(\big(; \langle prog\rangle\big)^? \,]\Big)^n \quad \big(; \langle prog\rangle\big)^?$$

NESP is the position in the monitored statement where the next step of execution of the thread will take place.

Thread semantics. The monitoring semantics of concurrent programs is based on the semantics of *monitored statements* which is described in Figure 4.9. This semantics is based on rules written in the format,

$$\langle\!\langle \zeta \Vdash S \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(s)} \langle\!\langle \zeta' \Vdash S' \rangle\!\rangle$$

The previous rule reads as follows: in *monitored execution state* ζ, *monitored statement* S evaluates to S' yielding state ζ' and output sequence o. A monitored execution state ζ is a pair (ς, q) composed of an execution state (ς) of the standard semantics and a monitoring automaton state (q).

Executions of *monitored statements* interact with the security automaton by sending it an input and receiving back an output. For simplicity, we assume that the automaton inputs are generated dynamically by the interpreter that implements the semantics in Figure 4.9 (But a compiler can easily embed such events into a program's compiled code). In the rules of the semantics, this is done using automaton transitions written:

$$(q, \phi) \xrightarrow{\psi} q'$$

This transition states that, in state q, if the automaton receives the input ϕ then it yields output ψ and the new automaton state is q'. The semantics of *monitored statements* also uses the semantics of *execution statements* — the standard semantics — for the evaluation of actions (assignments, outputs, or skip statements). Those semantics can be distinguished by their name appearing on the bottom right of the arrow.

There are three rules for atomic actions: **skip**, x **:=** e and **output** e. There is one rule for each possible automaton answer to the action executed. Either the automaton authorizes the execution (OK), denies the execution (NO), or replaces the action by another one. The rules use the standard semantics (Figure 4.4) when an action must be executed. In the case where the execution is denied, the evaluation omits the current action (as if the action was "**skip**"). For the case where, on reception of input A, the monitoring automaton returns A', the monitoring semantics executes A' instead of A. Note from Figure 4.7, that A' can only be the action that outputs the default value (**output** θ).

For conditionals, the automaton receives first the input "$\mathtt{branch}(\lambda, e, P^e, P^u)$" where λ is the set of variables whose lock is currently owned by a thread, e is the test of the conditional, P^e is the "branch" which will be executed and P^u is the "branch" which will not be executed. After the evaluation step, the monitored execution state is updated with the automaton state returned by the monitoring automaton and the conditional evaluated is replaced by the branch to execute.

A synchronization command "**with** $\bar{x}$ **when** e **do** P^{s} **done**" can be evaluated only if the needed locks $\bar{x}$ are not currently owned by another thread, the condition e is true, and the monitoring automaton allows the evaluation on reception of the input "$\mathtt{sync}(\bar{x}, e)$". After the evaluation step, the needed locks are added to the set of locks currently owned by a thread, the automaton state is updated, and the command is replaced by a *locked statement* indicating that the locks $\bar{x}$ are owned by the current thread for the evaluation of the program P^{s}.

Evaluating a *branched statement* ($\otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[S^{\mathrm{b}}]$) is equivalent to evaluating the enclosed statement (S^{b}) as long as this statement is not completely evaluated. If the evaluation of S^{b} is completed then the input "$\mathtt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})$" is sent to the automaton and the evaluation of the thread can proceed only if the automaton allows it.

In order to take one evaluation step of a *locked statement* ($\odot\bar{x}[S^{\mathrm{s}}]$), the set of locks $\bar{x}$ is temporarily removed from the set of owned locks in the execution state. Then, one evaluation step is taken for S^{s} and the locks $\bar{x}$ are put back in the set of owned locks in the execution state if the evaluation of S^{s} is not completed.

Concurrent program semantics. As explained in Section 4.2, the automaton states for monitoring the execution of a single thread are different from the states for monitoring concurrent programs. The description of automaton states for monitoring concurrent programs is given in Section 4.2. Let $\mathtt{Q}$ stand for the

$$\frac{(q,\ A) \xrightarrow{OK} q' \qquad \langle\!| \varsigma \vdash A |\!\rangle \overset{o}{\rightarrow}_s \langle\!| \varsigma' \vdash \emptyset |\!\rangle}{\langle\!| (\varsigma, q) \Vdash A |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| (\varsigma', q') \Vdash \emptyset |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-OK})$$

$$\frac{(q,\ A) \xrightarrow{A'} q' \qquad \langle\!| \varsigma \vdash A' |\!\rangle \overset{o}{\rightarrow}_s \langle\!| \varsigma' \vdash \emptyset |\!\rangle}{\langle\!| (\varsigma, q) \Vdash A |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| (\varsigma', q') \Vdash \emptyset |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-EDIT})$$

$$\frac{(q,\ A) \xrightarrow{NO} q'}{\langle\!| (\varsigma, q) \Vdash A |\!\rangle \overset{\epsilon}{\rightarrow}_{\mathcal{M}(s)} \langle\!| (\varsigma, q') \Vdash \emptyset |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-NO})$$

$$\frac{\sigma(e) = v \qquad (q,\ \mathtt{branch}(\lambda, e, P^{v}, P^{\neg v})) \xrightarrow{OK} q'}{\langle\!| ((\sigma,\lambda), q) \Vdash \textbf{if } e \textbf{ then } P^{\mathtt{true}} \textbf{ else } P^{\mathtt{false}} \textbf{ end} |\!\rangle \overset{\epsilon}{\rightarrow}_{\mathcal{M}(s)} \langle\!| ((\sigma,\lambda), q') \Vdash \otimes(P^{v}, P^{\neg v})[P^{v}] |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-IF})$$

$$\frac{\begin{array}{c} P^{\mathtt{true}} = P^{\mathrm{l}} \ ; \ \textbf{while } e \textbf{ do } P^{\mathrm{l}} \textbf{ done} \qquad P^{\mathtt{false}} = \textbf{skip} \\ \sigma(e) = v \qquad (q,\ \mathtt{branch}(\lambda, e, P^{v}, P^{\neg v})) \xrightarrow{OK} q' \end{array}}{\langle\!| ((\sigma,\lambda), q) \Vdash \textbf{while } e \textbf{ do } P^{\mathrm{l}} \textbf{ done} |\!\rangle \overset{\epsilon}{\rightarrow}_{\mathcal{M}(s)} \langle\!| ((\sigma,\lambda), q') \Vdash \otimes(P^{v}, P^{\neg v})[P^{v}] |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-WHILE})$$

$$\frac{\bar{x} \cap \lambda = \emptyset \qquad \sigma(e) = \mathtt{true} \qquad (q,\ \mathtt{sync}(\bar{x}, e)) \xrightarrow{OK} q'}{\langle\!| ((\sigma,\lambda), q) \Vdash \textbf{with } \bar{x} \textbf{ when } e \textbf{ do } P^{\mathrm{s}} \textbf{ done} |\!\rangle \overset{\epsilon}{\rightarrow}_{\mathcal{M}(s)} \langle\!| ((\sigma,\lambda \cup \bar{x}), q') \Vdash \odot\bar{x}[P^{\mathrm{s}}] |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-WITH})$$

$$\frac{}{\langle\!| \zeta \Vdash \emptyset \ ; \ S^{\mathrm{t}} |\!\rangle \overset{\epsilon}{\rightarrow}_{\mathcal{M}(s)} \langle\!| \zeta \Vdash S^{\mathrm{t}} |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-SEQ}_{\mathrm{exit}})$$

$$\frac{\langle\!| \zeta \Vdash S^{\mathrm{h}} |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| \zeta' \Vdash S^{\mathrm{h}'} |\!\rangle}{\langle\!| \zeta \Vdash S^{\mathrm{h}} \ ; \ S^{\mathrm{t}} |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| \zeta' \Vdash S^{\mathrm{h}'} \ ; \ S^{\mathrm{t}} |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-SEQ}_{\mathrm{step}})$$

$$\frac{(q,\ \mathtt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} q'}{\langle\!| (\varsigma, q) \Vdash \otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[\emptyset] |\!\rangle \overset{\epsilon}{\rightarrow}_{\mathcal{M}(s)} \langle\!| (\varsigma, q') \Vdash \emptyset |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-BRANCH}_{\mathrm{exit}})$$

$$\frac{\langle\!| \zeta \Vdash S^{\mathrm{b}} |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| \zeta' \Vdash S^{\mathrm{b}'} |\!\rangle}{\langle\!| \zeta \Vdash \otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[S^{\mathrm{b}}] |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| \zeta' \Vdash \otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[S^{\mathrm{b}'}] |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-BRANCH}_{\mathrm{step}})$$

$$\frac{}{\langle\!| ((\sigma,\lambda), q) \Vdash \odot\bar{x}[\emptyset] |\!\rangle \overset{\epsilon}{\rightarrow}_{\mathcal{M}(s)} \langle\!| ((\sigma,\lambda \setminus \bar{x}), q) \Vdash \emptyset |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-LOCKED}_{\mathrm{exit}})$$

$$\frac{\langle\!| ((\sigma,\lambda \setminus \bar{x}), q) \Vdash S^{\mathrm{s}} |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| ((\sigma',\lambda'), q') \Vdash S^{\mathrm{s}'} |\!\rangle}{\langle\!| ((\sigma,\lambda), q) \Vdash \odot\bar{x}[S^{\mathrm{s}}] |\!\rangle \overset{o}{\rightarrow}_{\mathcal{M}(s)} \langle\!| ((\sigma',\lambda' \cup \bar{x}), q') \Vdash \odot\bar{x}[S^{\mathrm{s}'}] |\!\rangle} \quad (\mathrm{E}_{\mathcal{M}(s)}\text{-LOCKED}_{\mathrm{step}})$$

Figure 4.9: Semantics combining the standard semantics and the monitoring automaton

state of the security automaton for concurrent programs. The monitoring semantics uses two functions for converting between automaton states for threads and automaton states for concurrent programs. Those functions are described in Section 4.2. Succinctly, extractState($\mathtt{Q}, \iota$) is the automaton state for monitoring the ι^{th} thread of the concurrent program whose automaton state is $\mathtt{Q}$. updateStates($\mathtt{Q}, \iota, q$) is an automaton state for concurrent program equivalent to $\mathtt{Q}$ except for the ι^{th} thread. Information concerning this thread has been updated using the information contained in the automaton state for thread monitoring q. The semantics of monitored executions of concurrent programs is given by the following rule:

$$\frac{\iota \in \text{dom}(\Theta) \qquad \langle\!| (\varsigma, \text{extractState}(\mathtt{Q}, \iota)) \Vdash \Theta(\iota) |\!\rangle \xrightarrow{o}_{\mathcal{M}(s)} \langle\!| (\varsigma', q) \Vdash S'_\iota |\!\rangle}{\langle\!| (\varsigma, \mathtt{Q}) \Vdash \Theta |\!\rangle \xrightarrow{o}_{\mathcal{M}(s)} \langle\!| (\varsigma', \text{updateStates}(\mathtt{Q}, \iota, q)) \Vdash \Theta[\iota \mapsto S'_\iota] |\!\rangle} \quad (\text{E}_{\mathcal{M}(s)}\text{-CONCUR})$$

To take one execution step of a thread pool Θ in monitored execution state (ς , $\mathtt{Q}$), the scheduler selects nondeterministically a thread ι whose monitored statement, $\Theta(\iota)$), can take one execution step. The automaton state for the execution of $\Theta(\iota)$, extractState($\mathtt{Q}, \iota$), is extracted from the automaton state of Θ, $\mathtt{Q}$. One monitored execution step of this monitored statement, in execution state ς, yields monitored statement S'_ι, execution state ς', automaton state q, and output sequence o. The new automaton state of Θ, updateStates($\mathtt{Q}, \iota, q$), is constructed from the previous automaton state of Θ, $\mathtt{Q}$, and the new automaton state of $\Theta(\iota)$, q. Consequently, one execution step of the thread pool Θ can, if ι is selected by the scheduler, yield the output sequence o, monitored execution state (ς', updateStates($\mathtt{Q}, \iota, q$)), and the thread pool in which the ι^{th} thread's statement is S'_ι and the other monitored statements are the same as in Θ.

Initial state of a monitored concurrent execution. There is a unique initial state for monitoring the execution of a given concurrent program with a given initial value store. Definition 4.3.1 states that the initial state of the monitored execution of a concurrent program is the monitoring execution state whose initial value store is the given one, set of owned lock is empty and automaton state designates the value of the private inputs as the only elements carrying variety.

Definition 4.3.1 (Initial State of Monitored Executions).
For all concurrent programs Θ and value stores σ, let $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ be the initial state of the monitored execution of Θ with initial value store σ. $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ is equal to $((\sigma, \emptyset), (\mathcal{S}(\Theta), \emptyset, \emptyset, \{i \mapsto \epsilon \mid i \in dom(\Theta)\}))$.

4.4 Example of monitored execution

Figure 4.10 is an example of a concurrent program having two threads. Figure 4.10(a) contains the code of the sequential program of the first thread and Figure 4.10(b) contains the code of the second thread. This program has only one private input (h) and no public input. It uses three internal variables (v, b and x). Both threads attempt to write in the variable v. Both assignments to v are protected by a synchronization on the lock of v. Additionally, before assigning a value to v, the second thread waits for b to be `true`. After assigning a value to v, the second thread outputs twice the value of x. The first thread, depending on the value of the private input h, either assigns a value to x and outputs "a", or assigns a value to v. The last command evaluated by the first thread resets the value of x to 0.

```
1 if h then
2   x := 1;
3   output "a"
4 else
5   with v when true do
6     v := v - 1;
7   done
8 end;
9 x := 0
```

(a) Thread Θ(1)

```
1 with v when b do
2   v := v + 1;
3 done;
4 output x;
5 output x
```

(b) Thread Θ(2)

Figure 4.10: Another concurrent program

Table 4.4 shows the evolution of the monitoring automaton for an execution of the program of Figure 4.10. This execution starts in an initial state where the private input h is `true`. As the program has only one private input (h) and two threads, the initial state of the monitoring automaton is the following one:

$$(\{h\}, \emptyset, \emptyset, [1 \mapsto \epsilon, 2 \mapsto \epsilon])$$

The execution's steps are numbered on the left. The first column of the table shows the value of the program counters of the two threads. "$\imath \triangleright \jmath$" and "$\imath \blacktriangleright \jmath$" indicates that the program counter of the $\imath$th thread maps to the line $\jmath$. 0 is used for $\jmath$ when the execution of the thread is completed. $\blacktriangleright$ is used to designate the thread which will be evaluated at this execution step. The following column contains the event abstractions which are sent by the semantics to the automaton. In this column, P^t stands for the lines 2 to 3 of the first thread and P^f stands for the lines 5 to 7 of the first thread. Then comes the answer of the automaton telling the semantics what action to take. The 4th column gives the new state of the automaton following the transition triggered by the automaton input that is shown on the same line. The automaton state before the transition is the one of the preceding line. Finally, the last column lists the actions which are actually evaluated by the monitored execution of the program.

	Program counters	Automaton:			Actions executed
		input	output	new state	
1	$1 \triangleright 1 \quad 2 \blacktriangleright 1$	$\texttt{sync}(\{v\}, b)$	*OK*	$(\{h\}, \emptyset, \emptyset, [1{>}\epsilon, 2{>}\epsilon])$	
2	$1 \triangleright 1 \quad 2 \blacktriangleright 2$	$v := v + 1$	*OK*	$(\{h\}, \emptyset, \emptyset, [1{>}\epsilon, 2{>}\epsilon])$	$v := v + 1$
3	$1 \blacktriangleright 1 \quad 2 \triangleright 4$	$\texttt{branch}(\emptyset, h, P^t, P^f)$	*OK*	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	
4	$1 \triangleright 2 \quad 2 \blacktriangleright 4$	**output** x	**output** θ	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	**output** θ
5	$1 \blacktriangleright 2 \quad 2 \triangleright 5$	$x := 1$	*OK*	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	$x := 1$
6	$1 \blacktriangleright 3 \quad 2 \triangleright 5$	**output** "a"	*NO*	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	
7	$1 \blacktriangleright 8 \quad 2 \triangleright 5$	$\texttt{merge}(P^t, P^f)$	*OK*	$(\{h, x, v\}, \emptyset, \emptyset, [1{>}\epsilon, 2{>}\epsilon])$	
8	$1 \blacktriangleright 9 \quad 2 \triangleright 5$	$x := 0$	*OK*	$(\{h, v\}, \emptyset, \emptyset, [1{>}\epsilon, 2{>}\epsilon])$	$x := 0$
9	$1 \triangleright 0 \quad 2 \blacktriangleright 5$	**output** x	*OK*	$(\{h, v\}, \emptyset, \emptyset, [1{>}\epsilon, 2{>}\epsilon])$	**output** x

Table 4.4: Example of the automaton evolution during an execution.

At the execution step 2, the only thread which can be executed is the second one. The reason is that at the preceding step, the second thread took the lock of v. The first command to be executed by the first thread is a conditional containing a synchronization command on v. Therefore, to execute the first thread, the monitor requires this thread to be able to acquire the lock of v. This is impossible as this lock is owned by the second thread. Hence, even if the branch to be executed is not the one containing the synchronization command, the program can not evaluate the conditional of the first thread. An example that justifies this rule has been given on page 57.

Execution step 3 evaluates the conditional of the first thread. This conditional, whose condition carries variety, contains an assignment to x in one branch and an assignment to v in the other one. In order to prevent an attack similar to the one exposed on page 58, the monitor considers those variables (x and v) as carrying variety straight away, even if there respective assignments have not been evaluated yet. This is done by adding the variables to the first element of the new automaton state. A clever newsmonger may try to dupe the monitor into believing that those variables do not carry variety anymore by resetting their value. To prevent it, the monitor also registers that those variables carry variety because of the processing of a conditional. This is done by adding them into the multiset which is the second element of the new state. Variables x and v will appear at least once in this multiset as long as the processing of the conditional which added them into it is not over. Furthermore, as stated in the explanation of step 2, the first thread acquires the lock of v because the conditional processed at this step is conditioned by an expression carrying variety and contains a synchronization command on v in one of its branches. Finally, the monitor registers the new context of execution — reflecting the fact that the branch that will be executed depend on an expression carrying variety — by adding $\top$ to the word associated with the identifier of the first thread in the forth element of the new state.

In step 4 of the execution, the second thread attempts to output the value of x. This variable belongs to the first element of the automaton state before the transition. This means that the value of x may carry variety. Therefore, the monitor replaces the value of x by a default value. It allows the user to know that an output as been prevented for security reasons. Five steps later, at step 9, the second thread tries again to output the value of x. However, in the meanwhile, the first thread has reset the value of x to a new value. Doing so, it removed the variety in this variable. This is reflected by the fact that x does not belong anymore to the first element of the automaton state. Therefore, the monitoring automaton lets the output occur.

In step 6 of the execution, the first thread attempts to output a constant while still in one of the branch of a conditional whose condition carried variety. This is an unsafe behavior that the monitor forbids. Consequently, the execution simply skips this command.

This section shows an example execution for which the monitoring mechanism is able to ensure the confidentiality of private inputs. The next section proves this fact for any monitored execution. In steps 4 and 6 of the example execution, the monitor alters the standard behavior of the program. Section 4.5.2 studies the impact of the monitor on a given category of programs.

4.5 Properties of the Monitoring Mechanism

This section collates the monitor's characteristics to the standard properties of *soundness* with regards to noninterference and, specific to monitors, *transparency* with regards to the output sequence generated. A monitor is transparent with regard to a program behavior if, for any execution, this behavior is the same whether the execution is monitored or not. Section 4.5.1 proves the soundness of the monitoring mechanism with regard to the notion of noninterference. However, as the noninterference problem is statically undecidable, it is of course impossible for the monitor to be complete without running all the executions starting with the same public inputs as the execution currently monitored. As well, it is impossible to have a sound and transparent monitor for noninterference, as with an interfering execution the monitor can be either transparent or sound but not both. However, section 4.5.2 still proves that the monitor is transparent for a nontrivial set of executions.

4.5.1 Soundness

The monitoring mechanism is proved to be sound with regard to the noninterference property for executions. This means that, for any output sequences generated during the monitored execution of any program Θ started in the initial state $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ and value stores σ' low equivalent to σ, there exists a thread interleaving such that during the monitored execution of Θ started in the initial state $\zeta^{\mathcal{I}}_{\Theta,\sigma'}$ the same output sequence is generated. This property is formally stated by theorem 4.5.1.

Theorem 4.5.1 (Monitored Executions are Noninterfering).
For all programs Θ, value stores σ, any monitored execution of Θ started in the initial state $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ is noninterfering. This is formally stated as follows:

$$\forall \Theta, \forall \sigma \in (\mathbb{X} \to \mathbb{D}) \ : \ o \in O[\![\Theta]\!]_{\mathcal{M}(O)} \zeta^{\mathcal{I}}_{\Theta,\sigma} \ \Rightarrow \ ni(\Theta, \mathcal{M}(O), \zeta^{\mathcal{I}}_{\Theta,\sigma}, o)$$

Proof sketch. The proof — which follows directly from lemma B.2.8 — goes by induction on the derivation tree of the current execution (e_c). First, it relies on the fact that, because of the monitoring mechanism, for any evaluation started with the same public inputs there exists an execution (e_2) having a thread interleaving "similar" to the one of the current execution. Additionally, it is demonstrated that, if a thread of e_2 follows a path in its sequential program different from the path followed by the equivalent thread in e_c, nothing will be output by those threads until they reach an "equivalent state". It is then possible to show that e_c and e_2 have the same output sequence.

4.5.2 Partial Transparency

A common property stated for monitors is *transparency*. A transparent monitor is one that does not alter the behavior of the monitored program. An interfering execution has, with regard to confidentiality, a faulty observable behavior. Therefore, a sound monitor enforcing noninterference has to alter the observable behavior of such an execution. Consequently, it is impossible for a sound monitor enforcing noninterference to also be transparent. However, for a precise set of programs, it is possible to prove that the monitoring mechanism presented in this chapter achieves transparency — i.e. it does not alter the observable behavior of any execution of those programs. This set of programs is the set of all programs which are well-typed under a type system similar to the one of Smith and Volpano (Smith and Volpano, 1998).

This type system is described in Figure 4.11. The language used in this chapter includes two structures which do not appear in the language used in (Smith and Volpano, 1998). The two typing rules added for those structures are the only salient differences with the type system of Smith and Volpano (Smith and Volpano, 1998). The lattice of types used has only two elements and is defined using the reflexive relation $\leq$ ($L \leq H$). L is the type for public data and H the type for private data. The typing environment, γ, prescribes types for identifiers and is extended to handle expressions. $\gamma(e)$ is the type of the expression e in the typing environment γ. It is equal to the least upper bound of the types of the free variables appearing in e, or L if there is no free variable in e. It is formally defined as follows:

$$\gamma(e) = \bigsqcup_{x \in FV(e)} \gamma(x)$$

For any sequential program $\mathtt{P}$, if "$\gamma \vdash \mathtt{P} : \tau$ cmd" for some τ and γ in which every secret input is typed secret — i.e. $\forall x \in \mathcal{S}(\mathtt{P}), \gamma(x) = H$ — then $\mathtt{P}$ is said to be well-typed under the typing environment γ. For any concurrent program Θ, if all its threads contain a program well-typed under the same typing environment γ, then Θ is said to be well-typed under this typing environment γ. This is written "$\gamma \vdash \Theta$".

Theorem 4.5.2 states that the monitoring mechanism does not alter executions of well-typed programs. In other words, the monitoring mechanism is transparent for any well-typed program. To understand that the monitor is transparent, and still sound, for a bigger set of programs, consider the concurrent program composed only of this sequential program : `x := h; x :=0;` **output** `x`. `h` is the only secret input. Every execution is noninterfering. But as the type system is flow insensitive, this program is ill-typed. However, the monitoring mechanism does not interfere with the outputs of this program while still guaranteeing that any monitored execution is noninterfering.

Theorem 4.5.2 (Partial Transparency: monitoring preserves *type-safe* programs).
For all programs Θ with secret inputs $\mathcal{S}(\Theta)$, typing environments γ with variables belonging to $\mathcal{S}(\Theta)$ typed secret, and value stores σ, if Θ is well-typed under γ then any monitored execution of Θ started in the initial state $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ outputs a sequence which belongs to the observable behavior of the unmonitored execution of Θ started in the initial state $(\sigma, \emptyset)$. This is formally stated as follows:

$$\gamma \vdash \Theta \quad \Rightarrow \quad O[\![\Theta]\!]_{\mathcal{M}(O)}\zeta^{\mathcal{I}}_{\Theta,\sigma} \; = \; O[\![\Theta]\!]_{O}(\sigma, \emptyset)$$

Proof sketch. The proof follows directly from lemmas B.3.2 and B.3.3 by induction on the length of the derivation of the executions.

$$\frac{\gamma(e) = \tau' \qquad \tau' \leq \tau}{\gamma \vdash e : \tau} \quad \text{(T-EXP)}$$

$$\frac{\gamma(x) = \tau' \qquad \gamma \vdash e : \tau' \qquad \tau \leq \tau'}{\gamma \vdash x \mathrel{\textbf{:=}} e : \tau \text{ cmd}} \quad \text{(T-ASSIGN)}$$

$$\frac{\tau \leq H}{\gamma \vdash \textbf{skip} : \tau \text{ cmd}} \quad \text{(T-SKIP)}$$

$$\frac{\gamma \vdash e : L}{\gamma \vdash \textbf{output}\ e : L \text{ cmd}} \quad \text{(T-OUTPUT)}$$

$$\frac{\gamma \vdash S_1 : \tau \text{ cmd} \qquad \gamma \vdash S_2 : \tau \text{ cmd}}{\gamma \vdash S_1 \,;\, S_2 : \tau \text{ cmd}} \quad \text{(T-SEQ)}$$

$$\frac{\gamma \vdash e : \tau' \qquad \gamma \vdash S_1 : \tau' \text{ cmd} \qquad \gamma \vdash S_2 : \tau' \text{ cmd} \qquad \tau \leq \tau'}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end} : \tau \text{ cmd}} \quad \text{(T-IF)}$$

$$\frac{\gamma \vdash e : L \qquad \gamma \vdash S : L \text{ cmd}}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} : L \text{ cmd}} \quad \text{(T-WHILE)}$$

$$\frac{\gamma \vdash e : L \qquad \gamma \vdash S : L \text{ cmd}}{\gamma \vdash \textbf{with}\ \bar{x}\ \textbf{when}\ e\ \textbf{do}\ S\ \textbf{done} : L \text{ cmd}} \quad \text{(T-SYNC)}$$

Figure 4.11: The type system used for comparison

4.6 Conclusion

This chapter addresses the problem of respecting the secret data confidentiality by concurrent programs. The problem is formalized using the noninterference property which states that a program is safe if and only if the publicly observable program's behavior is not influenced by the values of its private inputs.

This problem is usually addressed using static analyses. To the author's knowledge, the solution proposed in this chapter is the only dynamic analysis *enforcing* noninterference in a concurrent setting. It consists in a monitoring mechanism enforcing the respect of the confidentiality of secret inputs. It is defined as a special semantics communicating with a security automaton. The role of the semantics is to send abstractions of the events occurring during the execution to the automaton. Then, it executes the program in accordance to the answers sent back by the automaton. This latter tracks the information flows and controls — allows, modifies or denies — the execution of output statements and synchronization commands.

As the proposed mechanism deals with noninterference, which is not a property of an execution trace, it significantly differs from usual monitors. The dynamic analysis proposed is required to take into account the contents of branches which are not executed. Additionally, due to the fact that the monitor enforces noninterference on the fly, it is also required to change the position of synchronization commands while still enforcing the thread interleaving constraints induced by the original position of those commands.

Section 4.5 not only proves that this monitoring mechanism is sound, in the sense that it enforces noninterference for any execution, but also that it is transparent for any program well-typed under a type system similar to the one of Smith and Volpano (Smith and Volpano, 1998).

The next chapter will demonstrate that it is possible to be more precise by benefiting even more of the dynamic nature of the analysis. In a dynamic information flow analysis, implicit information flows are detected using a static analysis on the unexecuted branches of conditionals (either at compile time or at run time). In chapter 3 and 4, this static analysis does not take into account the program state at the time the static analysis is run. This is due to the fact that the information flow detection mechanism is handled by a security automaton separated from the monitoring semantics. Next chapter merges the information flow detection mechanism into the monitoring semantics and uses a context sensitive static analysis for implicit indirect flows. This feature allows for a greater precision in the detection of implicit and explicit indirect flows.

Chapter 5

More Context Means More Precision

This chapter proposes a noninterference monitor for sequential programs. This monitor enforces the absence of information flows between the secret inputs and the public outputs of a program. This implies a sound *detection* of information flows and a sound *correction* of forbidden flows during the execution. The monitor relies on a dynamic information flow analysis. For unexecuted pieces of code, this dynamic analysis uses any context sensitive static information flow analysis which respects a given set of three hypotheses. The soundness of the overall monitoring mechanism with regard to noninterference enforcement is proved, as well as its higher precision than the mechanism proposed in Chapter 3. Finally, the chapter proposes some extensions in order to take into account procedures and records.

The content of this chapter has not been published yet.

5.1 Introduction

This chapter presents a secure information flow monitoring mechanism for sequential programs. In Chapters 3 and 4, the monitoring mechanism is implemented by a security automaton. This automaton, analyzing the information flows during the execution, is separated from the semantics actually executing the monitored program. The automaton and the semantics communicate during the execution in order to enforce the confidentiality of secret data manipulated. However, the automaton and the semantics have their own distinct states. The security automaton is unaware of the content of the current program state, and thus of the precise values of the variables. This approach has the advantage of clearly distinguishing the features which are part of the dynamic information flow analysis from those which are part of the program computation. However, this approach also has an impact on the achievable precision of the information flow analysis.

In the program of Figure 5.1, h is the only private input and l is the only public input. This program initializes the value of the internal variable x to `0`; then, depending on the values of the inputs h and l, it sets x to 1; and finally, it outputs the value of x. This program is obviously interfering: if l is true, then it

```
1 x := 0; y := 1;
2 if h then
3   skip
4 else
5   if l then x := y else skip end
6 end;
7 output x
```

Figure 5.1: Example of a program where context sensitivity is important.

outputs `0` if h is true and 1 otherwise. However, executions where l is false are noninterfering; the program always outputs `0`. The monitoring mechanisms presented in Chapters 3 and 4 are unable to achieve a level of precision which allows to detect that executions where l is false are noninterfering. This is because the static analyses used to analyze unexecuted branches of conditionals whose test carries variety are not context sensitive. When h is true, the program "**if** l **then** x **:=**y **else skip end**" is statically analyzed to determine possible implicit indirect flows. In chapters 3 and 4, this analysis is done without any knowledge about the value of the input l. Therefore, the static analysis must take into consideration that x may be assigned in an execution where h has a different (`false`) value and l has the same value (even if the current value of l is false). Moreover, the dynamic analysis must compute the same information flows whatever the value of private inputs in order to prevent the creation of a new covert channel by the correction mechanism (Section 2.1.6). Therefore, even if h is false, and then line 5 is executed, the monitoring semantics can not use its knowledge of the value of l to prevent the analysis of a branch which can in no way be executed by any execution started with the same public inputs.

This chapter presents a monitoring mechanism able to accurately deal with the above example. To do so, implicit indirect flows are taken into account by the dynamic analysis using the results of context-sensitive static analyses of unexecuted pieces of code. The context sensitivity of the static analysis allows it to avoid analyzing pieces of code which will never be executed in any context low-equivalent to the current one — i.e. contexts in which public data have the same values as in the current context. For example, a context-sensitive analysis of the 5th line of Figure 5.1 in a context where l is `false`, and considered public data, does not analyze the then-branch of the conditional.

The dynamic information flow analysis described in this chapter is not limited to the use of a sole context-sensitive static analysis. A whole series of context-sensitive static analyses, characterized by a set of axioms, can be used in combination with the dynamic analysis to accurately track information flows. Those axioms emphasize the two main requirements for a noninterference monitor to *accurately detect* information flows and *safely enforce* noninterference.

Section 5.1.1 describes the syntax and semantics of the sequential imperative language studied. Section 5.1.2 formally defines the execution property enforced by the monitor and introduces the monitoring principles, which are then formalized in Section 5.2. Before stating and proving the main properties of the monitoring mechanism in Section 5.4, an example of monitored execution is detailed in Section 5.3. Preceding the conclusion in Section 5.6, an extension to the studied language is presented in Section 5.5.

5.1.1 The Language: Syntax & Standard Semantics

The language used to describe programs studied in this chapter is similar to the one of Chapter 3. It is an imperative language for sequential programs. This chapter focuses on the increase of precision in the dynamic information flow analysis and not on the correction mechanism.

For simplicity, the language does not include an output statement. Therefore, the notion of *noninterfering execution* used in this chapter (Definition 5.1.1) is not a relation between public inputs and an output sequence. The notion of noninterfering execution is defined as a relation between the public inputs of the program and the values, at the end of the execution, of a set of variables which are considered *observable* once the program has been executed. This definition is closer from the classical noninterference definition than those used in Chapters 3 and 4. It is still possible to simulate output statements by replacing them by assignments to fresh variables. The correction mechanism of previous chapters is still applicable in this setting — i.e. denying any assignment to those fresh variables when the context of execution (program counter) carries variety and replacing the values assigned by a default one when those values carry variety.

Syntax of the Language

The grammar of the language studied is given in Figure 5.2. In this grammar, ⟨*ident*⟩ stands for a variable identifier. ⟨*expr*⟩ is an expression of values and variable identifiers. Expressions in this language are deterministic – their evaluation in a given program state always results in the same value — and are free of side effects — their evaluation has no influence on the program state. Additionally, their evaluation in any program state always terminates.

$$\begin{array}{rcl}
\langle \mathit{prog} \rangle & ::= & \textbf{skip} \\
& | & \langle \mathit{ident} \rangle \ \textbf{:=} \ \langle \mathit{expr} \rangle \\
& | & \langle \mathit{prog} \rangle \ ; \ \langle \mathit{prog} \rangle \\
& | & \textbf{if} \ \langle \mathit{expr} \rangle \ \textbf{then} \ \langle \mathit{prog} \rangle \ \textbf{else} \ \langle \mathit{prog} \rangle \ \textbf{end} \\
& | & \textbf{while} \ \langle \mathit{expr} \rangle \ \textbf{do} \ \langle \mathit{prog} \rangle \ \textbf{done}
\end{array}$$

Figure 5.2: Grammar of the language

A program expressed with this language is either a skip statement (**skip**) which has no effect, an assignment of the value of an expression to a variable (⟨*ident*⟩ **:=** ⟨*expr*⟩), a sequence of programs (⟨*prog*⟩ ; ⟨*prog*⟩), a conditional executing one program — out of two — depending on the value of a given expression (**if** ⟨*expr*⟩ **then** ⟨*prog*⟩ **else** ⟨*prog*⟩ **end**), or a loop executing repetitively a given program as long as a given expression is true (**while** ⟨*expr*⟩ **do** ⟨*prog*⟩ **done**).

Standard Semantics of the language

The standard semantics of the language is given in Figure 5.3. The evaluation symbol ($\Downarrow$) is given a subscript letter in order to distinguish between the standard semantics ($\mathcal{S}$) and the monitoring one ($\mathcal{M}$). The standard semantics is based on rules written in the format:

$$\sigma \vdash P \Downarrow_{\mathcal{S}} \sigma'$$

Those rules mean that, with the initial program state σ, the evaluation of the program P yields the final program state σ'. Let $\mathbb{X}$ be the domain of variable identifiers and $\mathbb{D}$ be the semantic domain of values. A program state is a value store σ ($\mathbb{X} \to \mathbb{D}$) mapping variable identifiers to their respective value. The definition of value stores is extended to expressions, so that $\sigma(e)$ is the value of the expression e in the program state σ.

$$\frac{}{\sigma \vdash \textbf{skip} \Downarrow_{\mathcal{S}} \sigma} \qquad (\text{E}_{\mathcal{S}}\text{-SKIP})$$

$$\frac{}{\sigma \vdash x \mathbin{\textbf{:=}} e \Downarrow_{\mathcal{S}} \sigma[x \mapsto \sigma(e)]} \qquad (\text{E}_{\mathcal{S}}\text{-ASSIGN})$$

$$\frac{\sigma \vdash P^{\text{h}} \Downarrow_{\mathcal{S}} \sigma^{\text{h}} \qquad \sigma^{\text{h}} \vdash P^{\text{t}} \Downarrow_{\mathcal{S}} \sigma^{\text{t}}}{\sigma \vdash P^{\text{h}} \,;\, P^{\text{t}} \Downarrow_{\mathcal{S}} \sigma^{\text{t}}} \qquad (\text{E}_{\mathcal{S}}\text{-SEQUENCE})$$

$$\frac{\sigma(e) = v \qquad \sigma \vdash P^{v} \Downarrow_{\mathcal{S}} \sigma'}{\sigma \vdash \textbf{if } e \textbf{ then } P^{\texttt{true}} \textbf{ else } P^{\texttt{false}} \textbf{ end} \Downarrow_{\mathcal{S}} \sigma'} \qquad (\text{E}_{\mathcal{S}}\text{-IF})$$

$$\frac{\sigma(e) = \texttt{true} \qquad \sigma \vdash P^{\text{l}} \,;\, \textbf{while } e \textbf{ do } P^{\text{l}} \textbf{ done} \Downarrow_{\mathcal{S}} \sigma'}{\sigma \vdash \textbf{while } e \textbf{ do } P^{\text{l}} \textbf{ done} \Downarrow_{\mathcal{S}} \sigma'} \qquad (\text{E}_{\mathcal{S}}\text{-WHILE}_{\texttt{true}})$$

$$\frac{\sigma(e) = \texttt{false}}{\sigma \vdash \textbf{while } e \textbf{ do } P^{\text{l}} \textbf{ done} \Downarrow_{\mathcal{S}} \sigma} \qquad (\text{E}_{\mathcal{S}}\text{-WHILE}_{\texttt{false}})$$

Figure 5.3: Rules of the standard semantics

5.1.2 Monitoring Principles

The remainder of the current section gives some formal definitions which are used to formalize the execution property enforced by the monitoring mechanism. Subsequently, it introduces succinctly the mechanisms used by the monitoring semantics in order to enforce secure information flows.

A “safe” execution is a noninterfering execution. Unlike languages of Chapters 3 and 4, the language studied in this chapter does not include an output statement. Therefore, in this chapter, noninterference is not defined as the absence of *strong dependencies* (Cohen, 1977) between the secret (or private) inputs and a sequence of outputs, as done in the two previous chapters. Rather, as commonly done, noninterference is defined as the absence of strong dependencies between the secret inputs of an execution and the final values of some variables which are considered to be publicly observable at the end of the execution.

For every execution of a given program P, two sets of variable identifiers are defined. The set of variables corresponding to the secret inputs of the program is designated by $\mathcal{S}(\mathtt{P})$. The set of variables whose final value are publicly observable at the end of the execution is designated by $\mathcal{O}(\mathtt{P})$. No requirements are put on $\mathcal{S}(\mathtt{P})$ and $\mathcal{O}(\mathtt{P})$ other than requiring them to be subsets of $\mathbb{X}$. A variable x is even allowed to belong to both sets. In such a case, in order to be noninterfering, the program P would be required to, at least, reset the value of x. For example, a variable can belong to both sets if, at the beginning of the execution, the variable is used as a container for a secret data, and later is used to store other data which are not considered secret.

In the following definitions, we consider that a program state may contain more than just a value store — e.g. program states of the semantics given in Figure 5.4 also contain a “tag store”. This is the reason why a distinction is done between program states, whose generic notation used is X, and value stores (σ). Following Definition 4.1.3, two program states X_1, respectively X_2, containing the value stores σ_1, respectively σ_2, are *low equivalent* with regards to a set of variables V, written $X_1 \overset{V}{=} X_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2.

Definition 5.1.1 (Noninterfering Execution).
Let $\Downarrow_s$ denote a big-step semantics. Let $\mathcal{S}(P)^c$ be the complement of $\mathcal{S}(P)$ in the set $\mathbb{X}$. For all programs $\mathtt{P}$, program states X_1 and X_1', an execution with the semantics $\Downarrow_s$ of the program $\mathtt{P}$ in the initial state X_1 and yielding the final state X_1' is noninterfering, written $ni(\mathtt{P}, s, X_1)$, if and only if, for every program states X_2 and X_2' such that the execution with the semantics $\Downarrow_s$ of the program $\mathtt{P}$ in the initial state X_2 yields the final state X_2':

$$X_1 \overset{\mathcal{S}(P)^c}{=} X_2 \Rightarrow X_1' \overset{\mathcal{O}(P)}{=} X_2'$$

The monitoring mechanism is based on a flow and context sensitive static analysis. As with the automaton-based noninterference monitors of Chapters 3 and 4, the monitoring semantics treats directly the direct and explicit indirect flows. For implicit indirect flows, a static analysis is run on the unexecuted branch of every conditional whose test carries variety — i.e. is influenced by the secret inputs of the program.

The static analysis is context sensitive. An unexecuted branch P is analyzed in the context of the program state at the time the test of the conditional, to which P belongs, has been evaluated. The static analysis is then aware of the exact value of the variables which do not carry variety. During the analysis, the context is modified to reflect loss of knowledge — i.e. the exact value of variables at the different program points. The static analysis does not compute the values of variables. Therefore, when analyzing an assignment to a variable x, the context of the static analysis is modified to reflect the fact that the static analysis does not anymore have knowledge of the precise value of the variable x. When analyzing a conditional whose test value can be computed using the current context, only the branch designated by the test is analyzed. As the value of any variable which does not carry variety depends only on the public inputs, branches which are not designated by the test value would never be executed by any execution started with the same public inputs as the monitored execution. As the static analysis detects implicit indirect flows more accurately than context insensitive analyses, explicit indirect flows can also be treated more accurately. Recall from Section 2.2.4 that implicit indirect flows and explicit indirect flows must be treated with the same precision in order to prevent the creation of a new covert channel (Le Guernic and Jensen, 2005).

For the example on page 78, if h is true then "**if** l **then** x **:=** y **else** **skip** **end**" is analyzed using as context the program state before the evaluation of line 2. As l is a public input, it does not carry variety and the static analysis can take into consideration its value in the provided context. If l is true then the static analysis would analyze the assignment and conclude that there is an implicit indirect flow from h to x. Otherwise, the static analysis would only analyze the statement "**skip**". It would then accurately conclude that there is no "bad" flows in an execution where l is false.

This paragraph proposes a simple correction mechanism. As this chapter focuses on the increase of precision of the dynamic analysis, the proposed correction mechanism is kept as simple as possible. The notion of noninterfering execution used relates the initial and final value stores. Therefore, for simplicity, the proposed correction mechanism is applied on the final value store just before terminating the execution. It consists in resetting the final value of any observable variable — i.e. belonging to $\mathcal{O}(\mathtt{P})$ — which may carry variety. Theorem 5.4.2 ensures that applying this correction mechanism with the dynamic noninterference analysis of this chapter does not create any new covert channel.

The remain of this chapter focuses on the dynamic noninterference analysis computing which variables may carry variety at the end of the execution. The combination of this dynamic analysis and of the standard semantics of the language is "abusively" called *monitoring semantics* even if it does not include the correction mechanism proposed above. However, the correction mechanism of a noninterference monitor is quite independent from the dynamic noninterference analysis used as long as a theorem similar to Theorem 5.4.2 holds when the correction mechanism is applied. The next section formalizes the mechanisms presented above. It starts by presenting the monitoring semantics making use of static information flow analyses. Then, it describes which static analyses can be used in combination with this monitoring semantics.

5.2 The Monitoring Semantics

In previous chapters, the dynamic information flow analysis is defined independently of the monitoring semantics. Therefore, the static analysis used by the dynamic analysis on unexecuted branches can not be sensitive to the context of the monitoring semantics (which is inaccessible to the dynamic analysis). In this chapter, the dynamic information flow and the monitoring semantics are intrinsically linked. This allows the analysis of unexecuted branches with a static analysis sensitive to the context of execution. The dynamic information flow analysis and the monitoring semantics are defined together in Figure 5.4.

In previous chapters, information flows are tracked using a set of variables. At any execution step, this set contains the variables that may carry *variety* — i.e. whose value may be influenced by the secret inputs of the program. In this chapter, information flows are tracked using tags. At any execution step, every variable has a tag which reflects the fact that this variable may carry variety or not.

5.2.1 A semantics Making Use of a Context-sensitive Static Analysis

Let $\mathbb{X}$ be the domain of variable identifiers, $\mathbb{D}$ be the semantic domain of values, and $\mathbb{T}$ be the domain of tags. In the remainder of this chapter, $\mathbb{T}$ is equal to $\{\top, \bot\}$. Those tags form a lattice such that $\bot \sqsubset \top$. $\top$ is the tag associated to variables that *may* carry variety. $\bot$ is the tag associated to variables that *definitely do not* carry variety.

The monitoring semantics described in Figure 5.4 is based on rules written in the format:

$$\zeta,\ t^{\mathrm{pc}}\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ \zeta'$$

This reads as follows: in the monitoring execution state ζ, with a program counter tag equal to t^{pc}, program $\mathtt{P}$ yields the monitoring execution state ζ'. The program counter tag (t^{pc}) is a tag which reflects the security level of the information carried by the control flow. A monitoring execution state ζ is a pair $(\sigma,\ \rho)$ composed of a value store σ and a tag store ρ. A value store ($\mathbb{X} \to \mathbb{D}$) maps variable identifiers to values. A tag store ($\mathbb{X} \to \mathbb{T}$) maps variable identifiers to tags. The definitions of value store and tag store are extended to expressions. $\sigma(e)$ is the value of the expression e in a program state whose value store is σ. Similarly, $\rho(e)$ is the tag of the expression e in a program state whose tag store is ρ. $\rho(e)$ is formally defined as follows, with $FV(e)$ being the set of free variables appearing in the expression e:

$$\rho(e) \quad = \quad \bigsqcup_{x \in FV(e)} \rho(x)$$

The least upper bound operator ($\sqcup$) is also extended in order to apply to functions — i.e. relations (sets of pairs) which do not contain two pairs having the same first element. The least upper bound (lub) of two functions is the function: whose domain is the union of the domains of the two parameter functions, whose range is the union of the ranges of the two parameter functions, and which associates, to any element x of its domain, the lub of the elements associated to x by the two parameter functions. Its formal definition follows.

$$f \sqcup g \quad = \quad \left\{ (x, y) \;\middle|\; x \in (\mathrm{dom}(f) \cup \mathrm{dom}(g)) \ \wedge\ y = \left\{ \begin{array}{ll} f(x) \sqcup g(x) & \text{iff } x \in \mathrm{dom}(f) \cap \mathrm{dom}(g) \\ f(x) & \text{iff } x \in \mathrm{dom}(f) \setminus \mathrm{dom}(g) \\ g(x) & \text{iff } x \in \mathrm{dom}(g) \setminus \mathrm{dom}(f) \end{array} \right. \right\}$$

The semantic rules make use of static analyses results. In Figure 5.4, application of a static information flow analysis to the piece of code $\mathtt{P}$ in the context ζ is written: $[\![\zeta\ \vdash\ \mathtt{P}]\!]^{\sharp_{\mathcal{G}}}$. The analysis of a program $\mathtt{P}$ in a monitoring execution state ζ must return a pair $(\mathfrak{D}, \mathfrak{X})$. $\mathfrak{D}$, which belongs to $\mathcal{P}(\mathbb{X} \to \mathbb{X})$, is an over-approximation of the dependencies between the initial and final values of the variables created by an execution of $\mathtt{P}$ in the context ζ. $\mathfrak{D}(x)$, which is equal to $\{y \mid (x, y) \in \mathfrak{D}\}$, is the set of variables whose initial value may influence the final value of x after execution of P in the context ζ. $\mathfrak{X}$, which belongs to $\mathcal{P}(\mathbb{X})$, is an over-approximation of the set of variables which are potentially updated in an execution of P in the context ζ. This static information flow analysis can be any such analysis that satisfies a set of formal constraints which are stated below.

For example, let $\mathtt{P}^5$ be the 5th line of the program in Figure 5.1. $(\emptyset,\ \emptyset)$ is a valid result of the static analysis of $\mathtt{P}^5$ in a context where l is `false` and does not carry variety. However, it is not a valid result in a context where l is `true` or may carry variety. In such a context, $(\{(x,y),(x,l)\},\ \{x\})$ is a valid result of the static analysis of $\mathtt{P}^5$.

The monitoring semantic rules are given in Figure 5.4. As can be expected, the execution of a **skip** statement yields a final state equal to the initial state. The monitored execution of the assignment of the value of the expression e to the variable x yields a monitored execution state $(\sigma',\ \rho')$. The final value store (σ') is equal to the initial value store (σ) except for the variable x. The final value store maps the variable x to the value of the expression e evaluated with the initial value store $(\sigma(e))$. Similarly, the final tag store (ρ') is equal to the initial tag store (ρ) except for the variable x. The tag of x after the execution of the assignment is the least upper bound of the program counter tag (t^{pc}) and the tag of the expression computed using the initial tag store $(\rho(e))$. $\rho(e)$ represent the level of the information flowing into x through direct flows. t^{pc} represent the level of the information flowing into x through explicit indirect flows.

The monitored execution of a conditional whose test expression does not carry variety $(\rho(e) = \bot)$ follows the same scheme as with a standard semantics. For a conditional whose test expression e carries variety, the branch (P^v) designated by the value of e is executed and the other one $(P^{\neg v})$ is analyzed. The final value store is the one returned by the execution of P^v. The final tag store (ρ') is the least upper bound of the tag store returned by the execution of P^v and two new tag stores $(\rho^{\neg v}$ and $\rho^e)$ generated from the result of the analysis of $P^{\neg v}$ $((\mathfrak{D},\mathfrak{X}))$. By definition, $\rho \sqcup \rho'$ is equal to $\lambda x.\rho(x) \sqcup \rho'(x)$. The first new tag store $(\rho^{\neg v})$ is computed from the dependencies $(\mathfrak{D})$, were $P^{\neg v}$ executed, between the final value of variables in $P^{\neg v}$ and their initial values. In $\rho^{\neg v}$, the tag of a variable x is the least upper bound of the initial tags $(\rho(y))$ of the variables whose initial value may influence the final value of x in an execution of $P^{\neg v}$. The other new tag store (ρ^e) reflects the implicit indirect flows between the value of the test of the conditional and the variables $(\mathfrak{X})$ which may be updated in an execution of $P^{\neg v}$. In ρ^e, the tag of a variable x is equal to the initial tag of the test expression of the conditional $(\rho(e))$ if and only if x belongs to $\mathfrak{X}$; otherwise, its tag is $\bot$.

Which static information flow analyses can be used?

The static analysis used to determine the information flows in unexecuted branches is not formally defined. In fact, the dynamic analysis can use any static information flow analysis which complies with the three following hypotheses and returns a pair, whose first element is a relation between variables — i.e. a set of pairs of variables — and second element is a set of variables.

The first hypothesis ensures a valid detection of information flows. Hypothesis 5.2.1 simply requires the static analysis used to be a *sound* information flow analysis. More precisely, it requires that the second

$$\frac{}{\zeta,\ t^{\mathrm{pc}} \vdash \mathbf{skip} \Downarrow_{\mathcal{M}} \zeta} \qquad (\mathrm{E}_{\mathcal{M}}\text{-SKIP})$$

$$\frac{}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \vdash x \mathbf{:=} e \Downarrow_{\mathcal{M}} (\sigma[x \mapsto \sigma(e)],\ \rho[x \mapsto \rho(e) \sqcup t^{\mathrm{pc}}])} \qquad (\mathrm{E}_{\mathcal{M}}\text{-ASSIGN})$$

$$\frac{\zeta,\ t^{\mathrm{pc}} \vdash P^{\mathrm{h}} \Downarrow_{\mathcal{M}} \zeta^{\mathrm{h}} \qquad \zeta^{\mathrm{h}},\ t^{\mathrm{pc}} \vdash P^{\mathrm{t}} \Downarrow_{\mathcal{M}} \zeta^{\mathrm{t}}}{\zeta,\ t^{\mathrm{pc}} \vdash P^{\mathrm{h}}\ ;\ P^{\mathrm{t}} \Downarrow_{\mathcal{M}} \zeta^{\mathrm{t}}} \qquad (\mathrm{E}_{\mathcal{M}}\text{-SEQUENCE})$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = v \qquad (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \rho(e) \vdash P^{v} \Downarrow_{\mathcal{M}} \zeta'}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \vdash \mathbf{if}\ e\ \mathbf{then}\ P^{\mathtt{true}}\ \mathbf{else}\ P^{\mathtt{false}}\ \mathbf{end} \Downarrow_{\mathcal{M}} \zeta'} \qquad (\mathrm{E}_{\mathcal{M}}\text{-IF}_{\bot})$$

$$\frac{\begin{array}{c} \rho(e) = \top \qquad \sigma(e) = v \\ (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \rho(e) \vdash P^{v} \Downarrow_{\mathcal{M}} (\sigma^{v},\ \rho^{v}) \qquad [\![(\sigma,\ \rho) \vdash P^{\neg v}]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X}) \\ \rho^{\neg v} = \lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y) \qquad \rho^{\mathrm{e}} = (\mathfrak{X} \times \{\rho(e)\}) \cup ((\mathbb{X} - \mathfrak{X}) \times \{\bot\}) \end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \vdash \mathbf{if}\ e\ \mathbf{then}\ P^{\mathtt{true}}\ \mathbf{else}\ P^{\mathtt{false}}\ \mathbf{end} \Downarrow_{\mathcal{M}} (\sigma^{v},\ \rho^{v} \sqcup \rho^{\neg v} \sqcup \rho^{\mathrm{e}})} \qquad (\mathrm{E}_{\mathcal{M}}\text{-IF}_{\top})$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = \mathtt{false}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \vdash \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \Downarrow_{\mathcal{M}} (\sigma,\ \rho)} \qquad (\mathrm{E}_{\mathcal{M}}\text{-WHILE}_{skip})$$

$$\frac{\begin{array}{c} \rho(e) = \bot \qquad \sigma(e) = \mathtt{true} \\ (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \rho(e) \vdash P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \Downarrow_{\mathcal{M}} \zeta' \end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \vdash \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \Downarrow_{\mathcal{M}} \zeta'} \qquad (\mathrm{E}_{\mathcal{M}}\text{-WHILE}_{true_{\bot}})$$

$$\frac{\begin{array}{c} \rho(e) = \top \qquad \sigma(e) = \mathtt{true} \\ (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \rho(e) \vdash P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \Downarrow_{\mathcal{M}} (\sigma',\ \rho') \end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \vdash \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \Downarrow_{\mathcal{M}} (\sigma',\ \rho \sqcup \rho')} \qquad (\mathrm{E}_{\mathcal{M}}\text{-WHILE}_{true_{\top}})$$

$$\frac{\begin{array}{c} \rho(e) = \top \qquad \sigma(e) = \mathtt{false} \\ [\![(\sigma,\ \rho) \vdash P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X}) \\ \rho^{\mathrm{l}} = \lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y) \qquad \rho^{\mathrm{e}} = (\mathfrak{X} \times \{\rho(e)\}) \cup ((\mathbb{X} - \mathfrak{X}) \times \{\bot\}) \end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \vdash \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \Downarrow_{\mathcal{M}} (\sigma,\ \rho \sqcup \rho^{\mathrm{l}} \sqcup \rho^{\mathrm{e}})} \qquad (\mathrm{E}_{\mathcal{M}}\text{-WHILE}_{false_{\top}})$$

Figure 5.4: Rules of the monitoring semantics

element of the static analysis result ($\mathfrak{X}$) contains all the variables which may be updated by an execution of the analyzed program with the same public values as those used for the analysis. This is a straightforward requirement as the result of the static analysis is used to take into account implicit indirect flows.

For example, in a context where l is `true` or may carry variety ($\rho(l) = \top$), Hypothesis 5.2.1 requires that x *must* belong to the second element ($\mathfrak{X}$) of the result of any analysis of $\mathtt{P}^5$ (the 5th line of the program in Figure 5.1).

Hypothesis 5.2.1 (Correctness of the static analysis for information flow detection).
For all monitoring execution states $(\sigma,\ \rho)$, $(\sigma_i,\ \rho_i)$ *and* $(\sigma_o,\ \rho_o)$, *program counter tags* t^{pc}, *programs* P, *and analysis results* $(\mathfrak{D}, \mathfrak{X})$ *such that:*

1. $\forall x : (\rho(x) = \bot) \Rightarrow \sigma_i(x) = \sigma(x)$,
2. $(\sigma_i,\ \rho_i),\ t^{pc}\ \vdash\ P\ \Downarrow_{\mathcal{M}}\ (\sigma_o,\ \rho_o)$,

the following holds:

$$[\![(\sigma,\ \rho) \vdash P]\!]^{\sharp_\mathcal{G}} = (\mathfrak{D}, \mathfrak{X}) \quad \Rightarrow \quad \forall x \notin \mathfrak{X}.\ \sigma_o(x)\ =\ \sigma_i(x)$$

The second and third hypotheses allow a "safe" correction mechanism. They ensure that a simple "bug-free" correction mechanism based on the results of the overall dynamic analysis, as the one proposed on page 82, does not create any new covert channel. Hypothesis 5.2.2 requires the static information flow analysis to be as precise as the dynamic information flow analysis when the program counter carries variety. This is required in order to be able to correct "bad" flows. If this hypothesis does not hold then it may be possible for the attacker to deduce from the behavior of the correction mechanism which branch of a conditional whose test carries variety has been analyzed and whose branch has been executed. This information would allow the attacker to deduce the value of the test of the conditional.

Hypothesis 5.2.2 (Correctness of the static analysis for information flow correction).
For all monitoring execution states $(\sigma,\ \rho)$ *and* $(\sigma',\ \rho')$, *programs* P, *and analysis results* $(\mathfrak{D}, \mathfrak{X})$ *such that:*

1. $(\sigma,\ \rho),\ \top\ \vdash\ P\ \Downarrow_{\mathcal{M}}\ (\sigma',\ \rho')$,
2. $[\![(\sigma,\ \rho) \vdash P]\!]^{\sharp_\mathcal{G}} = (\mathfrak{D}, \mathfrak{X})$,

the following holds:

$$\rho' \quad = \quad (\ \lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)\ ) \quad \sqcup \quad (\ (\mathfrak{X} \times \{\top\})\ \cup\ (\mathfrak{X}^c \times \{\bot\})\ )$$

The last hypothesis (Hypothesis 5.2.3) requires the result of the static analysis to be determined solely by the value of public data. This hypothesis is needed for a similar reason than for the second hypothesis. In order to prevent the correction mechanism to be used as a covert channel, the result of the static analysis must be independent from the value of secret data.

For example, Hypothesis 5.2.3 requires that the result of the analysis of the 5th line of the program in Figure 5.1 must depend only on the values of variables l, x and y; and not on the value of variable h.

Hypothesis 5.2.3 (Static analysis is deterministic with regard to public data).
For all monitoring execution states $(\sigma,\ \rho)$ and $(\sigma',\ \rho')$, and programs P *such that:*

1. $\rho = \rho'$,
2. $\forall x : (\rho(x) \sqsubseteq \bot) \Rightarrow \sigma(x) = \sigma'(x)$,

the following holds:

$$[\![(\sigma,\ \rho) \vdash P]\!]^{\sharp_{\mathcal{G}}} \ = \ [\![(\sigma',\ \rho') \vdash P]\!]^{\sharp_{\mathcal{G}}}$$

The fact that the static analysis is deterministic with regard to public data is not strictly required. The least requirement is that the set of potential results must be determined by the value of public data. For simplicity, this hypothesis also requires the analysis to be deterministic with regard to public data.

5.2.2 A Finer Characterization of *Usable* Static Analyses

The above hypotheses define which static information flow analyses are *usable*, i.e. which static analyses can be used with the monitoring semantics given in Figure 5.4. However, Hypotheses5.2.1 and 5.2.2 are stated using the monitoring semantics itself. This make it more difficult to prove that a given static analysis satisfies those hypotheses.

Figure 5.5 defines a set of *acceptability* rules. The result $(\mathfrak{D}, \mathfrak{X})$ of a static information flow analysis of a given program (P) in a given context — i.e. monitoring execution state — (ζ) is *acceptable* for the monitoring semantics only if the result satisfies those rules. This is written in the format:

$$(\mathfrak{D}, \mathfrak{X}) \models (\zeta \vdash \mathrm{P})$$

In the definitions of those rules, $\mathcal{I}d$ denotes the identity relation. $\circ$ is the operation of composition of relations. It belongs to $(\mathcal{P}(\mathbb{B}\times\mathbb{C})\times\mathcal{P}(\mathbb{A}\times\mathbb{B})) \to \mathcal{P}(\mathbb{A}\times\mathbb{C})$ with $\mathbb{A}$, $\mathbb{B}$ and $\mathbb{C}$ being sets. Its formal definition follows.

$$(S \circ R) = \bigcup_{(a,b)\in R} \{(a,c) \mid (b,c) \in S\}$$

As used in the acceptability rules, $\mathbb{A}$, $\mathbb{B}$ and $\mathbb{C}$ are equal to the set of variable identifiers $\mathbb{X}$. $\{\![S^{\mathtt{true}}, S^{\mathtt{false}}]\!\}^t_v$ returns either the set $S^{\mathtt{true}}$, the set $S^{\mathtt{false}}$ or the union of both depending on the value of the tag t and the value v. Its formal definition follows.

$$\{\![S^{\mathtt{true}}, S^{\mathtt{false}}]\!\}^t_v = \begin{cases} S^{\mathtt{true}} \cup S^{\mathtt{false}} & \text{iff } t = \top \\ S^v & \text{iff } t = \bot \end{cases}$$

Using the acceptability rules of Figure 5.5, it is possible to characterize some static information flow analyses which are *usable* with the monitoring semantics without referring to the monitoring semantics itself. It is also possible to generate a *usable* static information flow analysis by fix-point computation on the acceptability rules; in fact, only on the rule for loop statements. However, those acceptability rules do not define a most precise usable static analysis.

$$(\mathfrak{D},\mathfrak{X}) \models (\ (\sigma,\ \rho) \vdash \textbf{skip}\ )$$
$$\text{iff}\quad \mathfrak{D} = \mathcal{I}d \quad \wedge \quad \mathfrak{X} = \emptyset$$

$$(\mathfrak{D},\mathfrak{X}) \models (\ (\sigma,\ \rho) \vdash x \mathrel{\textbf{:=}} e\ )$$
$$\text{iff}\quad \mathfrak{D} = \mathcal{I}d[x \mapsto FV(e)] \quad \wedge \quad \mathfrak{X} = \{x\}$$

$$(\mathfrak{D},\mathfrak{X}) \models \Big(\ (\sigma,\ \rho) \vdash \mathtt{P}^{\mathrm{h}}\ ;\ \mathtt{P}^{\mathrm{t}}\ \Big)$$
$$\begin{array}{ll}\text{iff} & \text{there exist } (\mathfrak{D}^{\mathrm{h}},\mathfrak{X}^{\mathrm{h}}) \text{ and } (\mathfrak{D}^{\mathrm{t}},\mathfrak{X}^{\mathrm{t}}) \text{ such that:}\\ & \quad (\mathfrak{D}^{\mathrm{h}},\mathfrak{X}^{\mathrm{h}}) \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathrm{h}})\\ & \quad \text{let } \rho' = \rho \sqcup ((\mathfrak{X}^{\mathrm{h}} \times \top) \cup ((\mathfrak{X}^{\mathrm{h}})^{c} \times \bot)) \text{ in } (\mathfrak{D}^{\mathrm{t}},\mathfrak{X}^{\mathrm{t}}) \models ((\sigma,\rho') \vdash \mathtt{P}^{\mathrm{t}})\\ & \quad \mathfrak{D} = \mathfrak{D}^{\mathrm{h}} \circ \mathfrak{D}^{\mathrm{t}} \quad \wedge \quad \mathfrak{X} = \mathfrak{X}^{\mathrm{h}} \cup \mathfrak{X}^{\mathrm{t}}\end{array}$$

$$(\mathfrak{D},\mathfrak{X}) \models \Big(\ (\sigma,\ \rho) \vdash \textbf{if } e \textbf{ then } \mathtt{P}^{\mathtt{true}} \textbf{ else } \mathtt{P}^{\mathtt{false}} \textbf{ end}\ \Big)$$
$$\begin{array}{ll}\text{iff} & \text{there exist } (\mathfrak{D}^{\mathtt{true}},\mathfrak{X}^{\mathtt{true}}) \text{ and } (\mathfrak{D}^{\mathtt{false}},\mathfrak{X}^{\mathtt{false}}) \text{ such that:}\\ & \quad (\mathfrak{D}^{\mathtt{true}},\mathfrak{X}^{\mathtt{true}}) \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathtt{true}})\\ & \quad (\mathfrak{D}^{\mathtt{false}},\mathfrak{X}^{\mathtt{false}}) \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathtt{false}})\\ & \quad \mathfrak{D} = \{\!\!\{\mathfrak{D}^{\mathtt{true}},\mathfrak{D}^{\mathtt{false}}\}\!\!\}^{\rho(e)}_{\sigma(e)} \cup (\{\!\!\{\mathfrak{X}^{\mathtt{true}},\mathfrak{X}^{\mathtt{false}}\}\!\!\}^{\rho(e)}_{\sigma(e)} \times FV(e))\\ & \quad \mathfrak{X} = \{\!\!\{\mathfrak{X}^{\mathtt{true}},\mathfrak{X}^{\mathtt{false}}\}\!\!\}^{\rho(e)}_{\sigma(e)}\end{array}$$

$$(\mathfrak{D},\mathfrak{X}) \models \Big(\ (\sigma,\ \rho) \vdash \textbf{while } e \textbf{ do } \mathtt{P}^{\mathrm{l}} \textbf{ done}\ \Big)$$
$$\begin{array}{ll}\text{iff} & \text{there exists } (\mathfrak{D}^{\mathrm{l}},\mathfrak{X}^{\mathrm{l}}) \text{ such that:}\\ & \quad \text{let } \rho' = \rho \sqcup ((\mathfrak{X}^{\mathrm{l}} \times \top) \cup ((\mathfrak{X}^{\mathrm{l}})^{c} \times \bot)) \text{ in } (\mathfrak{D}^{\mathrm{l}},\mathfrak{X}^{\mathrm{l}}) \models ((\sigma,\rho') \vdash \mathtt{P}^{\mathrm{l}})\\ & \quad \mathfrak{D} = \{\!\!\{\mathfrak{D}^{\mathrm{l}} \circ (\mathfrak{D} \cup \mathcal{I}d), \mathcal{I}d\}\!\!\}^{\rho(e)}_{\sigma(e)} \quad \wedge \quad \mathfrak{X} = \{\!\!\{\mathfrak{X}^{\mathrm{l}}, \emptyset\}\!\!\}^{\rho(e)}_{\sigma(e)}\end{array}$$

Figure 5.5: Acceptability rules for *usable* static information flow analysis results

As stated by Theorem 5.2.1, any *acceptable* static information flow analysis result satisfies Hypothesis 5.2.1.

Theorem 5.2.1 (Acceptability rules imply detection abilities).
For all monitoring execution states ζ, programs P, and analysis result $(\mathfrak{D},\mathfrak{X})$ *such that* $(\mathfrak{D},\mathfrak{X}) \models (\zeta \vdash P)$*, the Hypothesis 5.2.1 holds.*

Proof. The theorem follows directly from Lemma C.1.1.

Theorem 5.2.2 states that any *acceptable* static information flow analysis result, which satisfies Hypothesis 5.2.4, satisfies Hypothesis 5.2.2. Hypothesis 5.2.4 requires the dependencies between initial and final values computed by the analysis to be coherent with the set of potentially updated variables also computed by the analysis. The final value of a variable which may not be updated by a program is required to depend only on its own initial value. Hypothesis 5.2.4 holds if, but not only if, $\mathfrak{D}$ is the smallest fixed point obtained from the acceptability rules of Figure 5.5.

Hypothesis 5.2.4 (Static analysis results are coherent).
For all monitoring execution states (σ, ρ)*, programs* P*, and analysis result* $(\mathfrak{D}, \mathfrak{X})$*:*

$$[\![\zeta \vdash P]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X}) \quad \Rightarrow \quad \forall x \in \mathfrak{X}^c. \quad \mathfrak{D}(x) = \{x\}$$

Theorem 5.2.2 (Acceptability rules imply correction abilities).
For all monitoring execution states (σ, ρ)*, programs* P*, and analysis result* $(\mathfrak{D}, \mathfrak{X})$ *if* $(\mathfrak{D}, \mathfrak{X}) \models (\zeta \vdash P)$ *and the Hypothesis 5.2.4 holds then the Hypothesis 5.2.2 holds.*

Proof. The theorem follows directly from Lemma C.1.4.

Therefore, a static information flow analysis, which satisfies Hypotheses 5.2.3 and 5.2.4 and whose results are acceptable ($[\![\zeta \vdash \mathtt{P}]\!]^{\sharp_{\mathcal{G}}} \models (\zeta \vdash \mathtt{P})$), is *usable* by the monitoring semantics — i.e. it satisfies Hypotheses 5.2.1, 5.2.2 and 5.2.3.

Theorem 5.2.3 (Existence of a compliant static analysis). *There exists a static analysis* $[\![]\!]^{\sharp_{\mathcal{G}}}$ *such that, for all commands* C*, value stores* σ*, and tag stores* ρ*, there exists an analysis result* $(\mathfrak{D}, \mathfrak{X})$ *such that* $[\![\sigma, \rho \vdash C]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X})$ *and Hypotheses 5.2.3 and 5.2.4 hold.*

Proof. The theorem follows directly from the constructive proof of Lemma C.3.2.

5.3 Example of Monitored Execution

Figure 5.6 contains an adaptation of the motivating example given in Section 5.1. In this program ($\mathtt{P}$), variables h and l contain the program inputs at the beginning of the execution. h contains the only secret input ($\mathcal{S}(\mathtt{P}) = \{h\}$); l contains a public input. The only publicly observable behavior of the program is the value of the variable x at the end of the computation ($\mathcal{O}(\mathtt{P}) = \{x\}$). The final value of x is `1` if h is `false` and l is `true`. Otherwise, the final value of x is `0`. This program is obviously interfering as, if the final value of

```
x := 0;
if h then
  skip
else
  if l then x := 1 else skip end
end
```

Figure 5.6: The motivating program of the Introduction.

x is `1`, it is possible to deduce that the secret input h is `true`. However, any execution where l is `false` sets the final value of x to `0`. Therefore, any execution where l is `false` is noninterfering.

As stated in the introduction (Section 5.1), none of the monitoring mechanisms of Chapters 3 and 4 are able to detect that executions where l is `false` are noninterfering. The monitor proposed in this chapter is able to detect it. This section presents the behavior of the semantics presented in Figure 5.4 on executions of the program in Figure 5.6.

5.3.1 What happens when variable h is `true`?

Figure 5.7 presents the evaluation tree for executions where h is `true`. Such executions start by setting the value of x to `0`. Then, as h is `true`, the then-branch (**skip**) is executed and the else-branch (**if** l **then** $x := 1$ **else skip end**) is analyzed. In Figure 5.7, the result of the static analysis of the else-branch is represented by the wild card $\mathfrak{R}$.

$$\dfrac{\dfrac{}{\zeta_1,\ \bot\ \vdash\ x := \mathtt{0}\ \Downarrow_{\mathcal{M}}\ \zeta_2} \qquad \dfrac{\dfrac{}{\zeta_2,\ \top\ \vdash\ \textbf{skip}\ \Downarrow_{\mathcal{M}}\ \zeta_2} \qquad [\![\zeta_2 \vdash \mathtt{P}^\sharp]\!]^{\sharp_{\mathcal{G}}} = \mathfrak{R}}{\zeta_2,\ \bot\ \vdash\ \textbf{if } h \textbf{ then skip else } \mathtt{P}^\sharp \textbf{ end}\ \Downarrow_{\mathcal{M}}\ \zeta_3}}{\zeta_1,\ \bot\ \vdash\ x := \mathtt{0}\ ;\ \textbf{if } h \textbf{ then skip else } \mathtt{P}^\sharp \textbf{ end}\ \Downarrow_{\mathcal{M}}\ \zeta_3}$$

Figure 5.7: Evaluation tree of the program in Figure 5.6 for h = `true`.

And variable l is `true`? Table 5.1 shows the values of the wild cards used in Figure 5.7 (ζ_1, ζ_2, ζ_3, $\mathtt{P}^\sharp$ and $\mathfrak{R}$) for an execution where the initial value of h is `true`, the initial value of l is `true` and the initial value of x is 3.

Wild Card	Means	Value
ζ_1	$(\sigma_1,\ \rho_1)$	$(\ [h \mapsto \mathtt{true},\ l \mapsto \mathtt{true},\ x \mapsto 3],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot]\ )$
ζ_2	$(\sigma_2,\ \rho_2)$	$(\ [h \mapsto \mathtt{true},\ l \mapsto \mathtt{true},\ x \mapsto \mathtt{0}],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot]\ )$
$\mathtt{P}^\sharp$	**if** l **then** $x := 1$ **else skip end**	
$\mathfrak{R}$	$(\mathfrak{D}_l, \mathfrak{X}_l)$	$(\mathcal{I}d[x \mapsto \{l\}],\ \{x\})$
ζ_3	$(\sigma_3,\ \rho_3)$	$(\ [h \mapsto \mathtt{true},\ l \mapsto \mathtt{true},\ x \mapsto \mathtt{0}],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \top]\ )$

Table 5.1: Value of the wild cards of Figure 5.7 for l = `true`.

In the context of executions where the initial value of h is `true` and the initial value of l is `true`, $\mathfrak{R}$ is the result of the analysis of **if** l **then** $x := 1$ **else skip end** in the context $(\sigma_2,\ \rho_2)$ knowing that $\sigma_2(l)$ is `true` and $\rho_2(l)$ is $\bot$. $\mathfrak{R}$ stands for $(\mathfrak{D}_l, \mathfrak{X}_l)$. This analysis result, which is equal to $(\mathcal{I}d[x \mapsto \{l\}],\ \{x\})$ in our example, satisfies Hypotheses 5.2.1 and 5.2.2. Therefore, under the assumption that the static analysis used to compute this result satisfies Hypothesis 5.2.3, the result $(\mathfrak{D}_l, \mathfrak{X}_l)$ can be used by the dynamic analysis to monitor the information flows. For the purpose of this example, $(\mathfrak{D}_l, \mathfrak{X}_l)$ has been computed by manually solving the constraints implied by the *acceptability* rules of Figure 5.5. Using $(\mathfrak{D}_l, \mathfrak{X}_l)$ with the rule (E$_{\mathcal{M}}$-IF$_\top$), it comes out that $\rho_3(x) = \rho_2(x) \sqcup \rho_2(l) \sqcup \top$.

And variable l is `false`? Table 5.2 shows the values of the wild cards used in Figure 5.7 for an execution where the initial value of h is `true`, the initial value of l is `false` and the initial value of x is 3.

Wild Card	Means	Value
ζ_1	$(\sigma_1,\ \rho_1)$	$(\ [h \mapsto \texttt{true},\ l \mapsto \texttt{false},\ x \mapsto 3],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot]\ )$
ζ_2	$(\sigma_2,\ \rho_2)$	$(\ [h \mapsto \texttt{true},\ l \mapsto \texttt{false},\ x \mapsto \texttt{0}],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot]\ )$
$\texttt{P}^\sharp$	**if** l **then** x **:=** 1 **else skip end**	
$\mathfrak{R}$	$(\mathfrak{D}_{\neg l}, \mathfrak{X}_{\neg l})$	$(\mathcal{I}d,\ \emptyset)$
ζ_3	$(\sigma_3,\ \rho_3)$	$(\ [h \mapsto \texttt{true},\ l \mapsto \texttt{false},\ x \mapsto \texttt{0}],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot]\ )$

Table 5.2: Value of the wild cards of Figure 5.7 for $l = \texttt{false}$.

In the context of executions where the initial value of h is `true` and the initial value of l is `false`, $\mathfrak{R}$ is the result of the analysis of **if** l **then** x **:=** 1 **else skip end** in the context $(\sigma_2,\ \rho_2)$ knowing that $\sigma_2(l)$ is `false` and $\rho_2(l)$ is $\bot$. $\mathfrak{R}$ stands for $(\mathfrak{D}_{\neg l}, \mathfrak{X}_{\neg l})$. In our example, this analysis result is equal to $(\mathcal{I}d,\ \emptyset)$. Using $(\mathfrak{D}_{\neg l}, \mathfrak{X}_{\neg l})$ with the rule (E$_{\mathcal{M}}$-IF$_\top$), it comes out that $\rho_3(x) = \rho_2(x) \sqcup \rho_2(x) \sqcup \bot$.

5.3.2 What happens when variable h is `false`?

Figures 5.8 and 5.9 present the evaluation trees for executions where h is `false`. Such executions start by setting the value of x to `0`. Then, as h is `false`, the then-branch (**skip**) is analyzed and the else-branch (**if** l **then** x **:=** 1 **else skip end**) is executed. The analysis of the then-branch yields an analysis result, $(\mathcal{I}d,\ \emptyset)$, emphasizing the fact that there is no information flow created by a potential execution of the then-branch in a context similar to the current execution. Therefore, the tag store after the execution of the conditional branching on the variable h (ρ_4) is equal to the least upper bound of the tag store after the execution of the else-branch (ρ_3) and the tag store before the execution of the conditional (ρ_2): $\forall x.\, \rho_4(x) = \rho_2(x) \sqcup \rho_3(x)$.

And variable l is `true`? Figure 5.8 presents the evaluation tree for executions where h is `false` and l is `true`. Table 5.3 shows the values of the wild cards used in Figure 5.8 for an execution where the initial value of h is `false`, the initial value of l is `true` and the initial value of x is 3.

$$\dfrac{\dfrac{}{\zeta_1,\ \bot\ \vdash\ x \mathrel{\textbf{:=}} \texttt{0}\ \Downarrow_{\mathcal{M}}\ \zeta_2} \qquad \dfrac{\dfrac{\dfrac{}{\zeta_2,\ \top\ \vdash\ x \mathrel{\textbf{:=}} 1\ \Downarrow_{\mathcal{M}}\ \zeta_3}}{\zeta_2,\ \top\ \vdash\ \texttt{P'}\ \Downarrow_{\mathcal{M}}\ \zeta_3} \qquad [\![\zeta_2 \vdash \textbf{skip}]\!]^{\sharp\mathcal{G}} = (\mathcal{I}d, \emptyset)}{\zeta_2,\ \bot\ \vdash\ \textbf{if}\ h\ \textbf{then skip else}\ \texttt{P'}\ \textbf{end}\ \Downarrow_{\mathcal{M}}\ \zeta_4}}{\zeta_1,\ \bot\ \vdash\ x \mathrel{\textbf{:=}} \texttt{0}\ ;\ \textbf{if}\ h\ \textbf{then skip else}\ \texttt{P'}\ \textbf{end}\ \Downarrow_{\mathcal{M}}\ \zeta_4}$$

Figure 5.8: Evaluation tree of the program in Figure 5.6 for $h = \texttt{false} \wedge l = \texttt{true}$.

Wild Card	Means	Value
ζ_1	$(\sigma_1,\ \rho_1)$	$(\ [h \mapsto \texttt{false},\ l \mapsto \texttt{true},\ x \mapsto 3],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot]\ )$
ζ_2	$(\sigma_2,\ \rho_2)$	$(\ [h \mapsto \texttt{false},\ l \mapsto \texttt{true},\ x \mapsto \texttt{0}],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot]\ )$
`P'`	**if** l **then** x **:=** 1 **else skip end**	
ζ_3	$(\sigma_3,\ \rho_3)$	$(\ [h \mapsto \texttt{false},\ l \mapsto \texttt{true},\ x \mapsto 1],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \top]\ )$
ζ_4	$(\sigma_4,\ \rho_4)$	$(\ [h \mapsto \texttt{false},\ l \mapsto \texttt{true},\ x \mapsto 1],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \top]\ )$

Table 5.3: Value of the wild cards of Figure 5.8.

In a context where the initial value of h is `false` and the initial value of l is `true`, the conditional branching on variable l is evaluated with a program counter carrying variety (its tag is $\top$) and yields the assignment "x **:=** 1". This assignment is also evaluated with a program counter tag equal to $\top$. Therefore, after the execution of the assignment, variable x is also considered to carry variety ($\rho_3(x) = \top$).

And variable l is `false`? Figure 5.9 presents the evaluation tree for executions where h is `false` and l is `false`. Table 5.4 shows the values of the wild cards used in Figure 5.9 for an execution where the initial value of h is `false`, the initial value of l is `false` and the initial value of x is 3.

$$\dfrac{\dfrac{}{\zeta_1,\ \bot\ \vdash\ x := 0\ \Downarrow_{\mathcal{M}}\ \zeta_2} \qquad \dfrac{\dfrac{\dfrac{}{\zeta_2,\ \top\ \vdash\ \mathbf{skip}\ \Downarrow_{\mathcal{M}}\ \zeta_3}}{\zeta_2,\ \top\ \vdash\ \mathtt{P'}\ \Downarrow_{\mathcal{M}}\ \zeta_3} \qquad [\![\zeta_2 \vdash \mathbf{skip}]\!]^{\sharp_{\mathcal{G}}} = (\mathcal{I}d, \emptyset)}{\zeta_2,\ \bot\ \vdash\ \mathbf{if}\ h\ \mathbf{then\ skip\ else}\ \mathtt{P'}\ \mathbf{end}\ \Downarrow_{\mathcal{M}}\ \zeta_4}}{\zeta_1,\ \bot\ \vdash\ x := 0\ ;\ \mathbf{if}\ h\ \mathbf{then\ skip\ else}\ \mathtt{P'}\ \mathbf{end}\ \Downarrow_{\mathcal{M}}\ \zeta_4}$$

Figure 5.9: Evaluation tree of the program in Figure 5.6 for h = `false` $\wedge$ l = `false`.

Wild Card	Means	Value
ζ_1	(σ_1, ρ_1)	$([h \mapsto \mathtt{false},\ l \mapsto \mathtt{false},\ x \mapsto 3],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot])$
ζ_2	(σ_2, ρ_2)	$([h \mapsto \mathtt{false},\ l \mapsto \mathtt{false},\ x \mapsto 0],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot])$
`P'`	**if** l **then** x **:=** 1 **else skip end**	
ζ_3	(σ_3, ρ_3)	$([h \mapsto \mathtt{false},\ l \mapsto \mathtt{false},\ x \mapsto 0],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot])$
ζ_4	(σ_4, ρ_4)	$([h \mapsto \mathtt{false},\ l \mapsto \mathtt{false},\ x \mapsto 0],\quad [h \mapsto \top,\ l \mapsto \bot,\ x \mapsto \bot])$

Table 5.4: Value of the wild cards of Figure 5.9.

In a context where the initial value of h is `false` and the initial value of l is `false`, the conditional branching on variable l is evaluated with a program counter carrying variety (its tag is $\top$) and yields the statement "**skip**". The unexecuted branch of this conditional ("x **:=** 1") is not analyzed. The reason is that its test (l) does not carry variety. Therefore, the test of the conditional has the same value for any execution started with the same public inputs. Hence, the assignment contained in the unexecuted branch is never executed by any initially low equivalent executions. Consequently, the monitored execution state after the execution of the conditional branching on l (ζ_3) is equal to the monitored execution state before the execution of this conditional (ζ_2).

5.3.3 Conclusion

Table 5.5 summarizes the possible final values ($\sigma_f(x)$) and tags ($\rho_f(x)$) of the variable x depending on the initial values of the inputs h ($\sigma_i(h)$) and l ($\sigma_i(l)$). As can be seen in this table, if l is `false` then the final value of x is always 0 independently from the initial value of h. Therefore, whenever l is `false`, there is no flow from h to x. This is well detected by the dynamic analysis which sets the final tag of x to $\bot$ whenever l is `false`. On the other side, if l is `true` then the final value of x depends on the initial value of h. This is well detected by the dynamic analysis which sets the final tag of x to $\top$ whenever l is `true`. Additionally,

$\sigma_i(l)$	$\sigma_i(h)$	$\sigma_f(x)$	$\rho_f(x)$
`true`	`true`	0	$\top$
`true`	`false`	1	$\top$
`false`	`true`	0	$\bot$
`false`	`false`	0	$\bot$

Table 5.5: Final values and tags of x depending on the initial values of l and h.

the dynamic analysis behaves securely as it does not create any new covert channel. Indeed, the final tag associated to the variable x never depends on the initial value of the only secret input h.

5.4 Properties of the Dynamic Analysis

Section 5.2 formally defined the dynamic information flow analysis proposed in this chapter. The soundness of this dynamic analysis for the detection of information flows and as the basis of an information flow correction mechanism, in order to enforce the notion of noninterfering execution given in Definition 5.1.1, is proved in Section 5.4.1. Section 5.3 demonstrated on an example that the dynamic analysis proposed in this chapter is sometimes more precise than the dynamic analysis proposed in Chapter 3, and also more precise than the majority of information flow analyses. Section 5.4.2 proves that the dynamic analysis developed in this chapter is always at least as precise as the dynamic information flow analysis of Chapter 3.

5.4.1 Soundness

The dynamic noninterference analysis is proved to be a sound noninterfering execution detection analysis (Theorem 5.4.1). It is also proved to be sound with regard to serving as the basis of an information flow correction mechanism (Theorem 5.4.2). This means that, using a monitoring mechanism constructed from this dynamic noninterference analysis and the correction mechanism proposed on page 82, at the end of any two monitored executions of a given program P started with the same public inputs (variables which do not belong to $\mathcal{S}(\mathrm{P})$), the final values of observable outputs (variables which belong to $\mathcal{O}(\mathrm{P})$) are the same for both executions.

Theorem 5.4.1 proves that the proposed dynamic analysis is sound with regard to information flow detection. Any variable, whose tag at the end of the execution is $\bot$, has the same final value for any execution started with the same public inputs.

Theorem 5.4.1 (Sound Detection).
Assume that the monitoring semantics $\Downarrow_{\mathcal{M}}$ uses a static information flow analysis ($[\![]\!]^{\sharp\mathcal{G}}$) for which Hypotheses 5.2.1, 5.2.2 and 5.2.3 hold. For all programs P, monitoring execution states $(\sigma_1,\ \rho_1)$, $(\sigma_1',\ \rho_1')$, $(\sigma_2,\ \rho_2)$ and $(\sigma_2',\ \rho_2')$ such that:

- $(\sigma_1,\ \rho_1),\ \bot\ \vdash\ P\ \Downarrow_{\mathcal{M}}\ (\sigma_1',\ \rho_1')$,
- $(\sigma_2,\ \rho_2),\ \bot\ \vdash\ P\ \Downarrow_{\mathcal{M}}\ (\sigma_2',\ \rho_2')$,

- $\forall x \notin \mathcal{S}(P).\ \sigma_1(x) = \sigma_2(x)$,
- $\forall x \in \mathcal{S}(P).\ \rho_1(x) = \top$

the following holds:

$$\forall x. \quad (\rho'_1(x) = \bot) \Rightarrow (\sigma'_1(x) = \sigma'_2(x))$$

Proof. Follows directly from Lemma C.2.3.

Variables whose final tags are not $\bot$ may have different final values for two initially low equivalent executions. Therefore, the monitoring mechanism must modify the final value of any variable belonging to $\mathcal{O}(\mathrm{P})$ and whose final tag is not $\bot$. However, as emphasized in previous chapters and by Le Guernic and Jensen (2005), if the correction of "bad" information flows is done without enough care, the correction mechanism itself can become a new covert channel carrying secret information. Theorem 5.4.2 proves that any variable belonging to $\mathcal{O}(\mathrm{P})$ and whose final tag is not $\bot$ can safely be reset to a default value because the final tag of a variable does not depend on the secret inputs of the program.

Theorem 5.4.2 (Sound Basis of Correction).
Assume that the monitoring semantics $\Downarrow_{\mathcal{M}}$ uses a static information flow analysis ($[\![\,]\!]^{\sharp\mathcal{G}}$) for which Hypotheses 5.2.1, 5.2.2 and 5.2.3 hold. For all programs P, monitoring execution states $(\sigma_1,\ \rho_1)$, $(\sigma'_1,\ \rho'_1)$, $(\sigma_2,\ \rho_2)$ and $(\sigma'_2,\ \rho'_2)$ such that:

- $(\sigma_1,\ \rho_1),\ \bot\ \vdash\ P\ \Downarrow_{\mathcal{M}}\ (\sigma'_1,\ \rho'_1)$,
- $(\sigma_2,\ \rho_2),\ \bot\ \vdash\ P\ \Downarrow_{\mathcal{M}}\ (\sigma'_2,\ \rho'_2)$,
- $\forall x \notin \mathcal{S}(P).\ \sigma_1(x) = \sigma_2(x)$,
- $\forall x \in \mathcal{S}(P).\ \rho_1(x) = \top$
- $\rho_1 = \rho_2$

the following holds: $\rho'_1 = \rho'_2$.

Proof. Follows directly from Lemma C.2.5.

5.4.2 Precision

The remainder of this section compares the precision of the dynamic information flow analysis proposed in this chapter with the precision of the monitoring mechanism proposed in Chapter 3. Theorem 5.4.2 proves that a correction mechanism can be based on the tags computed by the dynamic analysis without creating new covert channels. As neither of the dynamic analyses of Chapter 3 and 5 modifies the internal state of the program (the value store), the correction mechanism of Chapter 3 can be used in combination with the dynamic analysis proposed in this chapter. However, as this chapter does not formally specifies a correction mechanism, only the precision of the dynamic information flow analyses of Chapter 3 and 5 are compared.

Theorem 5.4.3 states that, for any program P without output statements, if an execution monitored by the mechanism of Chapter 3 terminates concluding that variables not belonging to V' do not contain secret information, then an execution started with the same inputs and monitored by the dynamic analysis of this chapter does also terminates and concludes that variables not belonging to V' do not contain secret information.

Theorem 5.4.3 (Improved Precision).
Assume that the monitoring semantics $\Downarrow_{\mathcal{M}}$ uses a static information flow analysis ($[\![]\!]^{\sharp_{\mathcal{G}}}$) which is acceptable *and for which Hypotheses 5.2.3 and 5.2.4 hold. For all programs P, whose set of secret inputs is $\mathcal{S}(P)$, value stores σ and σ', monitoring automaton states (V', w'), and tag stores ρ and ρ' such that:*

- $((\mathcal{S}(P), \epsilon), \sigma) \Vdash P \overset{\epsilon}{\Rightarrow} ((V', w'), \sigma')$,
- $\forall x \notin \mathcal{S}(P),\ \rho(x) = \bot$,

the following holds:

- $(\sigma,\ \rho),\ \bot\ \vdash\ P\ \Downarrow_{\mathcal{M}}\ (\sigma',\ \rho')$,
- $\forall x \notin V',\ \rho(x) = \bot$.

Proof. Follows directly from Lemma C.3.5.

5.5 Adding Procedures and Records to the Monitoring Semantics

Previous sections presented a semantics-based dynamic information flow analysis for a core language including assignments, conditionals and loops. Its soundness and level of precision are stated in Section 5.4.2 and proved in Appendix C. The current sections explores some potential extensions to the language supported. Section 5.5.1 proposes an extension to the analysis in order to take into account procedure calls. Section 5.5.2 modifies the notion of values, tags, value stores and tag stores in order to incorporate records. The correction of those extensions have not been formally proved. However, during their development, it has been taken into account that all the lemmas of Appendix C should still hold with those extensions.

5.5.1 Procedures

Figure 5.10(a) describes the syntax of procedure calls. As it is usually done, a procedure call is composed of a procedure name (⟨*procedure-name*⟩) and a list of expressions (⟨*expr-list*⟩). Procedure definitions are not specified. However, the existence of a function, named pbody(), is assumed. This function takes as parameter the name of a procedure, and returns the body of this procedure and three sets which describe the variables used and updated by this procedure.

Let *pn* be the name of a procedure. pbody(*pn*) returns (P, $\bar{g}$, $\bar{p}$, $\bar{l}$) where:

P is the procedure's body. It is the program which is executed whenever *pn* is called.

$\bar{g}$ is a set of global variables. It must at least include all the global variables modified by the procedure. In particular, in order to simulate functions (i.e. procedures returning a value), it should include "*returnedValue*" which would be a reserved variable name. This variable would be used to store the returned value. An assignment to this variable should be the last command of every procedure. An assignment from *returnedValue* to another variable should follow directly any procedure call.

$\bar{p}$ is the list of parameter names of the procedure *pn*.

$\bar{l}$ is a set of local variables. It must at least include all the local variables defined as such by the method, and exclude all the global variables used by the method. Ideally, the intersection of $\bar{p}$ and $\bar{l}$ is empty.

For example, let *incX*(*n*) be a procedure. It increments *n* times the value of the global variable *x* by *n* and returns the new value of *x*. This procedure, whose body follows, uses a local variable named *i*.

```
i := n;
while (i > 0) do
  x := x + n;
  i := i - 1;
done;
returnedValue := x
```

For this procedure, pbody(*incX*) may return (P, {*x*, *returnedValue*}, {*n*}, {*i*}) where P is the program given above which corresponds to the procedure's body.

The semantics of procedure calls is straightforward. It is given in Figure 5.10(b). This semantics uses an "undefined" value written $\flat$ ($\flat \in \mathbb{D}$). It can be assimilated to the "null" value of Java. When a procedure of name *pn* has to be evaluated in a program state $((\sigma, \rho), t^{\mathrm{pc}})$ with the list of parameters $\bar{e}$, the procedure's body is evaluated with a modified monitoring execution state $(\sigma^{\mathrm{i}}, \rho^{\mathrm{i}})$. σ^{i} is equal to the value store σ except for the local variables which are mapped to the undefined value $\flat$ and parameters which are mapped to the value of their respective expressions ($\forall n \in \mathbb{N}.\ \sigma^{\mathrm{i}}(\bar{p}_n) = \sigma(\bar{e}_n)$). ρ^{i} is equal to the tag store ρ except for the local variables which are mapped to the tag $\perp$ and parameters which are mapped to the tag of their respective expressions ($\forall n \in \mathbb{N}.\ \rho^{\mathrm{i}}(\bar{p}_n) = \rho(\bar{e}_n)$). Let $(\sigma^{\mathrm{f}}, \rho^{\mathrm{f}})$ be the monitoring execution state after the evaluation of the procedure's body. The monitoring execution state after the evaluation of the procedure's call is (σ', ρ'). σ' is equal to the value store σ except for the global variables defined by *pn* which are mapped to their respective value in σ^{f} ($\forall n \in \mathbb{N}.\ \sigma'(\bar{g}_n) = \sigma^{\mathrm{f}}(\bar{g}_n)$). ρ' is equal to the tag store ρ except for the global variables which are mapped to their respective tag ρ^{f} ($\forall n \in \mathbb{N}.\ \rho'(\bar{g}_n) = \rho^{\mathrm{f}}(\bar{g}_n)$).

Constraints on the static analysis of procedure calls are given in Figure 5.10(c). When analyzing a procedure call in a given context, the procedure's body is analyzed in a modified context. The modifications to the context reflect the modifications which occur when executing a procedure call as described in Figure 5.10(b). The result returned by the static analysis for the procedure call must be the restriction to the global variables of the result returned by the static analysis for the procedure's body in the modified context.

$$\langle prog \rangle \quad ::= \quad \ldots \quad | \quad \langle procedure\text{-}name \rangle(\langle expr\text{-}list \rangle)$$

(a) Addition to the syntax of the language

$$\frac{\begin{array}{c} \mathrm{pbody}(m) = (\mathtt{P},\ \bar{g},\ \bar{p},\ \bar{l}) \\ \sigma^{\mathrm{i}} = \sigma[\bar{l} \mapsto \flat][\bar{p} \mapsto \sigma(\bar{e})] \qquad \rho^{\mathrm{i}} = \rho[\bar{l} \mapsto \bot][\bar{p} \mapsto \rho(\bar{e})] \\ (\sigma^{\mathrm{i}},\ \rho^{\mathrm{i}}),\ t^{\mathrm{pc}}\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma^{\mathrm{f}},\ \rho^{\mathrm{f}}) \end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ m(\bar{e})\ \Downarrow_{\mathcal{M}}\ (\sigma[\bar{g} \mapsto \sigma^{\mathrm{f}}(\bar{g})], \rho[\bar{g} \mapsto \rho^{\mathrm{f}}(\bar{g})])} \qquad (\mathrm{E}_{\mathcal{M}}\text{-CALL})$$

(b) New semantic rule

$(\mathfrak{D}, \mathfrak{X}) \models (\ (\sigma,\ \rho) \vdash m(\bar{e})\ )$

iff there exist $(\mathfrak{D}_m, \mathfrak{X}_m)$ such that:
$\mathrm{pbody}(m) = (\mathtt{P},\ \bar{g},\ \bar{p},\ \bar{l})$
let $\zeta' = (\sigma[\bar{p} \mapsto \sigma(\bar{e})][\bar{l} \mapsto \flat], \rho[\bar{p} \mapsto \rho(\bar{e})][\bar{l} \mapsto \bot])$ in $(\mathfrak{D}_m, \mathfrak{X}_m) \models (\zeta' \vdash \mathtt{P})$
$\mathfrak{D} = \mathcal{I}d[\bar{g} \mapsto \mathfrak{D}_m(\bar{g})] \quad \wedge \quad \mathfrak{X} = \mathfrak{X}_m \cap \bar{g}$

(c) New constraints on the analysis

Figure 5.10: Addition of procedures to the language for $\Downarrow_{\mathcal{M}}$

It is anticipated, but not formally proved yet, that with the addition of this constraint on the static analysis Theorems 5.4.1 and 5.4.2 are preserved.

5.5.2 Records

This section proposes an extension to the dynamic information flow analysis of Chapter 5 in order to take into account values of type *record*. A record is a value as are integers or booleans. A record is a set of pairs composed of a unique *field* name and a value. A value store, as defined in Chapter 5 and previous chapters, is a mapping from variable names to values. A record can be seen as a value store mapping *field* names to values. Extending previous work of Chapter 5 to incorporate records requires only to redefine the notions of values, tags, value stores and tag stores. The monitoring semantic rules, acceptability rules, hypotheses, and theorems remain the same.

Records allow the representation of any complex acyclic structure. For example a simply linked list can be represented using records having two fields (*head* and *tail*). Figure 5.11 shows the graphical representation of such a record for the list $[1, 2]$. This "list" is a record which maps the field *head* to 1 and the field *tail* to another record. This second record maps *head* to 2 and *tail* to an empty record.

Introduction of records into the default semantics is done by redefining values and value stores. $\mathbb{X}$ is the domain of identifiers. In previous work, $\mathbb{X}$ is the set of all variable names. Let $\mathbb{F}$ be the set of all field names. Let $\mathbb{X}^{\mathrm{var}}$ be the set of all variable names (previously $\mathbb{X}$). The domain of identifiers ($\mathbb{X}$) is redefined

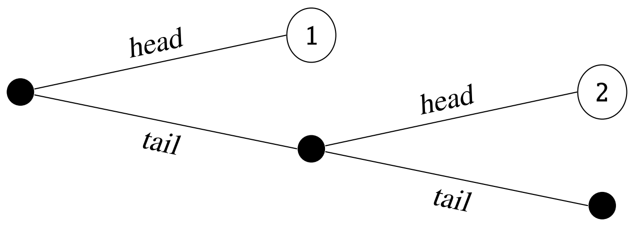


Figure 5.11: Record $\{head:1,\ tail:\{head:2,\ tail:\{\}\}\}$ (the list $[1,2]$)

recursively as follows:

$$\mathbb{X} \ = \ \mathbb{X}^{\text{var}} \ \cup \ \{id.f \mid id \in \mathbb{X} \wedge f \in \mathbb{F}\}$$

$\mathbb{D}$ is the semantic domain of values. Let $\mathbb{D}^{\text{core}}$ be the semantic domain of values as defined in previous work (integers, booleans, ...). The semantic domain of values ($\mathbb{D}$) is redefined recursively as follows:

$$\mathbb{D} \ = \ \mathbb{D}^{\text{core}} \ \cup \ (\mathbb{F} \to (\mathbb{D} \cup \{\flat\}))$$

A value is either a "core value" (integers, booleans, ...) or a total function mapping every field name to a value or the "undefined" value ($\flat$). The record of Figure 5.11 is then a function mapping the field *head* to the integer 1, the field *tail* to another function, and any other field to $\flat$.

A value store is a total function from variable names ($\mathbb{X}^{\text{var}}$) to values ($\mathbb{D}$) ($\sigma : \mathbb{X}^{\text{var}} \to \mathbb{D}$). To simplify access to values contained in variables, the definition of value store is extended to $\mathbb{X}$ as follows:

$$\sigma(id.f) \ = \ (\sigma(id))(f)$$

Therefore, if the variable x in the value store σ is mapped to the list of Figure 5.11 then $\sigma(x.tail.head)$ is 2. Additionally, to simplify field update, value store update is extended as follows:

$$\sigma[id.f \mapsto v] \ = \ \sigma[id \mapsto \sigma(id)[f \mapsto v]]$$

To introduce records into the monitoring semantics it remains to redefine tags and tag stores. $\mathbb{T}$ is the semantic domain of tags. Let $\mathbb{T}^{\text{core}}$ be the set of tags as defined previously ($\mathbb{T}$ in previous work). A tag is now a pair composed of an element of $\mathbb{T}^{\text{core}}$ and a record mapping field names to tags. The semantic domain of tags ($\mathbb{T}$) is recursively defined as follows:

$$\mathbb{T} \ = \ \mathbb{T}^{\text{core}} \ \times \ (\mathbb{F} \to (\mathbb{T} \cup \{\flat\}))$$

The graphical representation of a potential tag for the list $[1,2]$ is given in Figure 5.12. Suppose this list is stored in the variable x, $\sigma(x)$ is represented in Figure 5.11 and $\rho(x)$ is represented in Figure 5.12. From this tag, it is possible to deduce that, for every initially low-equivalent executions, $\sigma(x)$ is a list of at least one element and whose first element is 1.

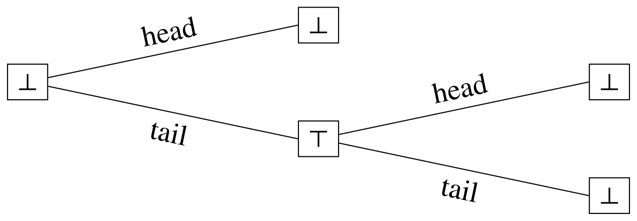


Figure 5.12: A potential tag for the record of Figure 5.11

A tag store is a total function from variable names ($\mathbb{X}^{\text{var}}$) to tags ($\mathbb{T}$) ($\rho : \mathbb{X}^{\text{var}} \to \mathbb{T}$). In the semantic rules and acceptability rules, access to the tag of a given identifier with field names is defined as follows:

$$\begin{array}{lll} \rho(id.f) & \text{stands for} & \text{let } (t^{id},\ r^{id}) = \rho(id) \text{ in} \\ & & \text{let } (t^{f},\ r^{f}) = r^{id}(f) \text{ in} \\ & & (t^{id} \sqcup t^{f},\ r^{f}) \end{array}$$

Similarly to value stores, update of a tag store for an identifier including field names is defined as follows:

$$\rho[id.f \mapsto t] \quad \text{stands for} \quad \rho[id \mapsto \rho(id)[f \mapsto t]]$$

The least upper bound operator is also redefined to operate on the new version of tags ($\sqcup : (\mathbb{T} \cup \{\flat\})^2 \to \mathbb{T}$). Its definition follows:

$$t_1 \sqcup t_2 \quad = \quad \begin{cases} t_2 & \text{iff} \quad t_1 = \flat \\ t_1 & \text{iff} \quad t_2 = \flat \\ (t_1^{\text{c}} \sqcup t_2^{\text{c}},\ \lambda f.\, r_1(f) \sqcup r_2(f)) & \text{iff} \quad t_1 = (t_1^{\text{c}},\ r_1) \ \wedge\ t_2 = (t_2^{\text{c}},\ r_2) \end{cases}$$

Comparison of tags is done exclusively on tags belonging to $\mathbb{T}^{\text{core}}$. A function, called flatten(), is introduced to convert any type of "tag" to a tag belonging to $\mathbb{T}^{\text{core}}$ (flatten() : ($\mathbb{T}^{\text{core}} \cup \{\flat\} \cup \mathbb{T}) \to \mathbb{T}^{\text{core}}$). Tag comparisons ($\sqsubseteq$, =, . . .) implicitly apply flatten() on both of their arguments. Definition of flatten() follows:

$$\text{flatten}(t) \quad = \quad \begin{cases} t & \text{iff} \quad t \in \mathbb{T}^{\text{core}} \\ \bot & \text{iff} \quad t = \flat \\ t^{\text{c}} \sqcup \bigsqcup_{f \in \mathbb{F}} \text{flatten}(r(f)) & \text{iff} \quad t = (t^{\text{c}},\ r) \end{cases}$$

It is anticipated, but not formally proved, that with those definitions Theorems 5.4.1 and 5.4.2 are preserved.

5.6 Conclusion

This chapter, as the previous ones, addresses the problem of monitoring executions in order to enforce the confidentiality of secret data. However, it focuses on the detection of information flows. No correction mechanism is formally defined; even if a suitable one is described on page 82. The confidentiality property to monitor is expressed using a notion of noninterference between secret inputs of the execution and its public outputs (Definition 5.1.1). The language taken into consideration is a sequential language with assignments and conditionals (including loops) as in Chapter 3. The main difference between the dynamic information flow analysis proposed in this chapter and the ones of previous chapters lies in the static analysis used to detect implicit indirect flows. The static information flow analyses used by the dynamic analysis in this chapter are context-sensitive, which is not the case in previous chapters. This allows to increase the precision of the overall dynamic information flow analysis for the detection of implicit and explicit indirect flows.

Another distinguishing feature of the dynamic information flow analysis developed in this chapter lies in the definition of the static analysis used for the detection of implicit indirect flows. No static information flow analysis is formally defined to be used by the dynamic analysis, even if such an analysis can easily be extracted from the acceptability rules of Figure 5.5 by fix point computation. Instead, three hypotheses are defined. It has been proved that any static information flow analysis respecting those hypotheses can be used by the dynamic noninterference analysis proposed in this chapter. The first hypothesis (Hypothesis 5.2.1) simply requires the static analysis to be sound with regard to the detection of the variables whose value may be modified by the execution of a given piece of code in a given context. The second hypothesis (Hypothesis 5.2.2) requires the static analysis to have a precision similar to the overall dynamic analysis for pieces of code executed in a secret context. Finally, Hypothesis 5.2.3 requires the static analysis to be deterministic with regard to public data given in the analysis context. Hypotheses 5.2.2 and 5.2.3 are not required for the overall dynamic analysis to be sound with regard to the detection of information flows. However, they are required in order to prevent a "bad flows" correction mechanism from creating new covert channels which may leak secret information.

In Section 5.4, the proposed noninterference analysis is proved to be sound both with regard to the detection of information flows and with regard to its ability to trigger the correction mechanism without creating a new covert channel. In the same section, Theorem 5.4.3 states that the noninterference analysis proposed in this chapter is more precise than the analysis of Chapter 3 and, by transitivity, more precise than the type system developed by Volpano et al. (1996).

Chapter 6

Conclusion

6.1 Epitome

This PhD thesis dissertation defends the opinion that dynamic confidentiality analyses are worth developing even considering the achievements of static confidentiality analyses. It asserts that noninterference monitors can be formally proved to *ensure* noninterference, deal with complex programming language features, and be more accurate than static analyses in some cases due to a more precise knowledge of the execution context.

In order to support those claims, this PhD thesis dissertation presented three different monitors for ensuring the confidentiality of secret data manipulated by sequential or concurrent programs. The property of confidentiality enforced is defined using the notion of *noninterference* introduced by Goguen and Meseguer (1982). The goal of those monitors is to prevent an attacker from deducing information about the secret inputs of an execution by looking at the public outputs of this execution. The attacker is considered to have access to the source code of the executed program, to the public input values and to the public output values.

All three monitors relies on a dynamic information flow analysis to *track secret information* and on a correction mechanism to *prevent secret information leakage*. In each monitor, the dynamic information flow analysis relies on a static analysis to detect flows created by unexecuted pieces of code. The correction mechanisms impact only the external view of the behavior of the program. The monitors *do not modify the internal state* of sequential programs monitored. For concurrent programs, the monitor can create new deadlocks when the original program synchronizes under the influence of secret information.

Chapter 3 proposed an automaton-based noninterference monitor. The monitor is described under the form of a big-step semantics (the monitoring semantics). Before executing a command, this semantics communicates with a security automaton. The automaton is in charge of tracking secret information and prevent secret information leakage. It keeps track of which variables, or other program state entities, contain secret information (or, as expressed in information theory (Ashby, 1956; Cohen, 1977) and in preceding chapters, "carry variety"). To prevent secret information leakage, this security automaton sends back an admonition to the monitoring semantics to control the execution of the program monitored. The security automaton can either allow the execution of a command, deny the execution of a command, or replace the command to be executed by another one.

The monitor proposed in Chapter 3 is proved to be sound with regard to the notion of noninterference (Theorem 3.4.1). It is also proved to be more precise than a type system for noninterference similar to the one proposed by Volpano et al. (1996) (Theorem 3.4.2). The precision relationships between the different dynamic information flow analyses proposed in this document and some type systems are summarized in Figure 6.1.

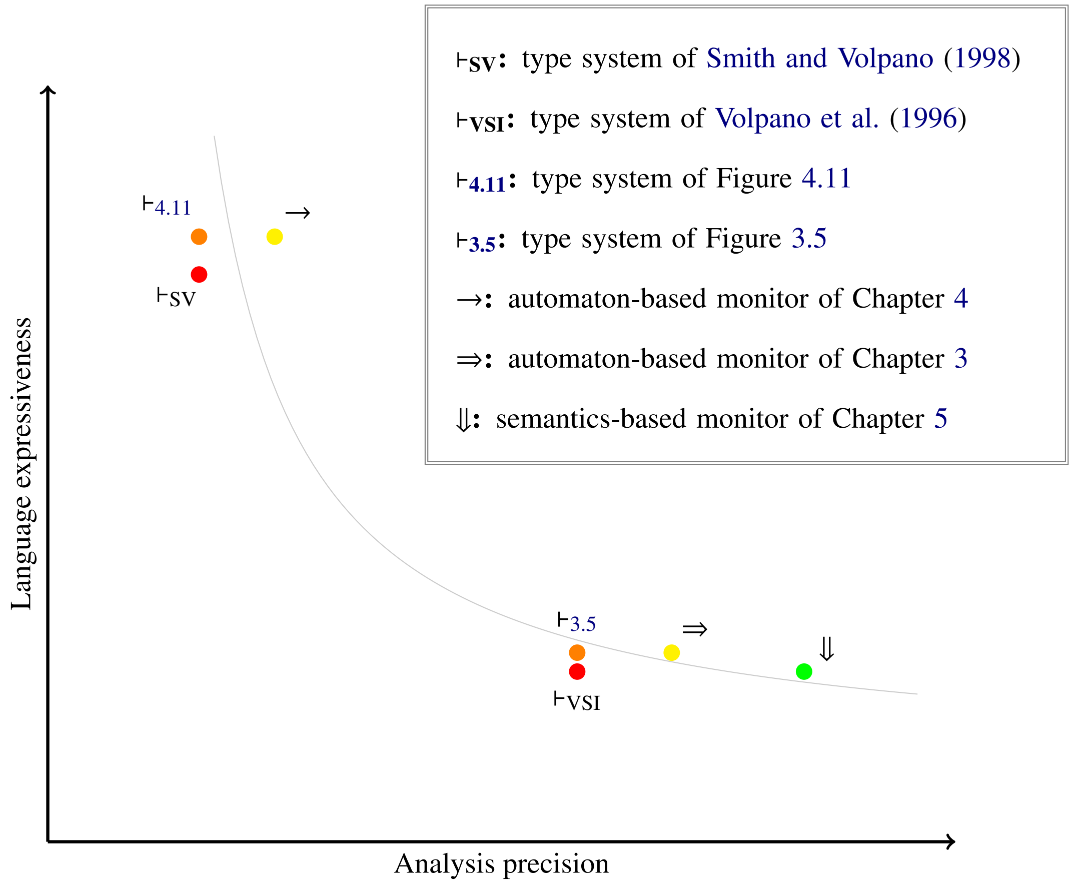


Figure 6.1: Precision of different information flow analyses

The noninterference monitor presented in Chapter 4 deals with concurrent programs. As in Chapter 3, the monitor is described as a semantics that communicates with a security automaton. As in the sequential case, the automaton uses the semantics messages to track the program state entities containing secret information. In order to prevent secret information leakage, the security automaton sends back directives to the semantics to control the execution.

Compared to Chapter 3, this monitor is able to enforce noninterference for multi-threaded programs including synchronization commands. The type of semantics used to describe the monitor is also different.

In Chapter 4, the monitor is described as a small-step semantics. This allows the confidentiality property enforced by the monitor to take into consideration nonterminating programs.

Once again, the proposed noninterference monitor for concurrent programs is proved to be sound with regard to the notion of noninterference (Theorem 4.5.1). It is also proved to be more precise for the detection of information flows than a type system similar to the one proposed by Smith and Volpano (1998) (Theorem 4.5.2).

The monitor's dynamic analysis of Chapter 5 relies on context-sensitive static analyses. This dynamic noninterference analysis is described using a big-step semantics. Unlike analyses of Chapters 3 and 4, application of the semantic rules is not controlled by an external security automaton. The dynamic information flow analysis, which is in charge of tracking secret information, is directly integrated into the monitoring semantics. However, detection of information flows created by unexecuted pieces of code is still supported by a static analysis.

This static analysis is not described, as it was the case for the monitors of Chapters 3 and 4. Instead, three hypotheses are formally defined (Hypotheses 5.2.1, 5.2.2 and 5.2.3). Any static analysis respecting those hypotheses can be "safely" used in combination with the dynamic information flow analysis for the detection of information flows at run-time. Those static analyses are context-sensitive. The result they return depends on the program state at the time the execution detects a piece of code which will not be executed. Roughly, the context sensitivity consists in the detection of part of code, in the unexecuted code `C`, which would not be executed (dead code) due to the program state at the time `C` would have been executed if some secret input values were different.

This dynamic noninterference analysis is proved to be sound both with regard to the detection of information flows and with regard to its ability to trigger the correction mechanism without creating a new covert channel (Theorems 5.4.1 and 5.4.2). Additionally, it is proved to be more precise than the analysis of Chapter 3 for the detection of information flows (Theorem 5.4.3). This last analysis being itself more precise than the type system of Volpano, Smith, and Irvine (1996).

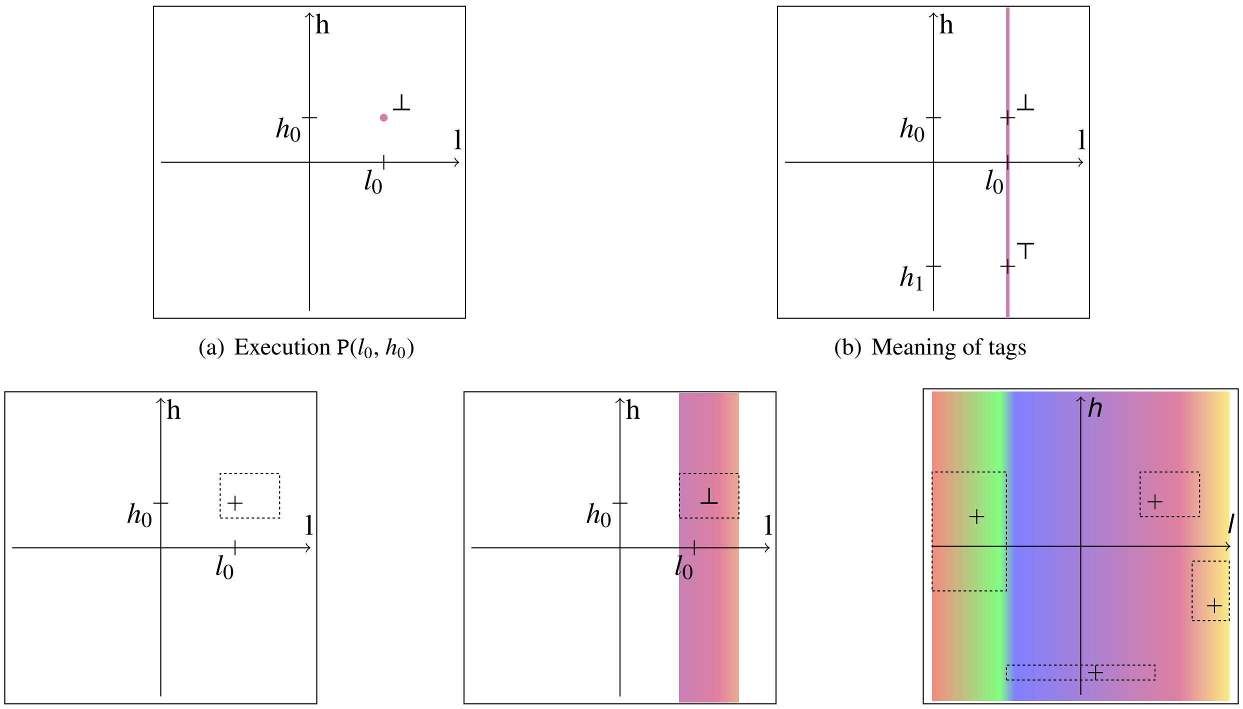
6.2 From Monitoring to Testing

As indicated above, the last noninterference monitor, presented in Chapter 5, relies on the combination of a *defined* dynamic information flow analysis and a *specified* static analysis. This static analysis can be any such analysis which respects Hypotheses 5.2.1, 5.2.2 and 5.2.3. Hypotheses 5.2.2 and 5.2.3 are required solely to ensure that the correction mechanism for "bad" flows does not create any new covert channel. Therefore, any static information flow which respects Hypothesis 5.2.1 (basically, any *sound* static information flow analysis) can be used in combination with the semantic rules of Figure 5.4 to obtain a *sound* dynamic analysis for the *detection* of information flows from private inputs to public outputs. Because a dynamic analysis for the *sole detection* of information flows is less constrained than a dynamic analysis for the *detection and correction* of information flows, other improvements of the precision of the dynamic analysis are possible. For example, with sequential programs, explicit indirect flows can be taken into account only at the end of the execution of the conditional creating them.

With regard to noninterference, a dynamic analysis suited only for the detection of information flows, and not their correction, can be used only for noninterference testing. The idea behind noninterference testing is to run enough executions of a program in order to cover a "high enough" subset of all possible executions of the program. In cases where the dynamic analysis results enable to conclude that all the executions covered are safe, users gain a confidence in the "safe" behavior of the program which is *proportional to the coverage level*. When dealing with the confidentiality of secret data, an incomplete coverage does not seem acceptable. The aim of noninterference testing is then to cover all possible executions. It is not possible to run an execution for every possible input set (as there are frequently infinitely many input values). However, the tags associated to variables — resulting from the dynamic analysis — are the same for many executions with different inputs. Therefore, it may be possible to conclude about the noninterference behavior of any execution of a program by testing a limited, hopefully finite, number of executions.

A limited number of executions may be sufficient to conclude that a program is noninterfering. The main idea behind noninterference testing has been exposed above. Figure 6.2 sketches this idea. Let P be a program whose secret inputs are represented by h, public inputs by l, and public outputs by a color. P(l_i, h_j), where (l_i, h_j) are input values, is the public output, represented by a color, of the execution of P with the inputs (l_i, h_j). In the representations of Figure 6.2, public input values are represented on the x-axis and secret input values are represented on the y-axis. Each point of the different graphs corresponds to the execution of P with, as inputs, the coordinates of this point. Whenever a point in the graph is colored, the color corresponds to the public output value of the execution of P with, as inputs, the coordinates of the colored point. Figure 6.2(a) represents the execution of P with inputs (l_0, h_0). Its public output value is represented by the color ● and its tag — result of the dynamic information flow analysis — is $\perp$ (the public output does not carry variety). Figure 6.2(b) shows the meaning of this tag. As the public output tag of P(l_0, h_0) is $\perp$, it means that for any secret inputs h_j the public output value of P(l_0, h_j) is the same as for P(l_0, h_0); it is ●. Even if there exist secret inputs h_1 for which the public output tag of P(l_0, h_1) is $\top$, any execution of P with public inputs l_0 is noninterfering (it corresponds to Theorem 5.4.1 for the analysis of Chapter 5). It only means that the dynamic analysis is not precise enough to directly detect that the execution of P with inputs (l_0, h_1) is noninterfering. However, the fact that P(l_0, h_1) is noninterfering can be indirectly deduced from the result of the dynamic analysis of the execution of P with inputs (l_0, h_0).

The main challenge of noninterference testing is to develop a dynamic analysis for which it is possible to characterize a set of executions which associate the same tag to the public output as an execution which as already been tested. For example, assume that it has been proved that all executions in the dashed area in Figure 6.2(c) associate the same tag to the public output as the execution of P with inputs (l_0, h_0). As this tag is $\perp$, it is possible to conclude from the single execution of P with inputs (l_0, h_0) that all colored executions in Figure 6.2(d) are noninterfering. Therefore, with only a limited number of executions, as in Figure 6.2(e), it is possible to deduce that the program is noninterfering for a wide range of inputs which can be characterized.



(a) Execution $\mathtt{P}(l_0, h_0)$ (b) Meaning of tags (c) Public output tag's scope (d) Meaning of a ⊥ area (e) Noninterfering executions

Figure 6.2: Sketch of the main idea of noninterference testing.

Developing a dynamic information flow analysis for noninterference testing. As exposed above, in order to be able to conclude on the interference behavior of a program by testing it, it is necessary to be able to characterize a finite number of executions which are sufficient to conclude about all executions of this program. It is then necessary to develop a dynamic analysis which has the right balance between the number of executions covered by one test and the precision of the analysis.

The solution approached here assumes there is no recursive method calls and is based on "acyclic Control Flow Graphs" (aCFG). As its name suggests an aCFG is a Control Flow Graph (CFG) without cycles. In an aCFG, there is no edge from the node corresponding to the body of a loop statement to the node corresponding to the test of this loop statement. Instead, there is an edge from the body of the loop to the block following the loop statement. Figure 6.4(a) shows the standard CFG of the code in Figure 6.3, and Figure 6.4(b) shows its aCFG. In an acyclic CFG, there is a finite number of paths. The maximum number of paths is equal to 2^b, where b is the number of branching statements (`if` and `while` statements) in the program.

The approach to noninterference testing proposed in this section is based on a dynamic information flow analysis which returns the same result for any execution that follows the same path in the aCFG of the program analyzed. Let the acyclic CFG trace of an execution be the list of nodes of the aCFG encountered during the execution. Let $\tau[\![\sigma \vdash \mathtt{P}]\!]$ be the acyclic CFG trace of the execution of program $\mathtt{P}$ with initial value store σ. Let $\mathbb{T}[\![\zeta \vdash \mathtt{P}]\!]$ be the result of the dynamic information flow analysis of the execution of

P in the initial program state ζ. The constraint imposed on the dynamic information flow analysis for the noninterference testing approach proposed is formalized in Hypothesis 6.2.1.

Hypothesis 6.2.1 (Hypothesis on the dynamic analysis for noninterference testing).
For all programs P, value stores σ_1 and σ_2, and tag stores ρ:

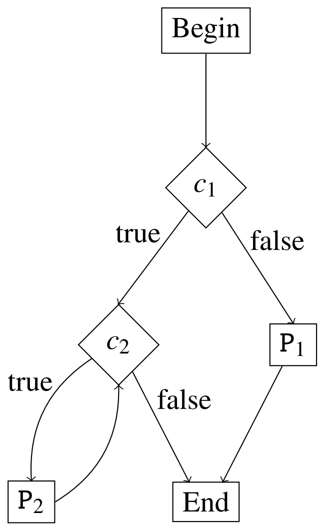
$$\tau[\![\sigma_1 \vdash P]\!] = \tau[\![\sigma_2 \vdash P]\!] \quad \Rightarrow \quad \mathbb{T}[\![(\sigma_1, \rho) \vdash P]\!] = \mathbb{T}[\![(\sigma_2, \rho) \vdash P]\!]$$

With such a dynamic analysis, the problem of verifying the noninterference behavior of any execution is reduced to the well known testing problem of achieving 100% feasible paths coverage (Beizer, 1990; Godefroid et al., 2005; Ntafos, 1988; Sen and Agha, 2006; Williams et al., 2005).

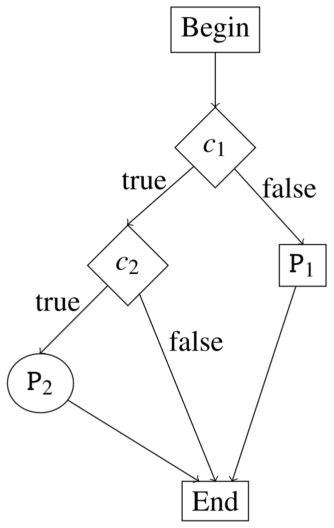
Noninterference testing seems as difficult as promising. Noninterference monitoring has one main advantage and one main drawback compared to static noninterference analyses. Dynamic noninterference analyses have a more precise knowledge of the control flow of the program than static analyses, which increases their precision. Of course, some parts of the code must still be statically analyzed even with a dynamic analysis. However, as shown in Chapter 5, the result of static analyses of unexecuted code in a dynamic analysis can be more precise than when analyzed with fully static analyses. The reason is that,

```
if c₁ then
  while c₂ do P₁ done
else
  P₂
end
```

Figure 6.3: Listing of an example code



(a) Standard CFG



(b) Acyclic CFG

Figure 6.4: CFG and aCFG of the code in Figure 6.3

with a dynamic analysis, pieces of code corresponding to unexecuted code are analyzed in a smaller context than with a fully static information flow analysis. This feature also increases the precision of dynamic information flow analyses. However, as stated previously in this document and by Le Guernic and Jensen (2005), to ensure noninterference a monitor must prevent its information flow correction mechanism from creating a new covert channel. This characteristic imposes a special constraint on the dynamic information flow analysis which is specific to noninterference monitors. In Chapter 5, this constraint is characterized by Hypothesis 5.2.2. This constraint implies that the dynamic analyses used by noninterference monitors are less precise than dynamic analyses developed only for the *detection* of information flows. The dynamic information flow analysis used for noninterference testing, except for the coverage problem, is only required to detect information flows. Therefore, a dynamic information flow analysis for noninterference testing can enjoy the main advantage of noninterference monitoring over static analyses without suffering from its main drawback.

It is easy to develop a dynamic information flow analysis for noninterference testing which gives sound results with regard to the noninterference behavior of tested executions. However, developing a noninterference testing methodology which give sound results with regard to the noninterference behavior of a program (i.e. with regard to the noninterference behavior of all its executions) is way harder. With the methodology proposed above, it requires achieving and certifying 100% feasible paths coverage in the aCFG (i.e. testing at least one execution for every path which can be followed by an execution of the program). Another approach may be to characterize executions which have been covered by the test and guard executions of the tested program to allow only those executions which have been covered. With a dynamic analysis respecting the Hypothesis 6.2.1, this could be achieved by running a symbolic execution in parallel with every test execution in order to collect the test values of the conditionals encountered expressed as functions of the input values.

Noninterference testing may be an interesting field of study having its own specific problems. It may never be as complete as static noninterference analyses or noninterference monitors. However, it is more precise, or at least as precise, than the static analyses or monitors on specific executions. Thus noninterference testing may allow to validate some specific programs whose validation is out of reach of the two other approaches, or at least help find information flow bugs.

6.3 Synthesis

In the last quarter of century, research on noninterference — a confidentiality property — has focused on static analyses. Early work on dynamic noninterference analysis, dating essentially from the seventies, have not been pushed further during two decades. This dissertation defend the thesis that dynamic noninterference analysis is a promising approach, even with the high level of development already achieved in static noninterference analysis. This point of view seems to be shared by part of the scientific community with a revival of dynamic information flow analyses in the last couple of years (McCamant and Ernst, 2007; Shroff et al., 2007; Vachharajani et al., 2004; Venkatakrishnan et al., 2006).

Contrary to a widely spread assumption, noninterference is a property within reach of dynamic analyses. McLean (1994) and Schneider (2000) proved that strong information flow properties, as noninterference, are not enforceable by *execution monitors*. Such monitors have only access to the sequence of actions which are really executed. However, in this thesis and other works (Masri et al., 2004; Shroff et al., 2007; Vachharajani et al., 2004), it is shown that a dynamic analysis having access to unexecuted code can effectively track all types of information flows.

Yet, a static noninterference analysis is only concerned with the *detection* of information flows. This is not the case for a dynamic noninterference analysis. It is acceptable for a static analysis to *warn a user before execution* that a program is or is not noninterfering. On the contrary, it is not acceptable for a dynamic noninterference analysis to *simply warn the user at run time* that some secret information have been spread in the public domain. A user expect a dynamic noninterference analysis to be coupled with a correction mechanism which prevents any secret information leakage based on the results of the dynamic analysis. Previous works on dynamic noninterference analysis (Masri et al., 2004; Vachharajani et al., 2004; Venkatakrishnan et al., 2006) lack formal proofs that the proposed monitor — the combination of a dynamic information flow analysis and a correction mechanism — actually enforces noninterference. The present thesis dissertation formally proved that it is possible to develop noninterference monitors that enforce noninterference (Chapter 3), even for languages including complex features as concurrency and synchronization (Chapter 4).

Dynamic analyses have some intrinsic advantages over static analyses. First of all, a monitor analyzes a program at run-time. A monitor is then inherently more user-centric than a static analysis. It is easier for a dynamic analysis to take into consideration the precise policy desired by a user for a precise execution than it is for a static analysis. Additionally, a noninterference monitor allows a user to safely use the noninterfering executions of a program while still preventing interfering ones to leak secret information (at the cost of the modification of the semantics of those interfering executions). This is more difficult with a static analysis. Such analyses return a sole conclusion which must be valid for all executions.

Moreover, due to their dynamic nature, noninterference monitors have a better knowledge of the control flow of the program executed. This allows a dynamic information flow analysis to be sometimes more precise than a static information flow analysis. As an example, let us consider the following program where h is the only secret input and l the only public input.

```
if ( f(l) ) then t := h else skip end;
if ( g(l) ) then x := t else skip end;
output x
```

Without information on f and g (and often, even with), a static analysis would conclude that this program is unsafe because the secret input information could be carried to x through t and then to the output. However, if f and g are such that no value of l makes both predicates true, then any execution of the program is perfectly safe. In that case, the monitor would allow any execution of this program. The reason is that, l being a public input, only executions following the same path as the current execution are taken care of by the monitoring mechanism. So, for such configurations where the branching conditions are not influenced by the secret inputs, a monitoring mechanism is at least as precise as any static analysis — and often more precise (Theorem 3.4.2).

Finally, as shown in Chapter 5, a dynamic information flow analysis can gain precision over a static analysis even for the detection of information flows created by unexecuted pieces of code. This is achieved by taking into consideration some knowledge about the program state in which the unexecuted code would have been executed if the secret inputs were different. For example, consider the following program where *h* is the only secret input and *l* the only public input.

```
x := 0; t := f(l);
if h then
  skip
else
  if t then x := 1 else skip end
end;
output x
```

A static analysis would accurately conclude that this program is probably interfering — it is the case if there exists a value for *l* such that `f(l)` is `true`. Depending on the complexity of the procedure *f*, it would be difficult to characterize statically which executions are interfering and which are not. The dynamic information flow analysis proposed in Chapter 5 is able to accurately detect if an execution of this program is interfering or not. Therefore, it allows to safely run any noninterfering execution without altering their semantics while still preventing any secret information leakage for any execution of this program.

Static *and* dynamic noninterference analyses are worth research investment and works well in combination. Executions of a program statically verified do not incur a run-time overhead, while final users have a strong guarantee that the program respects the confidentiality of their secret. However, a final user may not be able to run a static analysis on a program, or may not be willing to do it for a program he may just use once a year (a tax application for example). Therefore, a noninterference monitor is useful when the final user does not trust the source of a program or when a program could not pass a static analysis. The improved precision of a dynamic information flow analysis on precise executions may allow use of the program, or at least of the noninterfering executions, while still preserving the confidentiality of secret information.

Bibliography

Abadi, Martín (1996) and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996.

Abadi, Martín (2003) and Cédric Fournet. Access control based on execution history. In *Proc. of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121. ISOC, February 2003. ISBN Number 1-891562-16-9.

Abadi, Martín (1996), Butler W. Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 83–91. ACM Press, 1996. ISBN: 0-89791-770-7. doi: http://doi.acm.org/10.1145/232627.232638.

Abadi, Martín (1999), Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.

Agat, J. (2000). Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

Aiken, Alexander (1999). Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, November 1999. ISSN: 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(99)00007-6. http://theory.stanford.edu/~aiken/publications/papers/scp99.ps.

Alpern, Bowen (1985) and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4): 181–185, October 1985. doi: 10.1016/0020-0190(85)90056-0. http://www.sciencedirect.com/science/article/B6V0F-482R9HJ-1C/2/d0affa64a6086efa1a8c7fabbdcf9227.

Ashby, William Ross (1956). *An Introduction to Cybernetics*. Chapman & Hall, London, 1956. ISBN 0416683002.

Banâtre, J.-P. (1994), C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *Proc. European Symp. on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 55–73. Springer-Verlag, 1994.

Banerjee, Anindya (2002) and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, June 2002.

Banerjee, Anindya (2003) and David A. Naumann. Using access control for secure information flow in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 155–169. IEEE Computer Society, June 2003.

Banerjee, Anindya (2004) and David A. Naumann. History-based access control and secure information flow. In *Proc. of the workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS)*, May 2004. See http://www.cis.ksu.edu/~ab/bib.html to have the latest bibtex infos.

Banerjee, Anindya (2005) and David A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

Barendregt, Henk P. (1984). *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, October 1984.

Barthe, Gilles (2005) and Tamara Rezk. Non-interference for a jvm-like language. In ACM Press, editor, *Proceedings of the international workshop on Types in Languages Design and Implementation*, pages 103–112, New York, NY, USA, 2005. ISBN: 1-58113-999-3. doi: 10.1145/1040294.1040304.

Barthe, Gilles (1999) and B. Serpette. Partial evaluation and non-interference for object calculi. In *Proc. FLOPS*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67. Springer-Verlag, November 1999.

Barthe, Gilles (2004), Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 100–114, Washington, DC, USA, June 2004. IEEE Computer Society Press. ISBN: 0-7695-2169-X. doi: http://dx.doi.org/10.1109/CSFW.2004.17.

Barthe, Gilles (2006), Tamara Rezk, and David A. Naumann. Deriving an information flow checker and certifying compiler for java. In *Proceedings of Symposium on Security and Privacy*, pages 230–242, Washington, DC, USA, may 2006. IEEE Computer Society. ISBN: 0-7695-2574-1. doi: 10.1109/SP.2006.13.

Barthe, Gilles (2007), David Pichardie, and Tamara Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proceedings of the European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. to appear.

Beizer, Boris (1990). *Software Testing Techniques*. International Thomson Computer Press, 2nd edition edition, June 1990. ISBN: 1-85032-880-3.

Bell, David Elliott (1973) and Leonard J. (aka Len) LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, March 1973a.

Bell, David Elliott (1973) and Leonard J. (aka Len) LaPadula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, May 1973b. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

Birznieks, Gunther (1998). Cgi/perl taint mode faq, June 1998. http://gunther.web66.com/FAQS/taintmode.html.

Blizard, Wayne D. (1988). Multiset theory. *Notre Dame Journal of Formal Logic*, 30(1):36–66, 1988. ISSN: 0029-4527. doi: 10.1305/ndjfl/1093634995. http://projecteuclid.org/euclid.ndjfl/1093634995.

Bogor. Bogor: Software model checking framework, March 2007. http://bogor.projects.cis.ksu.edu/.

Boudol, Gérard (2001) and Ilaria Castellani. Noninterference for concurrent programs. In *Proc. ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 382–395, July 2001.

Boudol, Gérard (2002) and Ilaria Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

BRAND, Sheila L. (1985). Department of Defense Trusted Computer System Evaluation Criteria. National Computer Security Center, Fort Meade, Maryland, USA, December 1985.

BREWER, D. F. C. (1989) and M. J. NASH. The chinese wall security policy. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 206–214, May 1989.

Per Brinch Hansen, editor. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag, 2002.

BROOKES, Stephen D. (2004). A semantics for concurrent separation logic. In Gardner and Yoshida (2004), pages 16–34. ISBN: 3-540-22940-X.

BROWN, Jeremy (2001) and Thomas F. KNIGHT, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical Report ARIES-TM-015, MIT, November 2001.

CARDELLI, Luca (1985) and Peter WEGNER. On understanding types, data abstraction, and polymorphism. In Anthony I. Wasserman, editor, *The MIT Press scientific computation series*, volume 17 of *ACM Computing Surveys*, pages 471–522. ACM Press, New York, NY, USA, December 1985. doi: http://doi.acm.org/10.1145/6041.6042.

CLARK, David (2002), Chris HANKIN, and Sebastian HUNT. Information Flow for ALGOL-like Languages. In A. Cortesi and R. Focardi, editors, *Computer Languages and Security*, volume 28 of *Computer Languages, Systems & Structures*, pages 3–28. Elsevier, April 2002. doi: 10.1016/S0096-0551(02)00006-1.

CLARKE, Edmund M. (2000), Orna GRUMBERG, and Doron A. PELED. *Model Checking*. The MIT Press, Cambridge, Massachusetts 02142, USA, January 2000. ISBN: 0-262-03270-8. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3730.

COHEN, Ellis S. (1977). Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

Coq. The coq proof assistant, March 2007. http://pauillac.inria.fr/coq/.

COUSOT, Patrick (1996). Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996. ISSN: 0360-0300. doi: http://doi.acm.org/10.1145/234528.234740. http://www.di.ens.fr/~cousot/COUSOTpapers/CS96.shtml.

COUSOT, Patrick (1977) and Radhia COUSOT. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGACT-SIGPLAN Annual Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, January 1977. ACM Press. doi: http://doi.acm.org/10.1145/512950.512973. http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml.

CRANDALL, Jedidiah R. (2004) and Frederic T. CHONG. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, December 2004. IEEE Computer Society. ISBN: 0-7695-2126-6. doi: http://dx.doi.org/10.1109/MICRO.2004.26.

CRARY, Karl (2005), Aleksey KLIGER, and Frank PFENNING. A monadic analysis of information flow security with mutable sate. *Journal of Functional Programming*, 15(2):249–291, March 2005. ISSN: 0956-7968. doi: http://dx.doi.org/10.1017/S0956796804005441.

CYTRON, Ron (1991), Jeanne FERRANTE, Barry K. ROSEN, Mark N. WEGMAN, and F. Kenneth ZADECK. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. doi: http://doi.acm.org/10.1145/115372.115320.

DENNING, Dorothy E. (1982). *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1982.

DENNING, Dorothy E. (1976). A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/360051.360056.

DENNING, Dorothy E. (1977) and Peter J. DENNING. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/359636.359712.

ESWARAN, Sharanya (2006), Machigar ONGTANG, and David HADKA. Java information flow (jif) browser. Project report, Computer Science and Engineering, Pennsylvania State University, 342 IST Building, University Park, PA 16802, USA, December 2006a.

ESWARAN, Sharanya (2006), Machigar ONGTANG, and David HADKA. Jif Browser. `http://www.cse.psu.edu/~eswaran/CSE543-jifBrowser-Source.zip`, December 2006b. Software release.

FENTON, Jeffrey Stewart (1974). *Information Protection Systems*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge CB2 1TN, United Kingdom, May 1974a.

FENTON, Jeffrey Stewart (1974). Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974b. doi: 10.1093/comjnl/17.2.143. `http://comjnl.oxfordjournals.org/cgi/content/abstract/17/2/143`.

FOURNET, Cédric (2002) and Andrew D. GORDON. Stack inspection: theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 307–318. ACM Press, 2002. ISBN: 1-58113-450-9. doi: http://doi.acm.org/10.1145/503272.503301.

GANDHE, Milind (1995), G. VENKATESH, and Amitabha SANYAL. Labeled lambda-calculus and a generalized notion of strictness (an extended abstract). In *Proc. Asian C. S. Conf. on Algorithms, Concurrency and Knowledge*, pages 103–110. Springer-Verlag, 1995. ISBN: 3-540-60688-2.

Philippa Gardner and Nobuko Yoshida, editors. *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, 2004. Springer Berlin / Heidelberg. ISBN: 3-540-22940-X.

GAT, Israel (1976) and Harry J. SAAL. Memoryless execution: A programmer's viewpoint. *Software: Practice and Experience*, 6(4):463–471, October 1976. ISSN: 0038-6644. doi: 10.1002/spe.4380060404.

GENAIM, Samir (2005) and Fausto SPOTO. Information Flow Analysis for Java Bytecode. In Radhia Cousot, editor, *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag, January 2005.

GIACOBAZZI, Roberto (2004) and I. MASTROENI. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 186–197, Venice, Italy, January 2004. ACM-Press, NY. Venice, Italy, January 14-16,2004.

GODEFROID, Patrice (2005), Nils KLARLUND, and Koushik SEN. Dart: Directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the conference on Programming Language Design and Implementation*, volume 40 of *ACM SIGPLAN Notices*, pages 213–223, New York, NY, USA, June 2005. ACM Press. ISBN: 1-59593-056-6. doi: 10.1145/1065010.1065036.

GOGUEN, Joseph A. (1982) and José MESEGUER. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society, April 1982.

GOGUEN, Joseph A. (1984) and José MESEGUER. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86. IEEE Computer Society, April 1984.

GONG, Li (1997), M. MUELLER, H. PRAFULLCHANDRA, and R. SCHEMERS. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, 1997. http://www.usenix.org/publications/library/proceedings/usits97/full_papers/gong/gong.pdf.

GONG, Li (2003), Gary ELLISON, and Mary DAGEFORDE. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, May 2003. ISBN: 0-201-78791-1. Second Edition.

GOOD, Nathaniel S. (2003) and Aaron KREKELBERG. Usability and privacy: a study of Kazaa P2P file-sharing. In *Proceedings of the conference on Human factors in computing systems*, pages 137–144, New York, NY, USA, 2003. ACM Press. ISBN: 1-58113-630-7. doi: http://doi.acm.org/10.1145/642611.642636. Reprinted in (Good and Krekelberg, 2005).

GOOD, Nathaniel S. (2005) and Aaron KREKELBERG. Usability and Privacy: A Study of KaZaA P2P File Sharing. In Lorrie Faith Cranor and Simson Garfinkel, editors, *Security and Usability: Designing Secure Systems that People Can Use*, Theory In Practice, chapter 33, pages 651–668. O'Reilly Media, Inc., first edition, August 2005. ISBN: 0-596-00827-9.

GOSLING, James (2005), Bill JOY, Guy L. STEELE, Jr., and Gilad BRACHA. *The Java Language Specification*. Addison-Wesley Professional, 2005. ISBN: 0-321-24678-0. Third Edition.

GOUGH, K. John (2001). *Compiling for the .Net Common Language Runtime*. Prentice Hall PTR, Upper Saddle River, NJ, USA, October 2001. ISBN: 0130622966.

GUTTMAN, Joshua D. (2005), Amy L. HERZOG, John D. RAMSDELL, and Clement W. SKORUPKA. Verifying information flow goals in security-enhanced linux. *Journal of Computer Security*, 13(1):115–134, January 2005. ISSN: 0926-227X. Special issue on WITS'03.

HALDAR, Vivek (2005), Deepak CHANDRA, and Michael FRANZ. Practical, dynamic information flow for virtual machines. In *Proceedings of the International Workshop on Programming Language Interference and Dependence*, September 2005.

HEDIN, Daniel (2005) and David SANDS. Timing aware information flow security for a javacard-like bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 163–182, 2005.

HEINTZE, Nevin (1998) and Jon G. RIECKE. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.

HICKS, Boniface (2006), Kiyan AHMADIZADEH, and Patrick MCDANIEL. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, December 2006a. Awarded best student paper.

HICKS, Boniface (2006), Dave KING, Patrick MCDANIEL, and Michael HICKS. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)*. ACM Press, June 10 2006b. An extended version of this paper appears in the technical report NAS-TR-0033-2006.

HICKS, Michael (2005), Stephen TSE, Boniface HICKS, and Steve ZDANCEWIC. Dynamic updating of information-flow policies. In *Proc. of Foundations of Computer Security Workshop*, 2005.

HOARE, Charles Antony Richard (aka Tony) (1972). Towards a theory of parallel programming. In Charles Antony Richard Hoare and Ronald H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, 1972. Reprinted in (Brinch Hansen, 2002).

HOARE, Charles Antony Richard (aka Tony) (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. ISSN: 0001-0782. doi: 10.1145/363235.363259. http://portal.acm.org/citation.cfm?id=363259.

HOPCROFT, John E. (2001), Rajeev MOTWANI, and Jeffrey D. ULLMAN. *Introduction to Automata Theory, Languages, and Computation*, chapter 9.4, pages 392–403. Addison Wesley, second edition, 2001. ISBN: 0-201-44124-1.

HUNT, Sebastian (1991) and David SANDS. Binding time analysis: A new PERspective. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation*, pages 154–165, New York, NY, USA, September 1991. ACM Press. ISBN: 0-89791-433-3. doi: http://doi.acm.org/10.1145/115865.115881. ACM SIGPLAN Notices 26(9).

JACOBS, Bart (2005), Wolter PIETERS, and Martijn WARNIER. Statically checking confidentiality via dynamic labels. In *Workshop on Issues in the Theory of Security proceedings*, pages 50–56, New York, NY, USA, 2005. ACM Press. ISBN: 1581139802/05/01. doi: http://doi.acm.org/10.1145/1045405.1045411.

JONES, Anita K. (1975) and Richard J. LIPTON. The enforcement of security policies for computation. In *Proc. ACM Symp. on Operating System Principles*, pages 197–206, New York, NY, USA, 1975. ACM Press. doi: http://doi.acm.org/10.1145/800213.806538.

JONES, Neil D. (1995) and Flemming NIELSON. Abstract interpretation: A semantics-based tool for program analysis. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 527–636. Oxford University Press, Oxford, UK, June 1995. ISBN: 0-19-853780-8.

JOSHI, R. (2000) and K. R. M. LEINO. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

KAHN, David (1996). *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, second edition, December 1996. ISBN: 0684831309.

KAHN, Gilles (1987). Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39, Passau, Germany, February 1987. Springer-Verlag. ISBN: 3-540-17219-X.

Kröger, Fred (1987). *Temporal logic of programs*, volume 8 of *Monographs On Theoretical Computer Science*. Springer-Verlag New York, Inc., New York, NY, USA, May 1987. ISBN: 0-387-17030-8.

Lam, Lap Chung (2006) and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, December 2006. IEEE Computer Society. ISBN: 0-7695-2716-7. doi: http://dx.doi.org/10.1109/ACSAC.2006.6.

Lampson, Butler W. (1973). A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/362375.362389.

Le Guernic, Gurvan (2007). Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSFS20)*. IEEE Computer Society, July 6–8 2007a. ISBN: 0-7695-2819-8.

Le Guernic, Gurvan (2007). Automaton-based Non-interference Monitoring of Concurrent Programs. Technical Report 2007-1, Department of Computing and Information Sciences, College of Engineering, Kansas State University, 234 Nichols Hall, Manhattan, KS 66506, USA, February 2007b. http://www.cis.ksu.edu/~schmidt/techreport/2007.list.html.

Le Guernic, Gurvan (2005) and Thomas Jensen. Monitoring Information Flow. In Andrei Sabelfeld, editor, *Proceedings of the Workshop on Foundations of Computer Security*, pages 19–30. DePaul University, June 2005.

Le Guernic, Gurvan (2006), Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based Confidentiality Monitoring. In M. Okada and I. Satoh, editors, *Proceedings of the Annual Asian Computing Science Conference*, volume 4435 of *Lecture Notes in Computer Science*, pages 75–89, Berlin Heidelberg, December 6–8 2006a. Springer-Verlag.

Le Guernic, Gurvan (2006), Anindya Banerjee, and David A. Schmidt. Automaton-based Non-interference Monitoring. Technical Report 2006-1, Department of Computing and Information Sciences, College of Engineering, Kansas State University, 234 Nichols Hall, Manhattan, KS 66506, USA, April 2006b. http://www.cis.ksu.edu/~schmidt/techreport/2006.list.html.

Leroy, Xavier (2004), Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, July 2004. http://caml.inria.fr/ocaml/htmlman/.

Leyden, John (2006). Hk police complaints data leak puts city on edge. *The Register*, Tuesday 28th March 2006. http://www.theregister.co.uk/2006/03/28/hk_data_leak_rumpus/.

Ligatti, Jay (2005), Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec*, 4(1-2):2–16, 2005. http://dx.doi.org/10.1007/s10207-004-0046-8.

Lipner, Steven B. (1975). A comment on the confinement problem. In *Proc. ACM Symp. on Operating System Principles*, pages 192–196. ACM Press, 1975.

Mantel, Heiko (2001) and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 126–142, June 2001.

Masri, Wassim (aka Wes) (2005). *Dynamic Information Flow Analysis, Slicing and Profiling*. PhD thesis, Electrical Engineering and Computer Science Department, Engineering School, Case Western Reserve University, Cleveland, Ohio, USA, January 2005.

Masri, Wassim (aka Wes) (2004), Andy Podgurski, and David Leon. Detecting and debugging insecure information flows. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 198–209. IEEE, November 2004. doi: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2004.17.

Matos, Ana Almeida (2007) and Gérard Boudol. On Declassification and the Non-Disclosure Policy. *Journal of Computer Security*, March 2007. To appear.

McCamant, Stephen (2007) and Michael D. Ernst. A simulation-based proof technique for dynamic information flow. In *Proceedings of the workshop on Programming Languages and Analysis for Security*, June 2007.

McCue, Andy (2006). Cio jury: It bosses ban google desktop over security fears. *Silicon.com*, Thursday 2 March 2006. http://www.silicon.com/ciojury/0,3800003161,39156914,00.htm.

McDaniel, Patrick (2006), Boniface Hicks, Dave King, and Kiyan Ahmadizadeh. JPmail. http://siis.cse.psu.edu/jpmail/jpmaildetails.html, 2006a. Software release.

McDaniel, Patrick (2006), Boniface Hicks, Dave King, and Kiyan Ahmadizadeh. Jifclipse. http://siis.cse.psu.edu/jpmail/jifclipse.html, 2006b. Software release.

McHugh, J. (1985) and D. I. Good. An information flow tool for Gypsy. In *Proc. IEEE Symp. on Security and Privacy*, pages 46–48, April 1985.

McLean, J. (1994). A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.

Menezes, Alfred J. (1996), Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, volume 6 of *Discrete Mathematics and Its Applications*. CRC Press, October 1996. ISBN: 0-8493-8523-7. Fifth Printing.

Milner, Robin (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. doi: 10.1016/0022-0000(78)90014-4.

Minsky, Marvin Lee (1967). *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, June 1967. ISBN: 0-13-165449-7.

Mizuno, M. (1992) and David A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *J. Formal Aspects of Computing*, 4(6A):727–754, 1992.

Myers, Andrew C. (1999). JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. Principles of Programming Languages*, pages 228–241, January 1999a.

Myers, Andrew C. (1999). *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, January 1999b. Recipient of the George M. Sprowls Award for outstanding Ph.D. thesis in the MIT EECS Department.

Myers, Andrew C. (1997) and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. Operating System Principles*, pages 129–142, October 1997.

Myers, Andrew C. (1998) and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. Security and Privacy*, pages 186–197, May 1998.

Myers, Andrew C. (2001), N. Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. http://www.cs.cornell.edu/jif, July 2001. Software release.

Nagatou, Naoyuki (2006) and Takuo Watanabe. Run-time detection of covert channels. In *ARES*, pages 577–584. IEEE Computer Society, 2006.

National Security Agency (1995). Module eight - mandatory access control and labels, January 1995a. http://www.radium.ncsc.mil/tpep/library/ramp-modules/.

National Security Agency (1995). Module nine - discretionary access control, January 1995b. http://www.radium.ncsc.mil/tpep/library/ramp-modules/.

National Security Agency (2007). SELinux. http://www.nsa.gov/selinux/index.cfm, March 2007.

Naumann, David A. (2006). From coupling relations to mated invariants for checking information flow (extended abstract). In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Proc. European Symp. on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer, September 2006. ISBN: 978-3-540-44601-9. doi: 10.1007/11863908_18.

Newsome, James (2005) and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

Nielson, Flemming (1998) and Hanne Riis Nielson. Flow logic and operational semantics. In Andrew Gordon, Andrew Pitts, and Talcott Carolyn, editors, *Workshop on Higher-Order Operational Techniques in Semantics (December 1997)*, volume 10 of *Electronic Notes in Theoretical Computer Science*, pages 150–169. Elsevier, 1998. doi: 10.1016/S1571-0661(05)80695-4.

Nielson, Flemming (2005), Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2005. ISBN: 3-540-65410-0. http://www2.imm.dtu.dk/~riis/PPA/ppa.html. Corrected 2nd printing.

Ntafos, Simeon C. (1988). A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988. ISSN: 0098-5589. doi: http://dx.doi.org/10.1109/32.6165.

O'Hearn, Peter W. (2004). Resources, Concurrency and Local Reasoning. In Gardner and Yoshida (2004), pages 49–67. ISBN: 3-540-22940-X.

Ørbæk, P. (1995). Can you trust your data? In *Proc. TAPSOFT/FASE'95*, volume 915 of *Lecture Notes in Computer Science*, pages 575–590. Springer-Verlag, May 1995.

Page, Lewis (2007). Smut-swapping sailors leak secret missile specs. *The Register*, Thursday 5th April 2007. http://www.theregister.co.uk/2007/04/05/japanese_navy_porn_missile_shocker/.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts 02142, USA, February 2002. ISBN: 0-262-16209-1. http://www.cis.upenn.edu/~bcpierce/tapl/.

Pnueli, A. (1977). The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science*, pages 46–77, 1977.

Post, Emil L. (1946). A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

Pottier, François (2000) and Sylvain Conchon. Information flow inference for free. In *Proc. ACM International Conf. on Functional Programming*, pages 46–57. ACM Press, September 2000. ISBN: 1-58113-202-6. doi: http://doi.acm.org/10.1145/351240.351245.

Pottier, François (2003) and Vincent Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003a. ISSN: 0164-0925. doi: http://doi.acm.org/10.1145/596980.596983.

Pottier, François (2002) and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002a. ©ACM.

Pottier, François (2003) and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003b. ©ACM.

Pottier, François (2002) and Vincent Simonet. Information flow inference for ML. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330. ACM Press, 2002b. ISBN: 1-58113-450-9. doi: http://doi.acm.org/10.1145/503272.503302.

PVS. PVS: Specification and verification system, March 2007. `http://pvs.csl.sri.com/`.

Qin, Feng (2006), Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148. IEEE Computer Society, 2006.

Ranger, Steve (2007). Tk maxx customers urged to check card statements. *Silicon.com*, Friday 30 March 2007. `http://software.silicon.com/malware/0,3800003100,39166622,00.htm`.

Rotenberg, Leo Joseph (1973). *Making Computers Keep Secrets*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, June 1973.

Russo, Alejandro (2006), John Hughes, David A. Naumann, and Andrei Sabelfeld. Closing internal timing channels by transformation. In *Proceedings of the 11th Annual Asian Computing Science Conference*, LNCS. Springer-Verlag, December 6-8 2006.

Saal, Harry J. (1978) and Israel Gat. A hardware architecture for controlling information flow. In *ISCA '78: Proceedings of the 5th Annual Symposium on Computer Architecture*, pages 73–77, Palo Alto, California, April3–5 1978. IEEE Computer Society and ACM SIGARCH, ACM Press.

Sabelfeld, Andrei (2001). The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, July 2001.

Sabelfeld, Andrei (2003) and Andrew C. Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Communications*, 21(1):5–19, January 2003.

Sabelfeld, Andrei (2001) and David Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.

Sabelfeld, Andrei (2000) and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

Sabelfeld, Andrei (2007) and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007. To appear.

Saltzer, J. H. (1975) and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.

Schmidt, David A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Boston, Massachusetts, USA, 1986. http://www.cis.ksu.edu/~schmidt/text/densem.html.

Schmidt, David A. (1994). *The structure of typed programming languages*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts 02142, USA, March 1994. ISBN: 0-262-69171-X. http://www.cis.ksu.edu/~schmidt/text/info.html.

Schneider, Fred B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000. doi: http://doi.acm.org/10.1145/353323.353382.

Sen, Koushik (2006) and Gul Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the international conference on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, August 2006. ISBN: 3-540-37406-X. doi: 10.1007/11817963_38. (Tool Paper).

Shroff, Paritosh (2007), Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSFS20)*. IEEE Computer Society, July 6–8 2007. ISBN: 0-7695-2819-8. (Best Paper Award).

Simonet, Vincent (2002). Fine-grained information flow analysis for a λ-calculus with sum types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 223–237, June 2002a.

Simonet, Vincent (2003). Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003a.

Simonet, Vincent (2002). Fine-grained information flow analysis for a λ-calculus with sum types. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW 15)*, pages 223–237, Cape Breton, Nova Scotia (Canada), June 2002b. ©IEEE.

Simonet, Vincent (2003). The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003b. ©INRIA.

Simonet, Vincent (2003). Flow caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003c. citeseer.ist.psu.edu/simonet03flow.html.

Simonet, Vincent (2003). The flow caml system: Documentation and user's manual. Technical Report RT-0282, INRIA-Rocquencourt, July 2003d.

Smith, G. (1998) and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.

Stoy, Joseph E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, Massachusetts, USA, 1977.

SUH, G. Edward (2004), Jae W. LEE, David X. ZHANG, and Srinivas DEVADAS. Secure program execution via dynamic information flow tracking. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 85–96. ACM Press, 2004. ISBN: 1-58113-804-0. doi: http://doi.acm.org/10.1145/1024393.1024404.

SUN, Qi (2004), Anindya BANERJEE, and David A. NAUMANN. Modular and constraint-based information flow inference for an object-oriented language. In *Proceedings of the Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer Berlin / Heidelberg, August 2004. ISBN: 978-3-540-22791-5. doi: 10.1007/b99688.

SUTHERLAND, D. (1986). A model of information. In *Proceedings of the National Computer Security Conference*, pages 175–183, September 1986.

SYROPOULOS, Apostolos (2001). Mathematics of multisets. In Cristian S. Calude, Gheorghe Pun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Proceedings of the Workshop on Multiset Processing (Mathematical, Computer Science, and Molecular Computing Points of View)*, volume 2235 of *Lecture Notes in Computer Science*, pages 347–358. Springer-Verlag, 2001. ISBN: 3-540-43063-6. `http://www.springerlink.com/content/7dwngkyuvykd2tgc`.

TERAUCHI, Tachio (2005) and Alexander AIKEN. Secure information flow as a safety problem. In *Proceedings of the International Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, September 2005. ISBN: 3-540-28584-9. doi: 10.1007/11547662_24.

TSE, Stephen (2007) and Steve ZDANCEWIC. Run-time principals in information-flow type systems. *ACM Transactions on Programming Languages and Systems*, 2007. To appear.

VACHHARAJANI, Neil (2004), Matthew J. BRIDGES, Jonathan CHANG, Ram RANGAN, Guilherme OTTONI, Jason A. BLOME, George A. REIS, Manish VACHHARAJANI, and David I. AUGUST. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2004.

VENKATAKRISHNAN, V. N. (2006), Wei XU, Daniel C. DUVARNEY, and R. SEKAR. Provably correct runtime enforcement of non-interference properties. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Proceedings of the International Conference on Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 332–351. Springer-Verlag, December 2006. ISBN: 978-3-540-49496-6. doi: 10.1007/11935308_24. `http://www.springerlink.com/content/lmn4768325081h8w`.

VISWANATHAN, Mahesh (2000). *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, December 2000.

VOLPANO, D. (1997) and G. SMITH. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997a.

VOLPANO, D. (1999) and G. SMITH. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, November 1999.

VOLPANO, D. (1997) and G. SMITH. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, April 1997b.

VOLPANO, D. (1996), G. SMITH, and C. IRVINE. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

Wall, Larry (2000), Tom Christiansen, and Jon Orwant. *Programming Perl*, section Handling Insecure Data, pages 558–568. O'Reilly, third edition, July 2000. ISBN: 0-596-00027-8.

Weissman, Clark (1969). Security controls in the adept-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 35, pages 119–133, 1969.

Williams, Nicky (2005), Bruno Marre, Patricia Mouy, and Roger Muriel. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *Proceedings of the European Dependable Computing Conference*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292, Berlin / Heidelberg, April 2005. Springer. ISBN: 978-3-540-25723-3. doi: 10.1007/b107276.

Woodward, John P. L. (1987). Exploiting the dual nature of sensitivity labels. In *Proc. IEEE Symp. Security and Privacy*, pages 23–31, Oakland, CA, April 1987.

Xu, Wei (2006), Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium*, pages 121–136, August 2006. `http://www.usenix.org/events/sec06/tech/xu.html`.

Zdancewic, Steve (2001), Lantian Zheng, N. Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 1–14, October 2001.

Zheng, Lantian (2004) and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. Workshop Formal Aspects in Security and Trust*, August 2004.

List of Definitions and Lemmas Used in Proofs

Appendix A

Proofs of Chapter 3's Theorems

A.1 Soundness

Lemma A.1.1 (Same context before and after an execution). *For all statement S, automaton states $q = (V, w)$ and $q_f = (V_f, w_f)$, and value store σ, if $(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$ then $w_f = w$.*

Proof. The fact that $w_f = w$ follows directly from the definition of the semantics ($\mathrm{E}_{\mathcal{M}(O)}$), and the definition of the transition function of the monitoring automaton.

Lemma A.1.2 (No outputs under secret context).
For all statement S, automaton state (V, w), and value store σ, if $w \notin \{\bot\}^\star$ then $[\![S]\!]^{O}_{\mathcal{M}(O)}((V, w), \sigma) = \epsilon$.

Proof. This follows directly from the definitions of the semantics ($\mathrm{E}_{\mathcal{M}(O)}$) and ($\mathrm{E}_O$) and from the transition function of the monitoring automaton. The only statement outputting anything is "**output** e". The semantics ($\mathrm{E}_{\mathcal{M}(O)}$) always call the automaton to verify any action it will evaluates. Whenever the context of execution is secret (i.e. $w \notin \{\bot\}^\star$), the monitoring automaton deny the execution of any print statement.

Lemma A.1.3 (Automaton simulates execution in secret context).
For all statement S, automaton state $q = (V, w)$, and value store σ, if $(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$ and $w \notin \{\bot\}^\star$ then $(q, \texttt{not}\ S) \to q_f$.

Proof. The proof goes by induction on the derivation tree of "$((V, w), \sigma) \vdash S \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($\mathrm{E}_{\mathcal{M}(O)}$-OK) then we can conclude that :

> **(1)** $S =$ "**skip**" or $S =$ "y **:=** e" or $S =$ "**output** e", and "$(q, S) \xrightarrow{OK} q_f$".
> This follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-OK) and the grammar of the language. It can also be deduced from the rule ($\mathrm{E}_{\mathcal{M}(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E_O).
>
> **Case 1:** $S =$ "**skip**" or $S =$ "**output** e"

(a) $q_f = q$.
This follows directly from the transition function of the monitoring automaton.

(•) $(q, \texttt{not}\, S) \to q_f$.
This follows directly from the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}(S) = \emptyset$.

Case 2: S = "x **:=** e"

(a) $q_f = (V \cup \{x\}, w)$.
This follows directly from the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$.

(•) $(q, \texttt{not}\, S) \to q_f$.
This follows directly from the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}(x$ **:=** $e) = \{x\}$.

(E$_{\mathcal{M}(O)}$-EDIT) then we can conclude that :

(1) S = "**output** e", and "$(q, S) \xrightarrow{A'} q_f$".
This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-EDIT) and the grammar of the language. It can also be deduced from the rule (E$_{\mathcal{M}(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E$_O$).

(2) $q_f = q$.
This follows directly from the transition function of the monitoring automaton.

(•) $(q, \texttt{not}\, S) \to q_f$.
This follows directly from the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}($**output** $e) = \emptyset$.

(E$_{\mathcal{M}(O)}$-NO) then we can conclude that :

(1) S = "**output** e", and "$(q, S) \xrightarrow{A'} q_f$".
This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-NO) and the grammar of the language. It can also be deduced from the rule (E$_{\mathcal{M}(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E$_O$).

(2) $q_f = q$.
This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-NO).

(•) $(q, \texttt{not}\, S) \to q_f$.
This follows directly from the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}($**output** $e) = \emptyset$.

(E$_{\mathcal{M}(O)}$-IF) then we can conclude that :

(1) S = "**if** e **then** S_1 **else** S_2 **end**", $\sigma(e) = v$, and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_v \overset{o}{\Longrightarrow}_{\mathcal{M}(O)} (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q_3$
- $(q_3,\ \texttt{exit}) \xrightarrow{OK} q_f$

This follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-IF).

Let us define $q_i = (V_i, w_i)$ for all integer i from 1 to 3.

(2) $w_1 \notin \{\bot\}^\star$.
This follows directly from the global hypothesis $w \notin \{\bot\}^\star$ and the definition of the transition function of the monitoring automaton.

(3) $V_2 = V \cup \mathit{modified}(S_v)$.
This follows from the application of the inductive hypothesis to the evaluation of S_v in the local conclusion (1) and the definition of the transition (T-NOT-high).

(4) $q_f = (V \cup \mathit{modified}(S_v) \cup \mathit{modified}(S_{\neg v}), w)$.
From the local conclusion (2), the evaluation of S_v in the local conclusion (1) and lemma A.1.1, it is possible to show that $w_2 \notin \{\bot\}^\star$. So, the definition of the transition function of the monitoring automaton imply that $V_3 = V_2 \cup \mathit{modified}(S_{\neg v})$. Finally the desired result is obtain using the local conclusion (3), the definition of (T-EXIT) and lemma A.1.1.

(•) $(q, \texttt{not}\ S) \to q_f$.
This follows directly from the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}(\textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2)$.

($\mathrm{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$) then we can conclude that :

(1) S = "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_l \overset{o_l}{\Longrightarrow}_{\mathcal{M}(O)} (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$
- $(q_3,\ \sigma_1) \vdash$ **while** e **do** S_l **done** $\overset{o_w}{\Longrightarrow}_{\mathcal{M}(O)} (q_f,\ \sigma_2)$

This follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$).

Let us define $q_i = (V_i, w_i)$ for all integer i from 1 to 3.

(2) $w_1 \notin \{\bot\}^\star$.
This follows directly from the global hypothesis $w \notin \{\bot\}^\star$ and the definition of the transition function of the monitoring automaton.

(3) $V_3 = V \cup \mathit{modified}(S_l)$.
This follows from the application of the inductive hypothesis to the evaluation of S_l in the local conclusion (1) and the definition of the transition (T-NOT-high) and (T-EXIT).

(4) $V_f = V \cup \mathit{modified}(S_l)$.

The local conclusion (2) and the definition of the transition function imply $w_3 \notin \{\bot\}^\star$. Using the inductive hypothesis on the evaluation of **while** e **do** S_l **done** in the local conclusion (1), it is possible to show that $V_f = V_2 \cup \mathit{modified}(S_l)$ because $\mathit{modified}(\textbf{while } e \textbf{ do } S_l \textbf{ done}) = \mathit{modified}(S_l)$. Then, the local conclusion (3) implies the desired result.

(•) $(q, \texttt{not}\ S) \to q_f$.

This follows directly from the local conclusion (4) and the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}(\textbf{while } e \textbf{ do } S_l \textbf{ done}) = \mathit{modified}(S_l)$.

($\mathbf{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) S = "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q_2$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_f$

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$).

Let us define $q_i = (V_i, w_i)$ for all integer i from 1 to 2.

(2) $w_1 \notin \{\bot\}^\star$.

This follows directly from the global hypothesis $w \notin \{\bot\}^\star$ and the definition of the transition function of the monitoring automaton.

(3) $V_f = V \cup \mathit{modified}(S_l)$.

This follows from the local conclusion (2) and the definition of the transition function of the monitoring automaton.

(•) $(q, \texttt{not}\ S) \to q_f$.

This follows directly from the local conclusion (3) and the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}(\textbf{while } e \textbf{ do } S_l \textbf{ done}) = \mathit{modified}(S_l)$.

($\mathbf{E}_{\mathcal{M}(O)}$-SEQ) then we can conclude that :

(1) S = "S_1 ; S_2" and:

- $(q,\ \sigma) \vdash S_1 \overset{o_1}{\Longrightarrow}_{\mathcal{M}(O)} (q_1,\ \sigma_1)$
- $(q_1,\ \sigma_1) \vdash S_2 \overset{o_2}{\Longrightarrow}_{\mathcal{M}(O)} (q_f,\ \sigma_f)$

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-SEQ).

(2) $(q, \texttt{not}\ S_1) \to q_1$ and $(q_1, \texttt{not}\ S_2) \to q_f$.

Those results follow from the inductive hypothesis. The second result also makes use of lemma A.1.1 in order to prove that $w_1 \notin \{\bot\}^\star$.

(3) $q_f = (V \cup \mathit{modified}(S_1) \cup \mathit{modified}(S_2), w)$.

This follows directly from the local conclusion (2), the global hypothesis $w \notin \{\bot\}^\star$, lemma A.1.1, and the definition of the rule (T-NOT-high).

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
This follows directly from the transition function of the monitoring automaton because $w \notin \{\bot\}^\star$ and $\mathit{modified}(S_1\ ;\ S_2) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2)$.

Lemma A.1.4 (*modified*(S) is an over-approximation of the assigned variables in S). *For all statements S, automaton states $q = (V, w)$, and value stores σ, if " $(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$" with $q_f = (V_f, w_f)$ then:*

- $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$
- $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$

Proof. The proof goes by induction on the derivation tree of "$(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($\mathbf{E}_{\mathcal{M}(O)}$-OK) then we can conclude that :

(1) S = "**skip**" or S = "y **:=** e" or S = "**output** e", "$(q, S) \xrightarrow{OK} q_f$", and "$\sigma \vdash S \overset{o}{\Rightarrow}_O \sigma_f$".
This follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-OK) and the grammar of the language. It can also be deduced from the rule ($\mathrm{E}_{\mathcal{M}(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E_O).

Case 1: S = "**skip**" or S = "**output** e"

(a) $q_f = q$ and $\sigma_f = \sigma$.
This follows directly from the transition function of the monitoring automaton and the definition of the semantics (E_O).

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.
This follows directly from the local conclusion (a).

Case 2: S = "x **:=** e"

(a) $q_f = (V \cup \{x\}, w)$ or $q_f = (V - \{x\}, w)$, and $\sigma_f = \sigma[x \mapsto \sigma(e)]$.
This follows directly from the transition function of the monitoring automaton and the definition of the semantics (E_O).

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.
This follows directly from the local conclusion (a) because $\mathit{modified}(S) = \{x\}$.

($\mathbf{E}_{\mathcal{M}(O)}$-EDIT) then we can conclude that :

(1) S = "**output** e", "$(q, S) \xrightarrow{\textbf{output}\ \theta} q_f$", and "$\sigma \vdash \textbf{output}\ \theta \overset{o}{\Rightarrow}_O \sigma_f$".
This follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-EDIT), the grammar of the language, and the definition of the transition function of the monitoring automaton.

(2) $q_f = q$ and $\sigma_f = \sigma$.
This follows directly from the local conclusion (1), the transition function of the monitoring automaton and the definition of the semantics (E_O).

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.
This follows directly from the local conclusion (2).

($\mathbf{E}_{\mathcal{M}(O)}$-NO) then we can conclude that :

(1) $q_f = q$ and $\sigma_f = \sigma$.
This follows directly from the definition of ($\mathrm{E}_{\mathcal{M}(O)}$-NO), and the definition of the transition function of the monitoring automaton.

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.
This follows directly from the local conclusion (1).

($\mathbf{E}_{\mathcal{M}(O)}$-IF) then we can conclude that :

(1) S = "**if** e **then** $S_{\texttt{true}}$ **else** $S_{\texttt{false}}$ **end**", $\sigma(e) = v$, and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_v \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_2,\ \sigma_f)$
- $(q_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q_3$
- $(q_3,\ \texttt{exit}) \xrightarrow{OK} q_f$

This follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-IF).

(2) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$.
This follows from the inductive hypothesis used on the derivation tree of S_v in the local conclusion (1) and the fact that $\mathit{modified}($**if** e **then** $S_{\texttt{true}}$ **else** $S_{\texttt{false}}$ **end**$) = \mathit{modified}(S_{\texttt{true}}) \cup \mathit{modified}(S_{\texttt{false}})$.

Let us define $q_i = (V_i, w_i)$ for all integer i from 1 to 3.

(3) $V_1 = V$.
This follows directly from the definition of the transition function of the monitoring automaton.

(4) $V - \mathit{modified}(S_v) \subseteq V_2 \subseteq V \cup \mathit{modified}(S_v)$.
This follows from the inductive hypothesis applied to the derivation tree of S_v.

(5) $V_2 \subseteq V_f \subseteq V_2 \cup \mathit{modified}(S_{\neg v})$.
This follows from the local conclusion (1) and the definition of the transition function of the monitoring automaton.

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.
This follows from the fact that $\mathit{modified}($**if** e **then** $S_{\texttt{true}}$ **else** $S_{\texttt{false}}$ **end**$) = \mathit{modified}(S_{\texttt{true}}) \cup \mathit{modified}(S_{\texttt{false}})$ and the local conclusions (2), (4), and (5).

($\mathbf{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$) then we can conclude that :

(1) S = "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$

- $(q_1,\ \sigma)\ \vdash\ S_l\ \overset{o_l}{\Longrightarrow}_{\mathcal{M}(O)}\ (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{exit})\ \xrightarrow{OK}\ q_3$
- $(q_3,\ \sigma_1)\ \vdash$ **while** e **do** S_l **done** $\overset{o_w}{\Longrightarrow}_{\mathcal{M}(O)}\ (q_f,\ \sigma_f)$

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$).

(2) $V_1 = V$.

This follows directly from the definition of the transition function of the monitoring automaton.

(3) $\forall x \notin \mathit{modified}(S_l) : \sigma_1(x) = \sigma(x)$ and $V - \mathit{modified}(S_l) \subseteq V_2 \subseteq V_2 \cup \mathit{modified}(S_l)$.

This follows from the inductive hypothesis applied to the derivation tree of S_l found in the local conclusion (1).

(4) $V_3 = V_2$.

This follows directly from the definition of (T-EXIT).

(5) $\forall x \notin \mathit{modified}(S_l) : \sigma_f(x) = \sigma_1(x)$ and $V_3 - \mathit{modified}(S_l) \subseteq V_f \subseteq V_3 \cup \mathit{modified}(S_l)$.

This follows from the inductive hypothesis applied to the derivation tree of **while** e **do** S_l **done** found in the local conclusion (1) because $\mathit{modified}$(**while** e **do** S_l **done**) $= \mathit{modified}(S_l)$.

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.

This follows from the local conclusions (3), (4), and (5) because $\mathit{modified}$(**while** e **do** S_l **done**) $=$ $\mathit{modified}(S_l)$.

(E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) S = "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e)\ \xrightarrow{OK}\ q_1$
- $(q_1,\ \texttt{not}\ S_l)\ \xrightarrow{OK}\ q_2$
- $(q_2,\ \texttt{exit})\ \xrightarrow{OK}\ q_f$

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$).

(2) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$.

This follows directly from the definition of (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$).

(3) $V_f = V$ or $V_f = V \cup \mathit{modified}(S_l)$.

This follows directly from the definition of the transition function of the monitoring automaton.

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.

This follows directly from the local conclusions (2) and (3) because $\mathit{modified}$(**while** e **do** S_l **done**) $= \mathit{modified}(S_l)$.

(E$_{\mathcal{M}(O)}$-SEQ) then we can conclude that :

(1) S = "$S_1\ ;\ S_2$" and:

- $(q,\ \sigma)\ \vdash\ S_1\ \overset{o_1}{\Longrightarrow}_{\mathcal{M}(O)}\ (q_1,\ \sigma_1)$
- $(q_1,\ \sigma_1)\ \vdash\ S_2\ \overset{o_2}{\Longrightarrow}_{\mathcal{M}(O)}\ (q_f,\ \sigma_f)$

This follows directly from the definition of the rule ($E_{\mathcal{M}(O)}$-SEQ).

(2)
- $\forall x \notin \mathit{modified}(S_1) : \sigma_1(x) = \sigma(x)$
- $V - \mathit{modified}(S_1) \subseteq V_1 \subseteq V \cup \mathit{modified}(S_1)$
- $\forall x \notin \mathit{modified}(S_2) : \sigma_f(x) = \sigma_1(x)$
- $V_1 - \mathit{modified}(S_2) \subseteq V_f \subseteq V_1 \cup \mathit{modified}(S_2)$

Those results follow from the inductive hypothesis.

(•) $\forall x \notin \mathit{modified}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathit{modified}(S) \subseteq V_f \subseteq V \cup \mathit{modified}(S)$.

This follows directly from the local conclusion (2) because $\mathit{modified}(S_1\ ;\ S_2) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2)$.

Lemma A.1.5 (WDKL (while-dilemma killer lemma)). *For all statement S, expression e, automaton state $q = (V, w)$, and value store σ, if $FV(e) \cap V \neq \emptyset$ and $(q, \sigma) \vdash$ **while** e **do** S **done** $\overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$ with $q_f = (V_f, w_f)$ then:*

- $o = \epsilon$
- $V \subseteq V_f$

Proof. The proof goes by induction on the derivation tree of "$(q, \sigma) \vdash$ **while** e **do** S **done** $\overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$". The last rule used by the derivation tree is either ($E_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$) or ($E_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$). Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($E_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$) then we can conclude that :

(1) S = **while** e **do** S_l **done** and:
- $\sigma(e) = \texttt{true}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_l \overset{o_l}{\Longrightarrow}_{\mathcal{M}(O)} (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$
- $(q_3,\ \sigma_1) \vdash$ **while** e **do** S_l **done** $\overset{o_w}{\Longrightarrow}_{\mathcal{M}(O)} (q_f,\ \sigma_f)$
- $o = o_l\, o_w$

This follows directly from the definition of the rule ($E_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$).

(2) $w_1 \notin \{\perp\}^\star$.

This follows directly from the definition of the transition function of the monitoring automaton and the global hypothesis $FV(e) \cap V \neq \emptyset$.

(3) $o_l = \epsilon$.

This follows from the local conclusions (1) and (2), and from lemma A.1.2.

(4) $w_3 \notin \{\perp\}^\star$.

This follows from the local conclusions (1) and (2), lemma A.1.1, and the definition of the transition function of the monitoring automaton.

(5) $o_w = \epsilon$.
This follows from the local conclusions (1) and (4), and from lemma A.1.2.

(6) $V \subseteq V_3$.
From lemma A.1.3 and the local conclusion (1), $(q_1, \texttt{not}\ S_l) \xrightarrow{OK} q_2$. Then, the desired result follows from the definition of the transition function of the monitoring automaton.

(7) $V_3 \subseteq V_f$.
This follows directly from the inductive hypothesis which can be applied because of the local conclusion (1) and the fact that the global hypothesis $FV(e) \cap V \neq \emptyset$ still holds.

(•) $o = \epsilon$ and $V \subseteq V_f$.
This follows directly from the local conclusions (1), (3), (5), (6) and (7).

($\mathbf{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) S = **while** e **do** S_l **done** and:

- $\sigma(e) = \texttt{false}$
- $(q, \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1, \texttt{not}\ S_l) \xrightarrow{OK} q_2$
- $(q_2, \texttt{exit}) \xrightarrow{OK} q_3$

This follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$).

(•) $o = \epsilon$ and $V \subseteq V_f$.
The fact that $o = \epsilon$ follows directly from the definition of the rule ($\mathrm{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$). The fact that $V \subseteq V_f$ follows directly from the local conclusion (1) and the definition of the transition function of the monitoring automaton.

Lemma A.1.6 (Correctness). *For all statement S, automaton state $q = (V, w)$, value stores σ and σ' such that:*

$\star_1$ $(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$,

$\star_2$ $(q, \sigma') \vdash S \overset{o'}{\Rightarrow}_{\mathcal{M}(O)} (q'_f, \sigma'_f)$,

$\star_3$ $\forall x \notin V : \sigma(x) = \sigma'(x)$,

it is true that, there exist a variable set V_f and an integer w_f such that:

- $o = o'$, $q_f = q'_f = (V_f, w_f)$, *and* $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

Proof. The proof goes by induction on the derivation tree of "$((V, w), \sigma) \vdash S \overset{o}{\Rightarrow}_{\mathcal{M}(O)} (q_f, \sigma_f)$". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($\mathbf{E}_{\mathcal{M}(O)}$-OK) then we can conclude that :

(1) S = **skip** or S = y **:=** e or S = **output** e, "$(q, S) \xrightarrow{OK} q'$" and "$\sigma \vdash S \overset{o}{\Rightarrow}_O \sigma'$".

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-OK) and the grammar of the language. It can also be deduced from the rule (E$_{\mathcal{M}(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E$_O$).

Case 1: S = **skip**

(a) $o = \epsilon$, $q_f = q$, and $\sigma_f = \sigma$

This follows from the case hypothesis, the facts that "$(q, S) \xrightarrow{OK} q'$" and "$\sigma \vdash S \overset{o}{\Rightarrow}_O \sigma'$" (in the local conclusion (1)), the definition of the only transition on **skip** for the monitoring automaton (T-SKIP), and the definition of the only rule applying to **skip** in (E$_O$) (E$_O$-SKIP).

(b) $o' = \epsilon$, $q'_f = q$, and $\sigma'_f = \sigma'$

This follows from the global hypothesis $\star_2$ and the reasons invoked for the local conclusion (a).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

This follows from the global hypothesis $\star_3$ and the local conclusions (a) and (b).

Case 2: S = y **:=** e

Sub-case 2.a: $w \notin \{\bot\}^\star$ or $FV(e) \cap V_f \neq \emptyset$

(α) $o = \epsilon$, $q_f = (V \cup \{y\}, w)$, and $\sigma_f = \sigma[y \mapsto \sigma(e)]$

This follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{\mathcal{M}(O)}$) and (E_O).

(β) $o' = \epsilon$, $q'_f = (V \cup \{y\}, w)$, and $\sigma'_f = \sigma'[y \mapsto \sigma'(e)]$

This follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{\mathcal{M}(O)}$) and (E_O).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

This follows from the global hypothesis $\star_3$ and the local conclusions (α) and (β).

Sub-case 2.b: $w \in \{\bot\}^\star$ and $FV(e) \cap V_f = \emptyset$

(α) $o = \epsilon$, $q_f = (V - \{y\}, w)$, and $\sigma_f = \sigma[y \mapsto \sigma(e)]$

This follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{\mathcal{M}(O)}$) and (E_O).

(β) $o' = \epsilon$, $q'_f = (V - \{y\}, w)$, and $\sigma'_f = \sigma'[y \mapsto \sigma'(e)]$

This follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{\mathcal{M}(O)}$) and (E_O).

(γ) $\sigma(e) = \sigma'(e)$

This follows from the fact $FV(e) \cap V_f = \emptyset$ (from the sub-case hypothesis) and the global hypothesis $\star_3$.

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
This follows from the global hypothesis $\star_3$ and the local conclusions (α), (β), and (γ).

Case 3: S = **output** e

(a) $w \in \{\bot\}^\star$ and $FV(e) \cap V_f = \emptyset$
This follows from the definition of the rule (E$_{\mathcal{M}(O)}$-OK), the case hypothesis, and the definition (T) of the transition function of the monitoring automaton.

(b) $o = \sigma(e)$, $q_f = q$, and $\sigma_f = \sigma$
This follows from the case hypothesis, the facts that "$(q, S) \xrightarrow{OK} q'$" and the only such transition for **output** e (T-PRINT-ok), and the definition of the semantics (E$_O$).

(c) $o' = \sigma(e)$, $q'_f = q$, and $\sigma'_f = \sigma$
This follows from the case hypothesis, the local conclusion (a), the definition (T) of the transition function of the monitoring automaton, and the definition of the semantics (E$_{\mathcal{M}(O)}$) and (E$_O$).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
This follows from the global hypothesis $\star_3$ and the local conclusions (b) and (c).

(E$_{\mathcal{M}(O)}$-EDIT) then we can conclude that :

(1) "$(q, S) \xrightarrow{A'} q'$" and "$\sigma \vdash A' \overset{o}{\Rightarrow}_O \sigma'$".
This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-EDIT).

(2) S = **output** e, A' = **output** θ, $w \in \{\bot\}^\star$, and $FV(e) \cap V \neq \emptyset$.
This follows directly from the only transition outputting an evaluable transition (T-PRINT-def).

(3) $o = \sigma(\theta)$, $q_f = q$, and $\sigma_f = \sigma$
This follows from the local conclusion (2), the definition of the transition function of the monitoring automaton (T), and the definition of the semantics (E$_O$).

(4) $o' = \sigma(\theta)$, $q'_f = q$, and $\sigma'_f = \sigma$
This follows from the fact that $w \in \{\bot\}^\star$ and $FV(e) \cap V \neq \emptyset$ (from the local conclusion (2)), the definition of the semantics (E$_{\mathcal{M}(O)}$) for actions, the definition of the only transition of the monitoring automaton (T) for print actions whenever $w \in \{\bot\}^\star$ and $FV(e) \cap V \neq \emptyset$, and the definition of the semantics (E$_O$).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
This follows from the global hypothesis $\star_3$ and the local conclusions (3) and (4).

(E$_{\mathcal{M}(O)}$-NO) then we can conclude that :

(1) "$(q, S) \xrightarrow{NO} q'$".
This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-NO).

(2) S = **output** e, and $w \notin \{\bot\}^\star$.
This follows directly from the only transition outputting "NO" (T-PRINT-no).

(3) $o = \epsilon$, $q_f = q$, and $\sigma_f = \sigma$

This follows directly from the definition of the rule ($\text{E}_{\mathcal{M}(O)}$-NO).

(4) $o' = \epsilon$, $q'_f = q$, and $\sigma'_f = \sigma$

This follows from the fact that $w \notin \{\perp\}^\star$ (from the local conclusion (2)), the definition of the only transition of the monitoring automaton (T) for print actions in such a case, and the definition of the semantics ($\text{E}_{\mathcal{M}(O)}$) for actions whenever the automaton outputs "NO".

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

This follows from the global hypothesis $\star_3$ and the local conclusions (3) and (4).

($\text{E}_{\mathcal{M}(O)}$-IF) then we can conclude that :

(1) S = **if** e **then** S_1 **else** S_2 **end** and:

- σ(e) = v
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_v \overset{o}{\Longrightarrow}_{\mathcal{M}(O)} (q_2,\ \sigma_f)$
- $(q_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q_3$
- $(q_3,\ \texttt{exit}) \xrightarrow{OK} q_f$

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

Case 1: $\sigma'(e) = v$

(a)
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma') \vdash S_v \overset{o'}{\Longrightarrow}_{\mathcal{M}(O)} (q'_2,\ \sigma'_f)$
- $(q'_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q'_3$
- $(q'_3,\ \texttt{exit}) \xrightarrow{OK} q'_f$

This follows from the case hypothesis, the local conclusion (1), the definition of the only rule applying to if-statements ($\text{E}_{\mathcal{M}(O)}$-IF), and the definition of the transition rules (T-BRANCH-high) and (T-BRANCH-low) (Both evaluations use the same rule as the conditions for those depend only e and V).

(b) $o = o'$, $q_2 = q'_2 = (V_2, w_2)$, and $\forall x \notin V_2 : \sigma_f(x) = \sigma'_f(x)$.

This result is obtained by applying the inductive hypothesis to the derivations of S_v found in the local conclusions (1) and (a).

(c) There exists a such that $w_2 = w_f a$ and if $w_2 \notin \{\perp\}^\star$ then $q_f = (V_2 \cup \mathit{modified}(S_{\neg v}), w_f) = q'_f$ else $q_f = (V_2, w_f) = q'_f$.

This follows directly from the definition of the transition function of the monitoring automaton and the local conclusions (1), (a) and (b).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

This follows directly from the local conclusions (b) and (c) because all x, which does not belong to V_f, does not belong to V_2.

Case 2: $\sigma'(e) = \neg v$

(a) $FV(e) \cap V \neq \emptyset$.

The negation of this property is in contradiction with the case hypothesis, the local conclusion (1), and the global hypothesis $\star_3$.

(b) • $(q, \texttt{branch}\, e) \xrightarrow{OK} q_1$

• $(q_1, \sigma') \vdash S_{\neg v} \overset{o'}{\Longrightarrow}_{\mathcal{M}(O)} (q'_2, \sigma'_f)$

• $(q'_2, \texttt{not}\, S_v) \xrightarrow{OK} q'_3$

• $(q'_3, \texttt{exit}) \xrightarrow{OK} q'_f$

This follows from the case hypothesis, the local conclusion (1), the definition of the only rule applying to if-statements (E$_{\mathcal{M}(O)}$-IF), and the definition of the transition rule (T-BRANCH-high).

(c) $o = o' = \epsilon$.

Let $q_1 = (V_1, w_1)$. The local conclusion (1), and the definition of (T-BRANCH-high) imply that w_1 does not belong to $\{\bot\}^\star$. This and lemma A.1.2 imply the above result.

Let q_a and $q_b = (V_b, w_b)$ be monitoring automaton states such that "$(q_1, \texttt{not}S_v) \xrightarrow{OK} q_a$" and "$(q_a, \texttt{not}S_{\neg v}) \xrightarrow{OK} q_b$".

(d) There exists q_c such that "$(q_1, \texttt{not}S_{\neg v}) \xrightarrow{OK} q_c$" and "$(q_c, \texttt{not}S_v) \xrightarrow{OK} q_b$".

It is obvious from the definition of (T-NOT-high) and (T-NOT-low).

(e) $q_3 = q_b = q'_3$.

This follows from lemma A.1.3 and the local conclusions (1), (b), and (d).

(f) $q_f = q'_f$.

This follows directly from the definition of (T-EXIT) and the local conclusions (e), (1), and (b).

(g) $V_b = V \cup \mathit{modified}(S_v) \cup \mathit{modified}(S_{\neg v})$.

In the proof of local conclusion (c), we proved that w_1 does not belong to $\{\bot\}^\star$. Then the above result follows directly from the definition of (T-NOT-high).

(h) $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

From the definition of (T-EXIT) and the local conclusions (1), (e), and (g), $V_f = V \cup \mathit{modified}(S_v) \cup \mathit{modified}(S_{\neg v})$. From the evaluation of S_v (in the local conclusion (1)) and lemma A.1.4, if x does not belongs to V_f, and so does not belongs to $\mathit{modified}(S_v)$, $\sigma_f(x) = \sigma(x)$. From the evaluation of $S_{\neg v}$ (in the local conclusion (b)) and lemma A.1.4, if x does not belongs to V_f, and so does not belongs to $\mathit{modified}(S_{\neg v})$, $\sigma'_f(x) = \sigma'(x)$. Additionally, if x does not belongs to V_f then it does not belongs to V; which implies that $\sigma(x) = \sigma'(x)$ because of the global hypothesis $\star_3$. Finally, those three equalities imply that if x does not belongs to V_f then $\sigma_f(x) = \sigma'_f(x)$.

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

This follows directly from the local conclusions (c), (f) and (h).

($\mathbf{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$) then we can conclude that :

(1) S = **while** e **do** S_l **done** and:

- $\sigma(e) = \texttt{true}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_l \overset{o_l}{\Longrightarrow}_{\mathcal{M}(O)} (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$
- $(q_3,\ \sigma_1) \vdash$ **while** e **do** S_l **done** $\overset{o_w}{\Longrightarrow}_{\mathcal{M}(O)} (q_f,\ \sigma_f)$
- $o = o_l\ o_w$

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

Case 1: $\sigma'(e) = \texttt{true}$

(a)
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma') \vdash S_l \overset{o'_l}{\Longrightarrow}_{\mathcal{M}(O)} (q'_2,\ \sigma'_1)$
- $(q'_2,\ \texttt{exit}) \xrightarrow{OK} q'_3$
- $(q'_3,\ \sigma'_1) \vdash$ **while** e **do** S_l **done** $\overset{o'_w}{\Longrightarrow}_{\mathcal{M}(O)} (q'_f,\ \sigma'_f)$
- $o' = o'_l\ o'_w$

This follows directly from the local conclusion (1), the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$), and the definition of the transition function of the monitoring automaton.

(b) $o_l = o'_l$, $q_2 = q'_2 = (V_2, w_2)$, $\forall x \notin V_2 : \sigma_1(x) = \sigma'_1(x)$, and $w_2 = w$.
This result is obtained by applying the inductive hypothesis to the derivations of S_l found in the local conclusions (1) and (a).

(c) $q_3 = (V_2, w_3) = q'_3$ with $w_2 = w_3 a$.
This follows directly from the definition of transition function of the monitoring automaton and the local conclusions (1), (a) and (b).

(d) $o_w = o'_w$, $q_f = q'_f$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.
This result is obtained by applying the inductive hypothesis to the derivations of **while** e **do** S_l **done** found in the local conclusions (1) and (a). It is possible to apply the inductive hypothesis because of the local conclusions (c) and (b).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
This follows directly from the local conclusions (1), (a), (b), and (d).

Case 2: $\sigma'(e) = \texttt{false}$

(a) $FV(e) \cap V \neq \emptyset$.
The negation of this property is in contradiction with the case hypothesis, the local conclusion (1), and the global hypothesis $\star_3$.

(b) $w_1 \notin \{\bot\}^\star$.

The local conclusions (a) and (1), and the definition of (T-BRANCH-high) imply that w_1 does not belong to $\{\bot\}^\star$.

(c) $o_l = \epsilon$.

This follows from the local conclusions (1) and (b), and from lemma A.1.2.

(d)
- $(q, \texttt{branch}\, e) \xrightarrow{OK} q_1$
- $(q_1, \texttt{not}\, S_l) \xrightarrow{OK} q'_2$
- $(q'_2, \texttt{exit}) \xrightarrow{OK} q'_f$

This follows directly from the local conclusion (1), the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation ($\text{E}_{\mathcal{M}(O)}$-$\text{WHILE}_{\texttt{false}}$), and the definition of the transition function of the monitoring automaton.

(e) $o' = \epsilon$ and $\sigma'_f = \sigma'$.

This follows directly from the definition of the rule ($\text{E}_{\mathcal{M}(O)}$-$\text{WHILE}_{\texttt{false}}$).

(f) $q_3 = (V_3, w_3) = q'_f$.

This follows from the local conclusions (1) and (d), lemma A.1.3, and the definition of the transition function of the monitoring automaton for the input `exit`..

(g) $V_3 = V \cup \mathit{modified}(S_l)$.

This follows from the local conclusions (f), (d), and (b), and the definition of the transition function of the monitoring automaton.

(h) $o_w = \epsilon$ and $V_3 \subseteq V_f$.

This follows from the local conclusions (1), (a) and (g), and from lemma A.1.5.

(i) $\forall x \notin V_f : \sigma_f(x) = \sigma_1(x)$ and $V_f = V_3$.

Both results follow from lemma A.1.4, the fact that "$\mathit{modified}(\textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}) = \mathit{modified}(S)$". For the first result, from the local conclusion (g) and (h), any variable x, which does not belong to V_f, does not belong to $\mathit{modified}(S_l)$. For the second result, the local conclusions (g) imply that $V_3 = V_3 \cup \mathit{modified}(S_l)$. This result combine with the local conclusion (h) and the conclusions of lemma A.1.4 imply that $V_f = V_3$.

(j) $\forall x \notin V_f : \sigma_1(x) = \sigma'_f(x)$.

For all variable x, if x does not belong to V_f then the local conclusions (i) and (g) imply that x does not belong to V. And so, the global hypothesis $\star_3$ implies that $\sigma(x) = \sigma'(x)$. For all variable x, if x does not belong to V_f then the local conclusions (i) and (g) imply that x does not belong to $\mathit{modified}(S_l)$. Which, in turn, combined with the local conclusion (1) and lemma A.1.4, implies that $\sigma_1(x) = \sigma(x)$. Those two equalities combined with the local conclusion (e) imply the desired result.

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

From the local conclusion (1), $o = o_l\ o_w$. So, it follows from the local conclusions (c), (h), and (e) that $o' = o$. From the local conclusion (f), $q_3 = q'_f$. As done at the beginning of the proof of this lemma, it can be easily proved that $w_f = w_3$. This result combined with the local conclusions (i) and (f) imply that $q_f = q_3 = q'_f$. Finally, from the local conclusions (i) and (j) $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

($\mathbf{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) S = **while** e **do** S_l **done** and:

- $\sigma(e) = \texttt{false}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q_2$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$).

(2) $o = \epsilon$, $q_f = q_3$, $\sigma_f = \sigma$, and $V \subseteq V_f$.
This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$) and the definition of the transition function of the monitoring automaton.

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

Case 1: $\sigma'(e) = \texttt{false}$

(a) $o' = \epsilon$, $q'_f = q_3$, $\sigma'_f = \sigma'$.
This follows from the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation (E$_{\mathcal{M}(O)}$-WHILE$_{\texttt{false}}$), and the definition of the transition function of the monitoring automaton. In this case, the transitions depend only on the initial state q and the expression e.

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
The first two equalities follow from the local conclusions (2) and (a). From the local conclusion (2), all variable x, which does not belong to V_f, does not belong to V. And so, the global hypothesis $\star_3$ implies that $\sigma(x) = \sigma'(x)$. Then, from the local conclusions (2) and (a), $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

Case 2: $\sigma'(e) = \texttt{true}$

(a) $FV(e) \cap V \neq \emptyset$.
The negation of this property is in contradiction with the case hypothesis, the local conclusion (1), and the global hypothesis $\star_3$.

(b)
- $\sigma(e) = \texttt{true}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma') \vdash S_l \overset{o'_l}{\Longrightarrow}_{\mathcal{M}(O)} (q'_2,\ \sigma'_1)$
- $(q'_2,\ \texttt{exit}) \xrightarrow{OK} q'_3$
- $(q'_3,\ \sigma'_1) \vdash$ **while** e **do** S_l **done** $\overset{o'_w}{\Longrightarrow}_{\mathcal{M}(O)} (q'_4,\ \sigma'_f)$
- $o' = o'_l\ o'_w$

This follows from the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation ($\text{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$), the local conclusion (a), and the definition of the transition function of the monitoring automaton.

Let $q_1 = (V_1, w_1)$, $q'_3 = (V'_3, w'_3)$, and $q'_f = (V'_f, w'_f)$.

(c) $w_1 \notin \{\bot\}^\star$.
The local conclusions (a) and (b), and the definition of (T-BRANCH-high) imply that w_1 does not belong to $\{\bot\}^\star$.

(d) $o'_l = \epsilon$.
This follows from the local conclusions (b) and (c), and from lemma A.1.2.

(e) $q'_3 = q_3 = (V_f, w_f)$.
This follows from the local conclusions (1) and (b), lemma A.1.3, and the definition of the rule ($\text{E}_{\mathcal{M}(O)}$-WHILE$_{\texttt{true}}$).

(f) $V_f = V \cup \textit{modified}(S_l)$.
This follows from the local conclusions (c), (1), and (2), and the definition of the transition function of the monitoring automaton.

(g) $\forall x \notin V_f : \sigma_f(x) = \sigma'_1(x)$.
For all variable x, if x does not belong to V_f then the local conclusion (f) implies that x does not belong to V. And so, the global hypothesis $\star_3$ implies that $\sigma(x) = \sigma'(x)$. For all variable x, if x does not belong to V_f then the local conclusion (f) implies that x does not belong to $\textit{modified}(S_l)$. Which, in turn, combined with the local conclusion (b) and lemma A.1.4, implies that $\sigma'_f(x) = \sigma'(x)$. Those two equalities combined with the local conclusion (2) imply the desired result.

(h) $o'_w = \epsilon$ and $V'_3 \subseteq V'_f$.
This follows from the local conclusions (b) and (a), and from lemma A.1.5.

(i) $\forall x \notin V_f : \sigma'_f(x) = \sigma'_1(x)$ and $V'_f = V'_3$.
Both results follow from lemma A.1.4, the fact that "$\textit{modified}(\textbf{while } e \textbf{ do } S \textbf{ done}) = \textit{modified}(S)$". For the first result, from the local conclusion (f), any variable x, which does not belong to V_f, does not belong to $\textit{modified}(S_l)$. For the second result, the local conclusions (e) and (f) imply that $V'_3 = V'_3 \cup \textit{modified}(S_l)$. This result combine with the local conclusion (h) and the conclusions of lemma A.1.4 imply that $V'_f = V'_3$.

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.
From the local conclusion (b), $o' = o'_l\, o'_w$. So, it follows from the local conclusions (2), (d), and (h) that $o' = o$. From the local conclusion (e), $q_f = q'_3$. As done at the beginning of the proof of this lemma, it can be easily proved that $w'_f = w'_3$. This result combined with the local conclusion (i) implies that $q'_f = q'_3 = q_f$. Finally, from the local conclusions (i) and (g) $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

($\text{E}_{\mathcal{M}(O)}$-SEQ) then we can conclude that :

(1) $S = S_1 \,;\, S_2$, $(q,\sigma) \vdash S_1 \overset{o_1}{\Longrightarrow}_{\mathcal{M}(O)} (q_1,\sigma_1)$, $(q_1,\sigma_1) \vdash S_2 \overset{o_2}{\Longrightarrow}_{\mathcal{M}(O)} (q_f,\sigma_f)$, and $o = o_1\ o_2$.
This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-SEQ).

(2) $(q,\sigma') \vdash S_1 \overset{o'_1}{\Longrightarrow}_{\mathcal{M}(O)} (q'_1,\sigma'_1)$, $(q'_1,\sigma'_1) \vdash S_2 \overset{o'_2}{\Longrightarrow}_{\mathcal{M}(O)} (q'_f,\sigma'_f)$, and $o' = o'_1\ o'_2$.
This follows from the global hypothesis $\star_2$, the local conclusion (2), and the definition of the rule (E$_{\mathcal{M}(O)}$-SEQ).

(3) $o_1 = o'_1$, $q_1 = q'_1 = (V_1, w_1)$, and $\forall x \notin V_1 : \sigma_1(x) = \sigma'_1(x)$
This result can be obtained from the inductive hypothesis using the evaluations of S_1 in the local conclusions (1) and (2).

(4) $o_2 = o'_2$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
This result can be obtained from the inductive hypothesis using the evaluations of S_2 in the local conclusions (1) and (2), and the fact that $\forall x \notin V_1 : \sigma_1(x) = \sigma'_1(x)$ (from the local conclusion (3)).

(•) $o = o'$, $q_f = q'_f = (V_f, w_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
This follows from the global hypothesis the local conclusions (1), (2), (3), and (4).

A.2 Transparency

Lemma A.2.1 (Simple Security). *For all typing environment γ, expression e, and command type τ cmd, if $\gamma \vdash e : \tau$ then, for all variable x belonging to $FV(e)$, $\gamma(x) \leq \tau$.*

Proof. This lemma is just a reformulation of the (Simple Security) lemma appearing in (Volpano et al., 1996).

Lemma A.2.2 (Confinement). *For all typing environment γ, command C, and command type τ cmd, if $\gamma \vdash C : \tau$ cmd then, for all variable x belonging to* modified*(C), $\tau \leq \gamma(x)$.*

Proof. This lemma is just a reformulation of the (Confinement) lemma appearing in (Volpano et al., 1996).

Lemma A.2.3 (Confined Outputs). *For all command C, typing environment γ, and value stores σ and σ' if:*

$\star_1$ $\gamma \vdash C : H$ *cmd,*

$\star_2$ $\sigma \vdash C \overset{o}{\Rightarrow}_O \sigma'$,

then $o = \epsilon$.

Proof. The proof goes by contradiction. If $o \neq \epsilon$ then C contains a command **output** e. If it does, because of the typing rules, C must be typed L cmd. This is in contradiction with the hypothesis $\star_1$.

Lemma A.2.4 (Helper 1). *For all command C, automaton states (V, w) and (V', w), and value stores σ and σ', if $((V,w),\sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{\mathcal{M}(O)} ((V',w),\sigma')$ then, for all w' of which w is a prefix, $((V,w'),\sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{\mathcal{M}(O)} ((V',w'),\sigma')$.*

Proof. The monitoring mechanism does not influence the final value store obtained after the execution. So, changing the automaton state does not change the final value store. If the command C does not contain output statements, then any execution of C (whatever the automaton state) outputs nothing. If the command C contains an output statement executed, then it implies that w does not belong to $\{\bot\}^\star$ (otherwise something would be output). The behavior of the automaton with regard to output statements is the same for any w not belonging to $\{\bot\}^\star$. So, as w (which does not belong to $\{\bot\}^\star$) is a prefix of w', the behavior of the automaton is the same with w or w'. So the output is identical.

Lemma A.2.5 (Monitoring Automaton Precision). *For all command*[1] *C, typing environment γ, command type τ cmd, value stores σ and σ', and automaton state (V, w), if:*

$\star_1$ $\forall x \in V, \gamma(x) = H$,

$\star_2$ $w \notin \{\bot\}^\star \Rightarrow \tau = H$,

$\star_3$ $\gamma \vdash C : \tau\ cmd$,

$\star_4$ $\sigma \vdash C \overset{o}{\Rightarrow}_O \sigma'$,

then there exists an automaton state (V', w) such that:

- $((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$,
- $\forall x \in V', \gamma(x) = H$.

Proof. The proof is done by induction on the size of the derivation tree of "$\sigma \vdash C \overset{o}{\Rightarrow}_O \sigma'$". Assume the lemma holds for any sub-derivation tree. If the last semantic rule used is:

(E$_O$-ASSIGN) then we can conclude that :

(1)
- C is "x **:=** e",
- $\sigma \vdash C \overset{\epsilon}{\Rightarrow}_O \sigma[x \mapsto \sigma(e)]$.

This follows directly from the rule (E$_O$-ASSIGN).

(2) $((V, w), \sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma[x \mapsto \sigma(e)])$.

This follows directly from the local conclusion (1) and the rules (T-ASSIGN-sec), (T-ASSIGN-pub), (E$_{\mathcal{M}(O)}$-OK), and (E$_O$-ASSIGN).

(3) $\forall y \in V', \gamma(y) = H$.

Case 1: $w \in \{\bot\}^\star$ and $FV(e) \cap V = \emptyset$.

(a) $V' \subseteq V$.

This follows directly from the rule (T-ASSIGN-pub).

($\circ$) $\forall y \in V', \gamma(y) = H$.

This follows from the local conclusion (a) and the global hypothesis $\star_1$.

[1]We use "command" and "statements" as synonyms

Case 2: $w \notin \{\perp\}^\star$ or $FV(e) \cap V \neq \emptyset$.

(a) $V' = V \cup \{x\}$.

This follows directly from the rule (T-ASSIGN-sec).

(b) $\gamma(x) = H$.

If $w \notin \{\perp\}^\star$ then, because of the global hypothesis $\star_2$, $\tau = H$; and so, because of the global hypotheses $\star_3$ and the typing rule (T-ASSIGN) (which is the only one applying to "x **:=** e"), $\gamma(x) = H$. If $FV(e) \cap V \neq \emptyset$ then there exists a variable y in $FV(e)$ such that $y \in V$. The global hypothesis $\star_1$ implies that $\gamma(y) = H$. Then, the global hypothesis $\star_3$ and lemma A.2.1 imply that $\tau = H$; and so, because of the typing rule (T-ASSIGN) (which is the only one applying to "x **:=** e"), $\gamma(x) = H$.

(○) $\forall y \in V', \gamma(y) = H$.

This follows from the global hypothesis $\star_1$ and the local conclusions (a) and (b).

(•) there exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".

If follows directly from the local conclusions (2) and (3).

(E$_O$-SKIP) then we can conclude that :

(1)
- C is "**skip**",
- $\sigma \vdash C \overset{\epsilon}{\Rightarrow}_O \sigma$.

This follows directly from the rule (E$_O$-SKIP).

(2) $((V, w), \sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{\mathcal{M}(O)} ((V, w), \sigma)$.

This follows directly from the local conclusion (1) and the rules (T-SKIP), (E$_{\mathcal{M}(O)}$-OK), and (E$_O$-SKIP).

(•) there exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".

If follows directly from the local conclusion (2) and from the global hypothesis $\star_1$.

(E$_O$-PRINT) then we can conclude that :

(1)
- C is "**output** e",
- $\sigma \vdash C \xRightarrow{\sigma(e)}_O \sigma$.

This follows directly from the rule (E$_O$-PRINT).

(2) $w \in \{\perp\}^\star$ and $FV(e) \cap V = \emptyset$.

The global hypothesis $\star_3$, the local conclusion (1) and the typing rule (T-PRINT) (the only one applying to "**output** e") imply that $\tau = L$. Hence, the global hypothesis $\star_2$ implies $w \in \{\perp\}^\star$. The typing rule (T-PRINT) also implies that "$\gamma \vdash e : \tau$". Then lemma A.2.1 implies that, for all y in $FV(e)$, $\gamma(y) = L$. Hence, because of the global hypothesis $\star_1$, $FV(e) \cap V = \emptyset$.

(3) $((V,w),\sigma) \vdash C \overset{\sigma(e)}{\Longrightarrow}_{\mathcal{M}(O)} ((V,w),\sigma)$.

This follows from the transition (T-PRINT-ok) (which is the only one applying because of the local conclusions (1) and (2)) and the rules ($E_{\mathcal{M}(O)}$-OK) and (E_O-PRINT).

(•) there exists an automaton state (V', w) such that "$((V,w),\sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V',w),\sigma')$" and "$\forall x \in V', \gamma(x) = H$".

If follows directly from the local conclusion (3) and from the global hypothesis $\star_1$.

(E_O-IF) then we can conclude that :

(1)
- C is "**if** e **then** C_{true} **else** C_{false} **end**",
- $\sigma(e) = v$
- $\sigma \vdash C_v \overset{o}{\Rightarrow}_O \sigma'$.

This follows directly from the rule (E_O-IF).

(2) There exists a type τ' such that:
- $\tau \le \tau'$,
- $\gamma \vdash e : \tau'$,
- $\gamma \vdash C_{true} : \tau'$ cmd,
- $\gamma \vdash C_{false} : \tau'$ cmd.

This follows directly from the local conclusion (1), the global hypothesis $\star_3$ and the only typing rule applying to "**if** e **then** C_{true} **else** C_{false} **end**" (T-IF).

(•) There exists an automaton state (V', w) such that "$((V,w),\sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V',w),\sigma')$" and "$\forall x \in V', \gamma(x) = H$".

Case 1: $FV(e) \cap V \neq \emptyset$.

(a) $\tau' = H$.

From the local conclusion (2), $\gamma \vdash e : \tau'$. From the case hypothesis and the global hypothesis $\star_1$, there exists a variable y in $FV(e)$ such that $\gamma(y) = H$. Using lemma A.2.1, those two properties imply that $\tau' = H$.

(b) There exists an automaton state $(V', w\top)$ such that "$((V, w\top),\sigma) \vdash C_v \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w\top),\sigma')$", and "$\forall x \in V', \gamma(x) = H$".

This follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and the local conclusions (a), (2) and (1).

(c) $((V,w),\sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V' \cup \mathit{modified}(C_{\neg v}), w),\sigma')$.

This follows from the only rule applying to "**if** e **then** C_{true} **else** C_{false} **end**" ($E_{\mathcal{M}(O)}$-IF), the local conclusion (b), and the transition rules (T-BRANCH-high) (which is the only one applying to `branch` e because of the case hypothesis), (T-NOT-high), and (T-EXIT).

(d) $\forall x \in V' \cup \mathit{modified}(C_{\neg v}), \gamma(x) = H$.

As, from the local conclusion (a) $\tau' = H$ and from the local conclusion (2) $\gamma \vdash C_{\neg v}$:

τ' cmd, lemma A.2.2 implies that, for all x belonging to *modified*$(C_{\neg v})$, $\gamma(x) = H$. This result, combined with the local conclusion (b), implies the desired result.

(∘) There exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".

This follows directly from the local conclusions (c) and (d).

Case 2: $FV(e) \cap V = \emptyset$.

(a) There exists an automaton state $(V', w\bot)$ such that "$((V, w\bot), \sigma) \vdash C_v \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w\bot), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".

This follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and $\star_2$, and the local conclusions (2) and (1).

(b) $((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V'', w), \sigma')$ with $w \notin \{\bot\}^\star \Rightarrow (V'' = V' \cup \textit{modified}(C_{\neg v}))$ and $w \in \{\bot\}^\star \Rightarrow V'' = V'$.

This follows from the only rule applying to "**if** e **then** C_{true} **else** C_{false} **end**" ($E_{\mathcal{M}(O)}$-IF), the local conclusion (a), and the transition rules (T-BRANCH-low) (which is the only one applying to `branch`e because of the case hypothesis), (T-NOT-high), (T-NOT-low), and (T-EXIT).

(c) $\forall x \in V'', \gamma(x) = H$.

If $w \in \{\bot\}^\star$ then $V'' = V'$ and the desired property follows directly from the global hypothesis $\star_1$. If $w \notin \{\bot\}^\star$ then the global hypothesis $\star_2$ implies $\tau = H$. As, from the local conclusion (2) $\tau \leq \tau'$ and $\gamma \vdash C_{\neg v} : \tau'$ cmd, lemma A.2.2 implies that, for all x belonging to *modified*$(C_{\neg v})$, $\gamma(x) = H$. This result, combined with the local conclusions (b) and (a), implies that if $w \notin \{\bot\}^\star$ then $\forall x \in V'', \gamma(x) = H$.

(∘) There exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$", and "$\forall x \in V', \gamma(x) = H$".

This follows directly from the local conclusions (b) and (c).

(E_O-WHILE$_{true}$) then we can conclude that :

(1)
- C is "**while** e **do** C_l **done**",
- $\sigma(e) = \texttt{true}$
- $\sigma \vdash C_l$; **while** e **do** C_l **done** $\overset{o}{\Rightarrow}_O \sigma'$.

This follows directly from the rule (E_O-WHILE$_{true}$).

(2) There exists a type τ' such that:

- $\tau \leq \tau'$,
- $\gamma \vdash e : \tau'$,
- $\gamma \vdash C_l : \tau'$ cmd.

This follows directly from the local conclusion (1), the global hypothesis $\star_3$ and the only typing rule applying to "**while** e **do** C_l **done**" (T-WHILE).

(3) $\gamma \vdash C_l$; **while** e **do** C_l **done** : τ' cmd.

From the local conclusion (2) and the typing rule (T-WHILE), $\gamma \vdash$ **while** e **do** C_l **done** : τ' cmd. Hence, as from the local conclusion (2) $\gamma \vdash C_l : \tau'$ cmd, the typing rule (T-SEQ) implies the desired result.

(4) There exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C_l$; **while** e **do** C_l **done** $\overset{o}{\Rightarrow}_{\mathcal{M}(O)}$ $((V', w), \sigma')$", and "$\forall x \in V', \gamma(x) = H$".

This follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and $\star_2$, and the local conclusions (2), (3) and (1).

(5) There exist an automaton state (V_l, w) and a value store σ_l such that:

- $((V, w), \sigma) \vdash C_l \overset{o_l}{\Longrightarrow}_{\mathcal{M}(O)} ((V_l, w), \sigma_l)$,
- $((V_l, w), \sigma_l) \vdash$ **while** e **do** C_l **done** $\overset{o_w}{\Longrightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$,
- $o = o_l\, o_w$.

This follows from the local conclusion (4), the only semantic rule applying to C_l ; **while** e **do** C_l **done** (E$_{\mathcal{M}(O)}$-SEQ), and lemma A.1.1.

(6) "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$".

Case 1: $\tau' = H$.

(a) $o_l = \epsilon$.

This follows from the case hypothesis and lemma A.2.3 applied to the local conclusions (2) and (5).

(b) For all w' such that w is a prefix, $((V, w'), \sigma) \vdash C_l \overset{o_l}{\Longrightarrow}_{\mathcal{M}(O)} ((V_l, w'), \sigma_l)$.

This follows from lemma A.2.4 and the local conclusions (a) and (5).

($\circ$) "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$".

This follows from the semantic rule (E$_{\mathcal{M}(O)}$-WHILE$_{true}$), the transition (T-BRANCH-low) and (T-BRANCH-high), the local conclusion (b), the transition (T-EXIT), the local conclusion (5).

Case 2: $\tau' \neq H$.

(a) $w \in \{\bot\}^\star$ and $FV(e) \cap V = \emptyset$.

As $\tau \leq \tau'$ (from the local conclusion (2)), the case hypothesis and the global hypothesis $\star_2$ imply $w \in \{\bot\}^\star$. As $\gamma \vdash e : \tau'$ (from the local conclusion (2)), the case hypothesis, lemma A.2.1, and the global hypothesis $\star_1$ imply $FV(e) \cap V = \emptyset$.

(b) $((V, w\bot), \sigma) \vdash C_l \overset{o_l}{\Longrightarrow}_{\mathcal{M}(O)} ((V_l, w\bot), \sigma_l)$.

This follows from the local conclusion (5) and the fact that $w\bot$ belongs to $\{\bot\}^\star$ (because of the local conclusion (a)).

($\circ$) "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$".

This follows from the semantic rule (E$_{\mathcal{M}(O)}$-WHILE$_{true}$), the transition (T-BRANCH-low) (which is the only one applying because of the local conclusion (a)), the local conclusion (b), the transition (T-EXIT), the local conclusion (5).

(•) There exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$", and "$\forall x \in V', \gamma(x) = H$".
This follows directly from the local conclusions (6) and (4).

($\mathbf{E}_O$-WHILE$_{false}$) then we can conclude that :

(1)
- C is "**while** e **do** C_l **done**",
- $\sigma(e) = \texttt{false}$
- $o = \epsilon$
- $\sigma' = \sigma$.

This follows directly from the rule (E_O-WHILE$_{false}$).

(2) There exists a type τ' such that:
- $\tau \leq \tau'$,
- $\gamma \vdash e : \tau'$,
- $\gamma \vdash C_l : \tau'$ cmd.

This follows directly from the local conclusion (1), the global hypothesis $\star_3$ and the only typing rule applying to "**while** e **do** C_l **done**" (T-WHILE).

(3) $((V, w), \sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma)$ with $(w \notin \{\bot\}^\star \vee FV(e) \cap V \neq \emptyset) \Rightarrow (V' = V \cup \mathit{modified}(C_l))$ and $(w \in \{\bot\}^\star \wedge FV(e) \cap V = \emptyset) \Rightarrow V' = V$.
This follows from the only rule applying to "**while** e **do** C_l **done**" whenever $\sigma(e) = \texttt{false}$ ($\mathrm{E}_{\mathcal{M}(O)}$-WHILE$_{false}$) and the transition rules (T-BRANCH-low), (T-BRANCH-high), (T-NOT-high), (T-NOT-low), and (T-EXIT).

(4) $(w \notin \{\bot\}^\star \vee FV(e) \cap V \neq \emptyset) \Rightarrow (\forall x \in \mathit{modified}(C_l),\ \gamma(x) = H)$.
If $w \notin \{\bot\}^\star$ then the global hypothesis $\star_2$ imply that $\tau = H$. Hence, because of the local conclusion (2), $\tau' = H$. If $FV(e) \cap V \neq \emptyset$ then, from the global hypothesis $\star_1$, there exists a variable y in $FV(e)$ such that $\gamma(y) = H$. Using lemma A.2.1, as $\gamma \vdash e : \tau'$ (from the local conclusion (1)), this implies that $\tau' = H$. As $\tau' = H$ in both cases ($w \notin \{\bot\}^\star$ and $FV(e) \cap V \neq \emptyset$), the local conclusion (2) ($\gamma \vdash C_l : \tau'$ cmd) and lemma A.2.2 imply the desired result.

(•) There exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$", and "$\forall x \in V', \gamma(x) = H$".
This follows directly from the local conclusions (3) and (4) and from the global hypothesis $\star_1$.

($\mathbf{E}_O$-SEQ) then we can conclude that :

(1) There exists a value store σ'_1 such that:
- C is "C_1 ; C_2",
- $\sigma \vdash C_1 \overset{o_1}{\Longrightarrow}_O \sigma'_1$,
- $\sigma'_1 \vdash C_2 \overset{o_2}{\Longrightarrow}_O \sigma'$,
- $o = o_1\ o_2$.

This follows directly from the rule (E_O-SEQ).

(2)
- $\gamma \vdash C_1 : \tau$ cmd,
- $\gamma \vdash C_2 : \tau$ cmd.

This follows directly from the global hypothesis $\star_3$, the local conclusion (1), and the only typing rule applying to "$C_1 ; C_2$" (T-SEQ).

(3) there exists an automaton state (V'_1, w) such that "$((V, w), \sigma) \vdash C_1 \overset{o_1}{\Longrightarrow}_{\mathcal{M}(O)} ((V'_1, w), \sigma'_1)$" and "$\forall x \in V'_1, \gamma(x) = H$".
This follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and $\star_2$, and the local conclusions (2) and (1).

(4) there exists an automaton state (V'_2, w) such that "$((V'_1, w), \sigma'_1) \vdash C_2 \overset{o_2}{\Longrightarrow}_{\mathcal{M}(O)} ((V'_2, w), \sigma'_2)$", and "$\forall x \in V'_2, \gamma(x) = H$".
This follows directly from the inductive hypothesis, the local conclusion (3), the global hypothesis $\star_2$, and the local conclusions (2) and (1).

(5) $((V, w), \sigma) \vdash C \overset{o_1 o_2}{\Longrightarrow}_{\mathcal{M}(O)} ((V'_2, w), \sigma'_2)$.
This follows from the rule ($E_{\mathcal{M}(O)}$-SEQ) and the local conclusions (3) and (4).

(•) There exists an automaton state (V', w) such that "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{\mathcal{M}(O)} ((V', w), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".
This follows directly from the local conclusions (5) and (4).

Appendix B

Proofs of Chapter 4's Theorems

By convention, in order to simplify understanding of the proofs and reduce their size, we keep superscripts and subscripts of program states consistent whatever the level of expansion of the program state notation. Therefore, for any sup and any i, let ζ_i^{sup} be equal to $(\varsigma_i^{sup}, q_i^{sup})$, itself equal to $((\sigma_i^{sup}, \lambda_i^{sup}), (V_i^{sup}, W_i^{sup}, L_i^{sup}, w_i^{sup}))$.

B.1 Definitions

Definition B.1.1 (Owned locks).
For all monitored statements S, $OLocks(S)$ is the set of all locks currently owned by the thread executing S.

$$
\begin{aligned}
&OLocks(\emptyset) = \emptyset \\
OLocks(P) = \emptyset \quad &\begin{cases} OLocks(x := e) = \emptyset \\ OLocks(\textbf{output } e) = \emptyset \\ OLocks(\textbf{skip}) = \emptyset \\ OLocks(\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ end}) = OLocks(P_1) \cup OLocks(P_2) \\ OLocks(\textbf{while } e \textbf{ do } P \textbf{ done}) = OLocks(P) \\ OLocks(\textbf{with } \bar{x} \textbf{ when } e \textbf{ do } P \textbf{ done}) = OLocks(P) \end{cases} \\
&OLocks(S_1 \ ; \ P_2) = OLocks(S_1) \cup OLocks(P_2) \\
&OLocks(\odot\bar{x}[S]) = \bar{x} \cup OLocks(S) \\
&OLocks(\otimes(P_1, P_2)[S]) = OLocks(S)
\end{aligned}
$$

Definition B.1.2 (Strip Branching Data Function).
strip() takes into parameter a monitored statement (Figure 4.8) and returns an execution statement (Fig-

ure 4.3).

$$OLocks(P) = P \quad \begin{aligned} &strip(\emptyset) = \emptyset \\ &\begin{cases} strip(x := e) = x := e \\ strip(\textbf{output } e) = \textbf{output } e \\ strip(\textbf{skip}) = \textbf{skip} \\ strip(\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ end}) = \textbf{if } e \textbf{ then } strip(P_1) \textbf{ else } strip(P_2) \textbf{ end} \\ strip(\textbf{while } e \textbf{ do } P \textbf{ done}) = \textbf{while } e \textbf{ do } strip(P) \textbf{ done} \\ strip(\textbf{with } \bar{x} \textbf{ when } e \textbf{ do } P \textbf{ done}) = \textbf{with } \bar{x} \textbf{ when } e \textbf{ do } strip(P) \textbf{ done} \end{cases} \\ &strip(S_1 \ ; \ P_2) = strip(S_1) \ ; \ strip(P_2) \\ &strip(\odot\bar{x}[S]) = \odot\bar{x}[strip(S)] \\ &strip(\otimes(P_1, P_2)[S]) = strip(S) \end{aligned}$$

Definition B.1.3 (Low Equivalent Thread Monitoring States).
Two monitored execution states $((\sigma_1, \lambda_1), q_1)$ and $((\sigma_2, \lambda_2), q_2)$ are low equivalent, written $((\sigma_1, \lambda_1), q_1) \cong ((\sigma_2, \lambda_2), q_2)$, if and only if:

- $q_1 = q_2 = (V, W, L, w)$ *and* $w \in \{\bot\}^\star$,
- $\forall x \notin V : \sigma_1(x) = \sigma_2(x)$,
- $\forall x \notin L : x \in \lambda_1 \Leftrightarrow x \in \lambda_2$.

By extension, two monitored execution states — $(\varsigma_1, \mathsf{Q}_1)$ and $(\varsigma_2, \mathsf{Q}_2)$ — of a pool of threads Θ are considered low equivalent, written $(\varsigma_1, \mathsf{Q}_1) \cong (\varsigma_2, \mathsf{Q}_2)$, if and only if, for all ι in the domain of Θ, $(\varsigma_1, \text{extractState}(\mathsf{Q}_1, \iota)) \cong (\varsigma_2, \text{extractState}(\mathsf{Q}_2, \iota))$.

Definition B.1.4 ((State, Statement) Pair Coherency).
A monitored execution state ζ — equals to $((\sigma, \lambda), (V, W, L, w))$ — is coherent with a monitored statement S, written $\zeta \vdash S$, if and only if:

- $\forall x : x \in W \Rightarrow x \in V$,
- $S = \otimes(P^e, P^u)[S^b] \implies modified(S^b) \subseteq modified(P^e)$,
- $S = \otimes(P^e, P^u)[S^b] \implies (NLocks(S^b) \cup OLocks(S^b)) \subseteq NLocks(P^e)$,
- $S = \otimes(P^e, P^u)[S^b] \implies (\neg stoppable(P^e) \Rightarrow \neg stoppable(S^b))$,
- $(S = \otimes(P^e, P^u)[S^b] \ \wedge \ \exists\, w' \text{ s.t. } w = \top w') \implies modified(P^e) \cup modified(P^u) \subseteq W$,
- $(S = \otimes(P^e, P^u)[S^b] \ \wedge \ \exists\, w' \text{ s.t. } w = \top w') \implies NLocks(P^e) \cup NLocks(P^u) \subseteq L$,
- $(S = \otimes(P^e, P^u)[S^b] \ \wedge \ \exists\, w' \text{ s.t. } w = \top w') \implies (\forall l \in (NLocks(S^b) \setminus OLocks(S^b)) : l \notin \lambda)$,
- $S = \odot\bar{x}[S'] \implies ((\sigma, \lambda \setminus \bar{x}), (V, W, L, w)) \vdash S'$,

- $(S = \otimes(P^e, P^u)[S'] \quad \wedge \quad \exists\, w' \text{ s.t. } w = aw') \quad \Longrightarrow \quad (\varsigma, (V, W, L, w')) \vdash S'$,

By extension, a monitored execution state — $(\varsigma, \mathtt{Q})$ — is considered coherent with a pool of threads Θ , written $(\varsigma, \mathtt{Q}) \vdash \Theta$, if and only if, for all ι in the domain of Θ, $(\varsigma, \text{extractState}(\mathtt{Q}, \iota)) \vdash \Theta(\iota)$.

B.1.1 Evaluation context.

The proof makes intensive use of *evaluation contexts* (Nielson et al., 2005, Sect. 5.5.1).

Definition B.1.5 (Evaluation context).
An evaluation context is a monitored statement containing a unique hole *which is written* $\square$. *Evaluation contexts are described by the following grammar:*

$$\begin{array}{rcl} \langle \textit{eval-context} \rangle & ::= & \square \quad | \quad \langle \textit{eval-context} \rangle \;;\; \langle \textit{prog} \rangle \\ & | & \odot\langle \textit{ident-set} \rangle[\langle \textit{eval-context} \rangle] \\ & | & \otimes(\langle \textit{prog} \rangle, \langle \textit{prog} \rangle)[\langle \textit{eval-context} \rangle] \end{array}$$

Definition B.1.6 (Evaluation context as a function).
Every evaluation context E is overloaded such that it is also a function mapping monitored statements to monitored statements. This function returns the monitored statements corresponding to E where $\square$ is replaced by the function's argument. $E(S)$ is sometimes written $E[S]$. Formally, this function is defined as follows:

$$E(S) = \begin{cases} S & \text{if and only if } E = \square \\ E'(S) \;;\; P & \text{if and only if } E = E' \;;\; P \\ \odot\bar{x}[E'(S)] & \text{if and only if } E = \odot\bar{x}[E'] \\ \otimes(P_1, P_2)[E'(S)] & \text{if and only if } E = \otimes(P_1, P_2)[E'] \end{cases}$$

Definition B.1.7 (Context word of an evaluation context).
During any evaluation, a context word w evolves with the monitored statement S under evaluation. This monitored statement can be expressed using an evaluation context E and another monitored statement S' as follows: $S = E(S')$. $(w\downarrow_E)$ is the prefix of w which has been created in the same time as the evaluation context E. Formally, this function is defined as follows:

$$(w\downarrow_E) = \begin{cases} \epsilon & \text{if } E = \square \\ (w\downarrow_{E'}) & \text{if } E = E' \;;\; P \\ (w\downarrow_{E'}) & \text{if } E = \odot\bar{x}[E'] \\ a\,(w'\downarrow_{E'}) & \text{if } E = \otimes(P^e, P^u)[E'] \text{ and } w = aw' \\ \epsilon & \text{if } w = \epsilon \end{cases}$$

Definition B.1.8 (Inverse of an evaluation context).
The inverse of an evaluation context function $E()$ is the partial function written $E^{-1}()$ such that, for all statements S, $E^{-1}(E(S)) = S$.

B.1.2 Weak Bisimulation Candidate

The proofs rely heavily on a bisimilarity relation between program states. This relation is an invariant preserved during the execution of two noninterfering executions started with the same public inputs.

Definition B.1.9 (Weak Bisimilarity of Execution Stages).
An execution stage is a coherent pair composed of a monitored execution state and a monitored statement. The execution stage $(\zeta_1 \vdash S_1)$ *is bisimilar to the execution stage* $(\zeta_2 \vdash S_2)$*, written* $(\zeta_1 \vdash S_1) \sim (\zeta_2 \vdash S_2)$*, if and only if they are coherent —* $\zeta_1 \vdash S_1$ *and* $\zeta_2 \vdash S_2$*, which is true by definition of execution stages — and one, at least, of the following is true:*

${}^{B.1.9}\triangleleft_1$ $S_1 = S_2$ *and* $\zeta_1 \cong \zeta_2$*,*

${}^{B.1.9}\triangleleft_2$ *there exist an evaluation context E, four programs* P^e_1, P^u_1, P^e_2, P^u_2 *and two monitored statements* S^b_1 *and* S^b_2 *such that all the following is true:*

(a) $S_1 = E(\otimes(P^e_1, P^u_1)[S^b_1])$ *and* $S_2 = E(\otimes(P^e_2, P^u_2)[S^b_2])$*,*

(b) $V_1 = V_2$ *,* $W_1 = W_2$ *,* $L_1 = L_2$ *and* $(w_1 {\downarrow_E}) = (w_2 {\downarrow_E})$*,*

(c) $\forall i \in \{1, 2\} : \exists w^s_i$ *s.t.* $w_i = (w_i {\downarrow_E}) \top w^s_i$*,*

(d) $\forall x \notin V_1 : \sigma_1(x) = \sigma_2(x)$*,*

(e) $\forall l \notin L_1 : l \in \lambda_1 \Leftrightarrow l \in \lambda_2$*,*

(f) $(\neg stoppable(P^e_1) \wedge \neg stoppable(P^u_1)) \Leftrightarrow (\neg stoppable(P^e_2) \wedge \neg stoppable(P^u_2))$.

By extension, two execution states — $(\varsigma_1, \mathsf{Q}_1)$ and $(\varsigma_2, \mathsf{Q}_2)$ — and two pool of threads — Θ_1 and Θ_2 — are considered bisimilar, written $((\varsigma_1, \mathsf{Q}_1) \vdash \Theta_1) \sim ((\varsigma_2, \mathsf{Q}_2) \vdash \Theta_2)$, if and only if, for all ι in the domain of Θ, $((\varsigma_1, \text{extractState}(\mathsf{Q}_1, \iota)) \vdash \Theta_1(\iota)) \sim ((\varsigma_2, \text{extractState}(\mathsf{Q}_2, \iota)) \vdash \Theta_2(\iota))$.

B.2 Soundness

Lemma B.2.1 (Evaluation Context Generic Evaluation Rule).
For all evaluation contexts E, monitored statements S and S′, and monitored execution states $((\sigma, \lambda), q)$ *and* $((\sigma', \lambda'), q')$*,* $E(S)$ *in state* $((\sigma, \lambda), q)$ *evaluates to* $E(S')$ *yielding state* $((\sigma', \lambda' \cup OLocks(E(\emptyset))), q')$ *and output o in n steps if and only if S in state* $((\sigma, \lambda \setminus OLocks(E(\emptyset))), q)$ *evaluates to S′ yielding state* $((\sigma', \lambda'), q')$ *and output o in n steps.*

Proof. This follows, by induction on E, from the definitions of evaluation context (Definition B.1.5 and B.1.6) and the semantic rules applying to $E(S)$. $E'(S)$; P takes one evaluation step if and only if $E'(S)$ takes one evaluation step in the same state. From the rule ($\text{E}_{\mathcal{M}(O)}$-LOCKED$_{\text{step}}$), $\odot\bar{x}[E'(S)]$ takes one evaluation step in state $((\sigma, \lambda), q)$ yielding state $((\sigma', \lambda' \cup OLocks(E(\emptyset))), q')$ if and only if $E'(S)$ takes one evaluation step in state $((\sigma, \lambda \setminus OLocks(\odot\bar{x}[\emptyset])), q)$ yielding state $((\sigma', \lambda'), q')$. $\otimes(P_1, P_2)[E'(S)]$ takes one evaluation step if and only if $E'(S)$ takes one evaluation step in the same state.

Lemma B.2.2 (Preservation of the Evaluation Context's Branching Context).
For all evaluation contexts E, monitored statements S and S', and monitored execution states ζ and ζ':

$\star_1$ $\langle\!\langle \zeta \vdash E(S) \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!\langle \zeta' \vdash E(S') \rangle\!\rangle$

implies that $(w \downarrow_E) = (w' \downarrow_E)$.

Proof. The proof follows directly from Definitions B.1.6 and B.1.7 and the semantic rules of Figure 4.9 going by structural induction on E.

Lemma B.2.3 (Coherency preservation).
For all monitored statements S and S', and monitored execution states ζ and ζ' such that:

$\star_1$ $\zeta \vdash S$,

$\star_2$ $\langle\!\langle \zeta \vdash S \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!\langle \zeta' \vdash S' \rangle\!\rangle$,

it is true that:

- $\zeta' \vdash S'$.

This lemma states that coherency of monitored execution states is an execution invariant.

Proof. The proof of this lemma goes by structural induction on S. If S is an atomic action, only the first property ($\forall x : x \in W \Rightarrow x \in V$) can apply. It is then obvious from the semantic rules of Figure 4.9 and the transition function of Figure 4.7 that $\zeta' \vdash S'$. For *branched statements*, the properties are obvious after deducing from the semantic rules that initially such a statement is always of the form $\otimes(P^e, P^u)[P^e]$. For *locked statements*, it follows from the semantic rule for such statements which temporarily releases the locks owned by a thread before executing one step for this thread.

Lemma B.2.4 (Bisimilarity preservation).
For all monitored statements S_1 and S_2, and monitored execution states ζ_1, ζ_1', ζ_2 and ζ_2' such that:

$\star_1$ $(\zeta_1 \vdash S_1) \sim (\zeta_2 \vdash S_2)$,

$\star_2$ $V_1' = V_2'$, $W_1' = W_2'$, $L_1' = L_2'$, $w_1' = w_1$ *and* $w_2' = w_2$,

$\star_3$ $\forall x \notin V_1' : \sigma_1'(x) = \sigma_2'(x)$,

$\star_4$ $\forall l \notin L_1' : l \in \lambda_1' \Leftrightarrow l \in \lambda_2'$,

it is true that $(\zeta_1' \vdash S_1) \sim (\zeta_2' \vdash S_2)$.

Proof. This lemma is straightforward. It follows directly from Definition B.1.9. Every property in this definition involving the execution state is either one of $\star_2$, $\star_3$,$\star_4$ or follows directly from one of them.

Lemma B.2.5 (High Context Evaluation Step).
For all evaluation contexts E, monitored statements S and S′, output sequences o and monitored execution states $((\sigma, \lambda), (V, W, L, w))$ *and* $((\sigma', \lambda'), (V', W', L', w'))$ *such that:*

$\star_1$ $((\sigma, \lambda), (V, W, L, w)) \vdash S$,

$\star_2$ $\Lbag((\sigma, \lambda), (V, W, L, w)) \vdash E(S)\Rbag \xrightarrow{o}_{\mathcal{M}(O)} \Lbag((\sigma', \lambda'), (V', W', L', w')) \vdash E(S')\Rbag$,

$\star_3$ $(w{\downarrow_E}) \notin \{\bot\}^\star$,

it is true that:

- *o is equal to* ϵ,
- $V' = V$, $W' = W$, $L' = L$ *and* $(w'{\downarrow_E}) = (w{\downarrow_E})$,
- $\forall x \notin \mathit{modified}(S) : \sigma'(x) = \sigma(x)$,
- $\forall l \notin \mathit{NLocks}(S) \cup \mathit{OLocks}(S) : l \in \lambda' \Leftrightarrow l \in \lambda$.

Proof. This follows directly from lemma B.2.1 and the transition function of Figure 4.7. Because of the hypothesis $\star_3$ any output is forbidden and any possible transition does not modifies V, W, L and $(w{\downarrow_E})$; except for assignments. However, in the case of an assignment to x, because of hypotheses $\star_3$ and $\star_1$, x belongs to W and V so the transition as no effect on the automaton state and the property on σ and σ' holds. Finally, for any lock which is not needed or already owned for the execution of S, an evaluation step of S will not modify the fact that l belongs or not to λ.

Lemma B.2.6 (High Branch Traversal).
For all evaluation contexts E, monitored statements S and monitored execution states $((\sigma, \lambda), (V, W, L, w))$ *such that:*

$\star_1$ $((\sigma, \lambda), (V, W, L, w)) \vdash E(S)$,

$\star_2$ $\lambda \cap (\mathit{NLocks}(S) \setminus \mathit{OLocks}(S)) = \emptyset$,

$\star_3$ $\neg\mathit{stoppable}(S)$,

$\star_4$ $(w{\downarrow_E}) \notin \{\bot\}^\star$,

it is true that, there exists a monitored execution state $((\sigma', \lambda'), (V', W', L', w'))$ *such that:*

- $\Lbag((\sigma, \lambda), (V, W, L, w)) \vdash E(S)\Rbag \xrightarrow{\epsilon\ \star}_{\mathcal{M}(O)} \Lbag((\sigma', \lambda'), (V', W', L', w')) \vdash E(\emptyset)\Rbag$.

Proof. The proof of this lemma follows from lemma B.2.1 and then goes by structural induction on S. Such induction goes by reducing the "structural size" of S at any induction step. Usually this type of induction does not work well with while-statements. For this proof, the hypothesis $\star_3$ solves the problem of while-statements. Synchronization commands can also be a problem. This one is solved by the hypotheses $\star_3$ and $\star_2$. Additionally, in order for the proof to go smoothly, it is required to give a smaller "structural size" to the branched statement structure than to the structure of if-statements and while-statements; similarly for locked statements and synchronization commands. The small details of the proof follows directly from Lemma B.2.5.

Lemma B.2.7 (One Step Soundness).
For all monitored statements S and S', and monitored execution states ζ_1, ζ_1' and ζ_2 such that:

$\star_1$ $\zeta_1 \cong \zeta_2$,

$\star_2$ $\zeta_1 \vdash S$,

$\star_3$ $\langle\!| \zeta_1 \vdash S |\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!| \zeta_1' \vdash S' |\!\rangle$,

it is true that, there exists a monitored execution state ζ_2' such that $\zeta_1' \cong \zeta_2'$ and one of the following is true:

${}^{B.2.7}\triangleleft_1$ $\langle\!| \zeta_2 \vdash S |\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!| \zeta_2' \vdash S' |\!\rangle$,

${}^{B.2.7}\triangleleft_2$ *all the following is true:*

(a) there exist an evaluation context E and three monitored statements S^b, P^e and P^u such that S is equal to $E(S^b)$, S' is equal to $E(\otimes(P^e, P^u)[P^e])$ and $w_1 = (w_1 \downarrow_E)$,

(b) $w_2' = w_2\top$,

(c) $o = \epsilon$,

(d) $\langle\!| \zeta_2 \vdash S |\!\rangle \xrightarrow{\epsilon}_{\mathcal{M}(O)} \langle\!| \zeta_2' \vdash E(\otimes(P^u, P^e)[P^u]) |\!\rangle$.

Proof sketch. The idea behind this lemma is that the executions of a given statement starting in various low-equivalent states either take the same step, or take two different branches if the statement to execute is a conditional branching with a condition influenced by some secret value.

Formal proof. The proof goes by induction on the length of the derivation tree of the global hypothesis $\star_3$ and by cases on the first rule used for this derivation tree. If the first rule used for the derivation tree of the global hypothesis $\star_3$ is:

($\mathbf{E}_{\mathcal{M}(O)}$-OK) then we can conclude that :

(1) S is an action (*i.e.* S = "**skip**" or S = "y **:=** e" or S = "**output** e"), "$(q_1,\ S) \xrightarrow{\text{OK}} q_1'$" and "$\langle\!|\, \varsigma_1 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \varsigma_1' \vdash \emptyset \,|\!\rangle$".

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-OK) and the grammar of the language. It can also be deduced from the rule (E$_{\mathcal{M}(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E$_O$).

(2) There exists q_2' such that $(q_2,\ S) \xrightarrow{\text{OK}} q_2'$ and $q_1' = q_2'$.

This follows directly from the local conclusion (1) and the global hypothesis $\star_1$ — as well as the definition of low equivalence (Definition B.1.3).

(•) There exists ζ_2' such that $\langle\!|\, \zeta_2 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \zeta_2' \vdash S' \,|\!\rangle$ and $\zeta_1' \cong \zeta_2'$.

Case 1: S = **skip**

(a) There exists ς_2' such that $\langle\!|\, \varsigma_2 \vdash A \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \varsigma_2' \vdash \emptyset \,|\!\rangle$ and $\varsigma_1' = \varsigma_2'$.

This follows directly from the case hypothesis, the rule (E$_O$-SKIP), the local conclusion (1) and the global hypothesis $\star_1$.

(•) There exists ζ_2' such that $\langle\!|\, \zeta_2 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \zeta_2' \vdash S' \,|\!\rangle$ and $\zeta_1' \cong \zeta_2'$.

This follows directly from the rule (E$_{\mathcal{M}(O)}$-OK) and the local conclusions (2) and (a).

Case 2: S = **output** e

(a) $FV(e) \cap V_1 = \emptyset$.

This follows from the case hypothesis, the fact that "$(q_1,\ S) \xrightarrow{\text{OK}} q_1'$" (from the local conclusion (1)), and the only possible automaton transition (T-OUTPUT-ok).

(b) $\sigma_1(e) = \sigma_2(e)$.

This follows directly from the local conclusion (a), the global hypothesis $\star_1$ and the definition of low equivalence (Definition B.1.3).

(c) There exists ς_2' such that $\langle\!|\, \varsigma_2 \vdash A \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \varsigma_2' \vdash \emptyset \,|\!\rangle$ and $\varsigma_1' = \varsigma_2'$.

This follows directly from the case hypothesis, the rule (E$_O$-OUTPUT), the global hypothesis $\star_1$ and the local conclusions (1) and (b).

(•) There exists ζ_2' such that $\langle\!|\, \zeta_2 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \zeta_2' \vdash S' \,|\!\rangle$ and $\zeta_1' \cong \zeta_2'$.

This follows directly from the rule (E$_{\mathcal{M}(O)}$-OK) and the local conclusions (2) and (c).

Case 3: S = y **:=** e

(a) $\sigma_1' = \sigma_1[y \mapsto \sigma_1(e)]$ and $\lambda_1 = \lambda_1'$.

This follows directly from the case hypothesis, the local conclusion (1) and the only possible rule (E$_O$-ASSIGN).

(b) There exist σ_2' and λ_2' such that $\langle\!|\, \varsigma_2 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \varsigma_2' \vdash \emptyset \,|\!\rangle$, $\sigma_2' = \sigma_2[y \mapsto \sigma_2(e)]$ and $\lambda_2 = \lambda_2'$.

This follows directly from the case hypothesis and the only possible rule (E$_O$-ASSIGN).

(c) There exists ζ_2' such that $\langle\!|\, \zeta_2 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \zeta_2' \vdash S' \,|\!\rangle$.

This follows from the local conclusions (2) and (b).

(d) $\forall x \notin V_1' : \sigma_1'(x) = \sigma_2'(x)$.

Sub-case 2.a: $FV(e) \cap V_1 \neq \emptyset$ or $y \in W_1$.

(α) $V'_1 = V_1 \cup \{y\}$.

From the sub-case hypothesis, the automaton transition of the local conclusion (1) is (T-ASSIGN-sec). From which comes the desired result.

(•) $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$.

This follows directly from the global hypothesis $\star_1$ and local conclusions (a), (b) and (α).

Sub-case 2.b: $FV(e) \cap V_1 = \emptyset$ and $y \notin W_1$.

(α) $V'_1 = V_1 \setminus \{y\}$.

From the sub-case hypothesis, the automaton transition of the local conclusion (1) is (T-ASSIGN-pub). From which comes the desired result.

(β) $\sigma_1(e) = \sigma_2(e)$.

This follows directly from the global hypothesis $\star_1$ and the sub-case hypothesis $FV(e) \cap V_1 = \emptyset$.

(•) $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$.

This follows directly from the global hypothesis $\star_1$ and the local conclusions (α), (a), (b) and (β).

(•) There exists ζ'_2 such that $\langle\!|\, \zeta_2 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \zeta'_2 \vdash S' \,|\!\rangle$ and $\zeta'_1 \cong \zeta'_2$.

The first part of the above result comes directly from the local conclusion (c). The second part of the result — $\zeta'_1 \cong \zeta'_2$ — comes from the definition B.1.3, the local conclusions (2) for the first part of the definition, the local conclusions (d) for the second part of the definition, and the global hypothesis $\star_1$ and the local conclusions (a) and (b) for the last part.

(E$_{\mathcal{M}(O)}$-EDIT) then we can conclude that :

(1) the following holds:

- $(q_1,\ S) \xrightarrow{\textbf{output}\ \delta} q'_1$,
- S is "**output** e",
- $w_1 \in \{\bot\}^\star$ and $FV(e) \cap V_1 \neq \emptyset$,
- $q'_1 = q_1$,
- $o = \delta$,
- $\varsigma'_1 = \varsigma_1$.

This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-EDIT), the only automaton transition returning an action as output (T-OUTPUT-def) and the rule (E$_O$-OUTPUT).

(2) There exists q'_2 such that $(q_2,\ S) \xrightarrow{\textbf{output}\ \delta} q'_2$ and $q'_1 = q'_2$.

From the local conclusion (1), $w_1 \in \{\bot\}^\star$ and $FV(e) \cap V_1 \neq \emptyset$. Hence, from the global hypothesis $\star_1$, $w_2 \in \{\bot\}^\star$ and $FV(e) \cap V_2 \neq \emptyset$. Then the automaton transition (T-OUTPUT-def) applies to q_2; with the consequences that $q'_2 = q_2$. Hence, from the global hypothesis $\star_1$, $q'_1 = q'_2$.

(•) There exists ζ_2' such that $\Lbag \zeta_2 \vdash S \Rbag \xrightarrow{o}_{\mathcal{M}(O)} \Lbag \zeta_2' \vdash S' \Rbag$ and $\zeta_1' \cong \zeta_2'$.

The rule (E$_O$-OUTPUT) implies that there exists ς_2' such that $\Lbag \varsigma_2 \vdash$ **output** $\delta \Rbag \xrightarrow{\delta}_O \Lbag \varsigma_2' \vdash \emptyset \Rbag$ with $\varsigma_2' = \varsigma_2$. Hence the rule (E$_{\mathcal{M}(O)}$-EDIT) and the local conclusion (2) imply that $\Lbag \zeta_2 \vdash S \Rbag \xrightarrow{o}_{\mathcal{M}(O)} \Lbag \zeta_2' \vdash S' \Rbag$. From the global hypothesis $\star_1$ and the local conclusions (1) and (2), $q_1' = q_2'$. Hence from the global hypothesis $\star_1$, the fact that $\varsigma_1' = \varsigma_1$ (from the local conclusion (1)) and the previous conclusion $\varsigma_2' = \varsigma_2$, $\zeta_1' \cong \zeta_2'$.

(E$_{\mathcal{M}(O)}$-NO) then we can conclude that :

(1) $w_1 \notin \{\perp\}^\star$. This follows directly from the definition of the rule (E$_{\mathcal{M}(O)}$-NO) and the only automaton transition denying an action (T-OUTPUT-no).

(•) The first rule used can not be (E$_{\mathcal{M}(O)}$-NO). This follows from the fact that the local conclusion (1) is in contradiction with the global hypothesis $\star_1$.

(E$_{\mathcal{M}(O)}$-IF) then we can conclude that :

(1) there exists v belonging to the set $\{\texttt{true}, \texttt{false}\}$ such that:

- S is "**if** e **then** $P^{\texttt{true}}$ **else** $P^{\texttt{false}}$ **end**",
- $\sigma_1(e) = v$,
- $(q_1, \texttt{branch}(\lambda_1, e, P^v, P^{\neg v})) \xrightarrow{OK} q_1'$,
- $\varsigma_1' = \varsigma_1$,
- S' is "$\otimes(P^v, P^{\neg v})[P^v]$".

This follows directly from the rule (E$_{\mathcal{M}(O)}$-IF).

(•) the property holds.

Case 1: $\sigma_2(e) = \sigma_1(e)$

(a) there exists q_2' such that $(q_2, \texttt{branch}(\lambda_2, e, P^v, P^{\neg v})) \xrightarrow{OK} q_2'$ and $q_2' = q_1'$.

The conditions of the two possible transitions for the automaton input $\texttt{branch}(\lambda_2, e, P^v, P^{\neg v})$ are based on values which, because of the global hypothesis $\star_1$, are equal for q_2 and q_1. Therefore, the same transition can be taken by the automaton.

(•) There exists ζ_2' such that $\Lbag \zeta_2 \vdash S \Rbag \xrightarrow{o}_{\mathcal{M}(O)} \Lbag \zeta_2' \vdash S' \Rbag$ and $\zeta_1' \cong \zeta_2'$.

This follows from the rule (E$_{\mathcal{M}(O)}$-IF), the global hypothesis $\star_1$, the case hypothesis and the local conclusions (1) and (a).

Case 2: $\sigma_2(e) \neq \sigma_1(e)$

(a) there exist an evaluation context E and a monitored statement S^b such that S is equal to $E(S^b)$ and S' is equal to $E(\otimes(P^v, P^{\neg v})[P^v])$.

This follows directly from the rule (E$_{\mathcal{M}(O)}$-IF). E is $\square$ and S^b is S.

(b) $FV(e) \cap V_1 \neq \emptyset$.

This follows by contradiction using the global hypothesis $\star_1$ and the case hypothesis.

(c) there exists w^{p} such that $w'_1 = w^{\mathrm{p}}\top$.

The local conclusion (b) implies that the transition used for the automaton input $\mathtt{branch}(\lambda, e, P^{v}, P^{\neg v})$ in state q_1 is (T-BRANCH-high). Hence, the desired result holds.

(d) there exists q'_2 such that $(q_2,\ \mathtt{branch}(\lambda_2, e, P^{v}, P^{\neg v})) \xrightarrow{OK} q^{\mathrm{t}}_2$ and $q'_2 = q'_1$.

The conditions of the two possible transitions for the automaton input $\mathtt{branch}(\lambda_2, e, P^{v}, P^{\neg v})$ are based on values which, because of the global hypothesis $\star_1$, are equal for q_2 and q_1. Therefore, the same transition can be taken by the automaton.

(e) there exists ζ'_2 such that $\langle\!| \zeta_2 \vdash S |\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!| \zeta'_2 \vdash E(\otimes(P^{\neg v}, P^{v})[P^{\neg v}]) |\!\rangle$ and $\zeta'_1 \cong \zeta'_2$.

The rule (E$_{\mathcal{M}(O)}$-IF) and the local conclusions (1) and (d) imply the desired result.

(•) the second possible global conclusion holds.

This follows directly from the local conclusions (1), (a), (c), (d) and (e).

(E$_{\mathcal{M}(O)}$-WHILE) then we can conclude that :

(•) the property holds.

The proof is similar to the one for (E$_{\mathcal{M}(O)}$-IF).

(E$_{\mathcal{M}(O)}$-WITH) then we can conclude that :

(1) there exists q^{t}_1 such that:

- S is **with** $\bar{x}$ **when** e **do** S^{w} **done**,
- $\bar{x} \cap \lambda_1 = \emptyset$,
- $\sigma_1(e) = \mathtt{true}$,
- $(q_1,\ \mathtt{sync}(\bar{x}, e)) \xrightarrow{OK} q'_1$,
- $FV(e) \cap V_1 = \emptyset$,
- $L_1 \cap \bar{x} = \emptyset$,
- $\zeta'_1 = ((\sigma_1, \lambda_1 \cup \bar{x}), q'_1)$,
- $S' = \odot\bar{x}[S^{\mathrm{w}}]$.

This follows directly from the global hypothesis $\star_1$, the semantic rule (E$_{\mathcal{M}(O)}$-WITH) and from the only automaton transition applying to the automaton input.

(2) $\bar{x} \cap \lambda_2 = \emptyset$ and $\sigma_2(e) = \mathtt{true}$.

Both equalities follow from the global hypothesis $\star_1$, the definition of low equivalence for states of monitored executions (definition B.1.3) and the local conclusion (1) — $L_1 \cap \bar{x} = \emptyset$ and $FV(e) \cap V_1 = \emptyset$.

(3) There exists q'_2 such that $(q_2, \mathtt{sync}(\bar{x}, e)) \xrightarrow{OK} q'_2$ and $q'_2 = q'_1$.

The global hypothesis $\star_1$ and the local conclusion (1) imply that $L_2 \cap \bar{x} = \emptyset$ and $FV(e) \cap V_2 = \emptyset$. Therefore, the desired results hold.

(4) There exists ζ'_2 such that $\langle\!| \zeta_2 \vdash S |\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!| \zeta'_2 \vdash S' |\!\rangle$ and $\zeta'_2 = ((\sigma_2, \lambda_2 \cup \bar{x}), q'_2)$.

This follows from the rule (E$_{\mathcal{M}(O)}$-WITH) and the local conclusions (2) and (3).

(5) $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$.

The local conclusion (1) and the automaton transition (T-SYNC) imply that V'_1 is equal to V_1. In addition, the evaluation rule ($\mathrm{E}_{\mathcal{M}(O)}$-WITH), the global hypothesis $\star_3$ and the local conclusion (4) imply that σ'_1 is equal to σ_1 and σ'_2 is equal to σ_2. Therefore, the desired result follows from the global hypothesis $\star_1$.

(6) $\forall x \notin L'_1 : x \in \lambda'_1 \Leftrightarrow x \in \lambda'_2$.

The local conclusion (1) and the automaton transition (T-SYNC) imply that $L'_1 = L_1$. In addition, the evaluation rule ($\mathrm{E}_{\mathcal{M}(O)}$-WITH), the global hypothesis $\star_3$ and the local conclusion (4) imply that $\lambda'_1 = \lambda_1 \cup \bar{x}$ and $\lambda'_2 = \lambda_2 \cup \bar{x}$. So the desired result is implied by the global hypothesis $\star_1$.

(•) There exists ζ'_2 such that $\langle\!\langle \zeta_2 \vdash S \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!\langle \zeta'_2 \vdash S' \rangle\!\rangle$ and $\zeta'_1 \cong \zeta'_2$.

This follows directly from the local conclusions (4), (5) and (6).

($\mathbf{E}_{\mathcal{M}(O)}$-SEQ$_{\mathbf{exit}}$) then we can conclude that :

(1) S is "$\emptyset$; S^{r}", $o = \epsilon$, $\zeta'_1 = \zeta_1$ and $S' = S^{\mathrm{r}}$.

This follows directly from ($\mathrm{E}_{\mathcal{M}(O)}$-SEQ$_{\mathrm{exit}}$).

(•) There exists ζ'_2 such that $\langle\!\langle \zeta_2 \vdash S \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!\langle \zeta'_2 \vdash S' \rangle\!\rangle$ and $\zeta'_1 \cong \zeta'_2$.

This follows directly from ($\mathrm{E}_{\mathcal{M}(O)}$-SEQ$_{\mathrm{exit}}$) and the global hypothesis $\star_1$.

($\mathbf{E}_{\mathcal{M}(O)}$-SEQ$_{\mathbf{step}}$) then we can conclude that :

(1) the following holds:

- S is S^{i} ; S^{r},
- $\langle\!\langle \zeta_1 \Vdash S^{\mathrm{i}} \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!\langle \zeta'_1 \Vdash S^{\mathrm{j}} \rangle\!\rangle$ is a sub-derivation tree of the derivation tree of S,
- $S' = S^{\mathrm{j}}$; S^{r}.

This follows directly from the rule ($\mathrm{E}_{\mathcal{M}(O)}$-SEQ$_{\mathrm{step}}$).

(2) There exists ζ'_2 such that $\zeta'_1 \cong \zeta'_2$ and one of the possible global conclusions holds for the monitored statement S^{i}.

This follows directly from the inductive hypothesis.

(•) The property holds.

Case 1: the first global conclusion holds for S^{i}.

(a) $\langle\!\langle \zeta_2 \vdash S^{\mathrm{i}} \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!\langle \zeta'_2 \vdash S^{\mathrm{j}} \rangle\!\rangle$

This follows directly from the local conclusion (2) and the case hypothesis.

(•) $\langle\!\langle \zeta_2 \vdash S \rangle\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!\langle \zeta'_2 \vdash S' \rangle\!\rangle$

This follows directly from the local conclusion (a) and the definition of the evaluation rule ($\mathrm{E}_{\mathcal{M}(O)}$-SEQ$_{\mathrm{step}}$).

Case 2: the second global conclusion holds for S^{i}.

(a) there exist an evaluation context E and three monitored statements S^b, P^e and P^u such that:

- S^{i} is equal to $E(S^b)$ and S^{j} is equal to $E(\otimes(P^e, P^u)[P^e])$,
- $w_2' = w_2\top$,
- $o = \epsilon$,
- $\llparenthesis\, \zeta_2 \vdash S^{\mathrm{i}} \,\rrparenthesis \xrightarrow{\epsilon}_{\mathcal{M}(O)} \llparenthesis\, \zeta_2' \vdash E(\otimes(P^u, P^e)[P^u]) \,\rrparenthesis$.

This follows directly from the local conclusion (2) and the case hypothesis.

(b) $\llparenthesis\, \zeta_2 \vdash S^{\mathrm{i}}\ ;\ S^{\mathrm{r}} \,\rrparenthesis \xrightarrow{\epsilon}_{\mathcal{M}(O)} \llparenthesis\, \zeta_2' \vdash E(\otimes(P^u, P^e)[P^u])\ ;\ S^{\mathrm{r}} \,\rrparenthesis$

This follows directly from the local conclusion (a) and the evaluation rule ($\mathrm{E}_{\mathcal{M}(O)}$-$\mathrm{SEQ}_{\mathrm{step}}$).

(c) there exist E', equal to "$E\ ;\ S^{\mathrm{r}}$", such that S is equal to $E'(S^b)$ and S' is equal to $E(\otimes(P^e, P^u)[P^e])$.

This result is implied by the local conclusions (1) and (a).

(d) $\llparenthesis\, \zeta_2 \vdash S \,\rrparenthesis \xrightarrow{\epsilon}_{\mathcal{M}(O)} \llparenthesis\, \zeta_2' \vdash E'(\otimes(P^u, P^e)[P^u]) \,\rrparenthesis$.

From the local conclusion (c), $E'(\otimes(P^u, P^e)[P^u])$ is equal to "$E(\otimes(P^u, P^e)[P^u])\ ;\ S^{\mathrm{r}}$". Hence, the local conclusions (1) and (b) imply the desired result.

(•) the second possible global conclusion holds.

This follows directly from the local conclusions (c), (a) and (d).

($\mathbf{E}_{\mathcal{M}(O)}$-BRANCH$_{\mathbf{exit}}$) then we can conclude that :

(1) there exists q_1^{t} such that:

- S is $\otimes(P^e, P^u)[\emptyset]$,
- $(q_1, \mathtt{merge}(P^e, P^u)) \xrightarrow{OK} q_1$,
- $\varsigma_1' = \varsigma_1$
- S' is $\emptyset$.

This follows directly from the rule ($\mathrm{E}_{\mathcal{M}(O)}$-$\mathrm{BRANCH}_{\mathrm{exit}}$) and the global hypothesis $\star_1$.

(2) There exists q_2' such that $(q_2, \mathtt{merge}(P^e, P^u)) \xrightarrow{OK} q_2$.

This follows directly from the global hypothesis $\star_1$ and the definition of the transition (T-MERGE-low).

(•) There exists ζ_2' such that $\llparenthesis\, \zeta_2 \vdash S \,\rrparenthesis \xrightarrow{o}_{\mathcal{M}(O)} \llparenthesis\, \zeta_2' \vdash S' \,\rrparenthesis$ and $\zeta_1' \cong \zeta_2'$.

This follows directly from the global hypothesis $\star_1$ and local conclusions (1) and (2).

($\mathbf{E}_{\mathcal{M}(O)}$-BRANCH$_{\mathbf{step}}$) then we can conclude that :

(1) there exist three monitoring statements S^b, $S^{b'}$, P^e and P^u such that:

- $S^b \neq \emptyset$,
- S is equal to $\otimes(P^e, P^u)[S^b]$,
- S' is equal to $\otimes(P^e, P^u)[S^{b'}]$,
- $\llparenthesis\, \zeta_1 \Vdash S^b \,\rrparenthesis \xrightarrow{o}_{\mathcal{M}(O)} \llparenthesis\, \zeta_1' \Vdash S^{b'} \,\rrparenthesis$ is a sub-derivation tree of the global hypothesis $\star_3$.

This follows directly from the definition of ($E_{\mathcal{M}(O)}$-BRANCH$_{\text{step}}$).

(2) There exists ζ_2' such that $\zeta_1' \cong \zeta_2'$ and one of the possible global conclusions holds for the monitored statement S^b.
This follows directly from the inductive hypothesis and the local conclusion (1).

(•) The property holds.

Case 1: the first global conclusion holds for S^b.

(a) $\Lbag \zeta_2 \Vdash S^b \Rbag \xrightarrow{o}_{\mathcal{M}(O)} \Lbag \zeta_2' \Vdash S^{b'} \Rbag$
This follows directly from the local conclusion (2) and the case hypothesis.

(•) There exists ζ_2' such that $\Lbag \zeta_2 \Vdash S \Rbag \xrightarrow{o}_{\mathcal{M}(O)} \Lbag \zeta_2' \Vdash S' \Rbag$ and $\zeta_1' \cong \zeta_2'$.
This is direct using the definition of the rule ($E_{\mathcal{M}(O)}$-BRANCH$_{\text{step}}$) and the local conclusion (a).

Case 2: the second global conclusion holds for S^b.

(a) there exist an evaluation context E and three monitored statements S^{b_2}, P^{e_2} and P^{u_2} such that:

- S^b is equal to $E(S^{b_2})$ and $S^{b'}$ is equal to $E(\otimes(P^{e_2}, P^{u_2})[P^{e_2}])$,
- $w_2' = w_2\top$,
- $o = \epsilon$,
- $\Lbag \zeta_2 \vdash S^b \Rbag \xrightarrow{\epsilon}_{\mathcal{M}(O)} \Lbag \zeta_2' \vdash E(\otimes(P^{u_2}, P^{e_2})[P^{u_2}]) \Rbag$.

This follows directly from the local conclusion (2) and the case hypothesis.

(b) $\Lbag \zeta_2 \vdash \otimes(P^e, P^u)[S^{\text{b}}] \Rbag \xrightarrow{\epsilon}_{\mathcal{M}(O)} \Lbag \zeta_2' \vdash \otimes(P^e, P^u)[E(\otimes(P^{u_2}, P^{e_2})[P^{u_2}])] \Rbag$
This follows directly from the local conclusion (a) and the evaluation rule ($E_{\mathcal{M}(O)}$-BRANCH$_{\text{step}}$).

(c) there exist E', equal to "$\otimes(P^e, P^u)[E]$", such that S is equal to $E'(S^{b_2})$ and S' is equal to $E(\otimes(P^{e_2}, P^{u_2})[P^{e_2}])$.
This result is implied by the local conclusions (1) and (a).

(d) $\Lbag \zeta_2 \vdash S \Rbag \xrightarrow{\epsilon}_{\mathcal{M}(O)} \Lbag \zeta_2' \vdash E'(\otimes(P^{u_2}, P^{e_2})[P^{u_2}]) \Rbag$.
From the local conclusion (c), $E'(\otimes(P^{u_2}, P^{e_2})[P^{u_2}])$ is equal to "$\otimes(P^e, P^u)[E(\otimes(P^{u_2}, P^{e_2})[P^{u_2}])]$". Hence, the local conclusions (1) and (b) imply the desired result.

(•) the second possible global conclusion holds.
This follows directly from the local conclusions (c), (a) and (d).

($E_{\mathcal{M}(O)}$-LOCKED$_{\text{exit}}$) then we can conclude that :

(1) there exists a set of identifiers $\bar{x}$ such that:

- S is $\odot\bar{x}[\emptyset]$,
- S' is $\emptyset$,
- $\zeta_1' = ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1)$.

This follows directly from the rule ($E_{\mathcal{M}(O)}$-LOCKED$_{\text{exit}}$).

(•) There exists ζ_2' such that $\langle\!|\, \zeta_2 \vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \zeta_2' \vdash S' \,|\!\rangle$ and $\zeta_1' \cong \zeta_2'$.
This follows directly from the local conclusions (1), the global hypothesis $\star_1$ and the rule ($E_{\mathcal{M}(O)}$-LOCKED$_{\text{exit}}$).

($E_{\mathcal{M}(O)}$-LOCKED$_{\text{step}}$) then we can conclude that :

(1) there exist a set of identifiers $\bar{x}$, a set of locks λ_1^{t} and two monitored statements S^{s} and $S^{\text{s}'}$, such that:

- S is equal to $\odot\bar{x}[S^{\text{s}}]$,
- S' is equal to $\odot\bar{x}[S^{\text{s}'}]$,
- $S^{\text{s}} \neq \emptyset$,
- $\langle\!|\, ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1) \Vdash S^{\text{s}} \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, ((\sigma_1', \lambda_1^{\text{t}}), q_1') \Vdash S^{\text{s}'} \,|\!\rangle$,
- $\lambda_1' = \lambda_1^{\text{t}} \cup \bar{x}$.

This follows directly from the definition of ($E_{\mathcal{M}(O)}$-LOCKED$_{\text{step}}$).

(2) $\zeta_1 \cong \zeta_2 \Rightarrow ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1) \cong ((\sigma_2, \lambda_2 \setminus \bar{x}), q_2)$.
This is obvious from the Definition B.1.3.

(3) $\zeta_1 \vdash \odot\bar{x}[S^{\text{s}}] \Rightarrow ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1) \vdash S^{\text{s}}$.
This is obvious from the Definition B.1.4.

(4) There exists $((\sigma_2', \lambda_2^{\text{t}}), q_2')$ such that $((\sigma_1', \lambda_1^{\text{t}}), q_1') \cong ((\sigma_2', \lambda_2^{\text{t}}), q_2')$ and one of the possible global conclusions holds for the monitored statement S^{s}.
This follows directly from the inductive hypothesis and the local conclusions (1), (2) and (3).

(•) The property holds.

Case 1: the first global conclusion holds for S^{s}.

(a) $\langle\!|\, ((\sigma_2, \lambda_2 \setminus \bar{x}), q_2) \Vdash S^{\text{s}} \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, ((\sigma_2', \lambda_2^{\text{t}}), q_2') \Vdash S^{\text{s}'} \,|\!\rangle$
This follows directly from the local conclusion (2) and the case hypothesis.

(•) There exists ζ_2', equals to $((\sigma_2', \lambda_2^{\text{t}} \cup \bar{x}), q_2')$, such that $\langle\!|\, \zeta_2 \Vdash S \,|\!\rangle \xrightarrow{o}_{\mathcal{M}(O)} \langle\!|\, \zeta_2' \Vdash S' \,|\!\rangle$ and $\zeta_1' \cong \zeta_2'$.
This is direct using the definition of the rule ($E_{\mathcal{M}(O)}$-BRANCH$_{\text{step}}$) and the local conclusions (a), (4) and (1).

Case 2: the second global conclusion holds for S^{s}.

(a) there exist an evaluation context E and three monitored statements S^b, P^e and P^u such that:

- S^{s} is equal to $E(S^b)$ and $S^{\text{s}'}$ is equal to $E(\otimes(P^e, P^u)[P^e])$,
- $w_2' = w_2\top$,
- $o = \epsilon$,
- $\langle\!|\, ((\sigma_2, \lambda_2 \setminus \bar{x}), q_2) \vdash S^{\text{s}} \,|\!\rangle \xrightarrow{\epsilon}_{\mathcal{M}(O)} \langle\!|\, ((\sigma_2', \lambda_2^{\text{t}}), q_2') \vdash E(\otimes(P^u, P^e)[P^u]) \,|\!\rangle$.

This follows directly from the local conclusion (4) and the case hypothesis.

(b) there exists E', equal to "$\odot\bar{x}[E]$", such that S is equal to $E'(S^b)$ and S' is equal to $E'(\otimes(P^e, P^u)[P^e])$.
This result is implied by the local conclusions (1) and (a).

(c) $\langle\!|\, ((\sigma_2, \lambda_2), q_2) \vdash \odot\bar{x}[S^s] \,|\!\rangle \xrightarrow{\epsilon}_{\mathcal{M}(O)} \langle\!|\, ((\sigma_2', \lambda_2^t \cup \bar{x}), q_2') \vdash \odot\bar{x}[E(\otimes(P^u, P^e)[P^u])] \,|\!\rangle$.
This follows directly from the evaluation rule ($E_{\mathcal{M}(O)}$-LOCKED$_{step}$) and the local conclusions (1) and (a).

(d) $\langle\!|\, \zeta_2 \vdash S \,|\!\rangle \xrightarrow{\epsilon}_{\mathcal{M}(O)} \langle\!|\, \zeta_2' \vdash E'(\otimes(P^u, P^e)[P^u]) \,|\!\rangle$ with $\lambda_2' = \lambda_2^t \cup \bar{x}$.
This result is implied by the local conclusions (c), (1), and (b).

(e) $\zeta_1' \cong \zeta_2'$.
The local conclusion (4) and the definition B.1.3 imply $\forall x \notin L_1' : x \in \lambda_1^t \Leftrightarrow x \in \lambda_2^t$. Combined with the local conclusions (1) and (d), it implies $\forall x \notin L_1' : x \in \lambda_1' \Leftrightarrow x \in \lambda_2'$. This property and the local conclusion (4) imply the desired result.

(•) the second possible global conclusion holds.
This follows directly from the local conclusions (b), (a), (d) and (e).

Lemma B.2.8 (General Soundness). *For all thread pools Θ_1, Θ_2 and Θ_1^f, concurrent execution states ϑ_1, ϑ_2 and ϑ_1^f, and output sequences o such that:*

$\star_1$ $(\vartheta_1 \vdash \Theta_1) \sim (\vartheta_2 \vdash \Theta_2)$,

$\star_2$ $\langle\!|\, \vartheta_1 \vdash \Theta_1 \,|\!\rangle \xrightarrow{o}{}^{\star}_{\mathcal{M}(O)} \langle\!|\, \vartheta_1^f \vdash \Theta_1^f \,|\!\rangle$,

it is true that, there exists a concurrent execution state ϑ_2^f and a thread pool Θ_2^f such that:

- $\langle\!|\, \vartheta_2 \vdash \Theta_2 \,|\!\rangle \xrightarrow{o}{}^{\star}_{\mathcal{M}(O)} \langle\!|\, \vartheta_2^f \vdash \Theta_2^f \,|\!\rangle$.

Proof. The proof goes by induction on the length of the derivation of the global hypothesis $\star_2$.

Obviously, the lemma holds if the length of the derivation is equal to 0. In that case o is equal to ϵ and the conclusion holds with $\Theta_2^f = \Theta_2$ and $\vartheta_2^f = \vartheta_2$.

Otherwise, the global hypothesis $\star_2$ implies that there exist a concurrent execution state ϑ_1' and a thread pool Θ_1' such that:

$$\langle\!|\, \vartheta_1 \Vdash \Theta_1 \,|\!\rangle \xrightarrow{o^p}_{\mathcal{M}(O)} \langle\!|\, \vartheta_1' \Vdash \Theta_1' \,|\!\rangle \tag{B.1}$$

$$\langle\!|\, \vartheta_1' \vdash \Theta_1' \,|\!\rangle \xrightarrow{o^s}{}^{\star}_{\mathcal{M}(O)} \langle\!|\, \vartheta_1^f \vdash \Theta_1^f \,|\!\rangle \tag{B.2}$$

$$o = o^p o^s \tag{B.3}$$

The remaining of the proof demonstrates the existence of a concurrent execution state ϑ_2' and a thread pool Θ_2' such that:

$$\langle\!|\, \vartheta_2 \vdash \Theta_2 \,|\!\rangle \xrightarrow{o^p}{}^{\star}_{\mathcal{M}(O)} \langle\!|\, \vartheta_2' \vdash \Theta_2' \,|\!\rangle$$
$$(\vartheta_1' \vdash \Theta_1') \sim (\vartheta_2' \vdash \Theta_2')$$

The desired property will then follow by induction. As $(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$, using the inductive hypothesis on (B.2), we get that there exists an execution state ϑ^f_2 and a thread pool Θ^f_2 such that:

$$\langle\!| \vartheta'_2 \vdash \Theta'_2 |\!\rangle \xrightarrow{o^s}{}^{\star}_{\mathcal{M}(O)} \langle\!| \vartheta^f_2 \vdash \Theta^f_2 |\!\rangle$$

Consequently, $\langle\!| \vartheta_2 \vdash \Theta_2 |\!\rangle \xrightarrow{o}{}^{\star}_{\mathcal{M}(O)} \langle\!| \vartheta^f_2 \vdash \Theta^f_2 |\!\rangle$.

Let ϑ_1 be equal to $(\varsigma_1, \mathtt{Q}_1)$. The only possible rule for the previous one step evaluation — (B.1) — is ($\mathrm{E}_{\mathcal{M}(O)}$-CONCUR). It implies that there exists ι belonging to the domain of Θ_1 such that:

$$\langle\!| (\varsigma_1, \mathrm{extractState}(\mathtt{Q}_1, \iota)) \Vdash \Theta_1(\iota) |\!\rangle \xrightarrow{o^p}_{\mathcal{M}(O)} \langle\!| (\varsigma'_1, q'_1) \Vdash S'_1 |\!\rangle \tag{B.4}$$

$$\vartheta'_1 = (\varsigma'_1, \mathrm{updateStates}(\mathtt{Q}_1, \iota, q'_1)) \tag{B.5}$$

$$\Theta'_1 = \Theta_1[\iota \mapsto S'_1] \tag{B.6}$$

Let ζ_1 be $(\varsigma_1, \mathrm{extractState}(\mathtt{Q}_1, \iota))$, ζ_2 be $(\varsigma_2, \mathrm{extractState}(\mathtt{Q}_2, \iota))$ — with $\vartheta_2 = (\varsigma_2, \mathtt{Q}_2)$ —, S_1 be $\Theta_1(\iota)$ and S_2 be $\Theta_2(\iota)$. Because of the global hypothesis $\star_1$ and the definition of bisimulation for thread pools, the following is true: $(\zeta_1 \vdash S_1) \sim (\zeta_2 \vdash S_2)$. Therefore, one of the conclusions of definition B.1.9 has to hold.

Case 1: The conclusion ${}^{\text{B.1.9}}\triangleleft_1$ of definition B.1.9 holds.

This conclusion states that "$\zeta_1 \cong \zeta_2$", "$\zeta_1 \vdash S_1$" and S_1 is equal to S_2. Hence, as from (B.4) the monitored statement S_1 in the monitored execution state ζ_1 evaluates to S'_1 yielding the output sequence o^p and the monitored execution state ζ'_1, lemma B.2.7 applies and one of its global conclusions has to hold.

Sub-case 1.a: The global conclusion ${}^{\text{B.2.7}}\triangleleft_1$ of lemma B.2.7 holds.

This conclusion states that there exists a monitored execution state ζ'_2 such that:

$$\zeta'_1 \cong \zeta'_2 \tag{B.7}$$

$$\langle\!| \zeta_2 \vdash S_2 |\!\rangle \xrightarrow{o^p}_{\mathcal{M}(O)} \langle\!| \zeta'_2 \vdash S'_1 |\!\rangle \tag{B.8}$$

Let ϑ'_2 be equal to $(\varsigma'_2, \mathrm{updateStates}(\mathtt{Q}_2, \iota, q'_2))$ — with $\zeta'_2 = (\varsigma'_2, q'_2)$ — and Θ'_2 be equal to $\Theta_2[\iota \mapsto S'_1]$. The previous evaluation of S_2 — (B.8) — and the rule ($\mathrm{E}_{\mathcal{M}(O)}$-CONCUR) implies:

$$\langle\!| \vartheta_2 \Vdash \Theta_2 |\!\rangle \xrightarrow{o^p}_{\mathcal{M}(O)} \langle\!| \vartheta'_2 \Vdash \Theta'_2 |\!\rangle$$

The global hypothesis $\star_1$ implies that $\zeta_1 \vdash S_1$ and $\zeta_2 \vdash S_2$. Hence, lemma B.2.3, (B.4) and (B.8) are sufficient to conclude that $\zeta'_1 \vdash S_1$ and $\zeta'_2 \vdash S_2$. Therefore, as $\zeta'_1 \cong \zeta'_2$ and $\Theta'_1(\iota) = \Theta'_2(\iota)$, the following is true:

$$(\zeta'_1 \vdash \Theta'_1(\iota)) \sim (\zeta'_2 \vdash \Theta'_2(\iota))$$

and, with $\vartheta'_1 = (\varsigma'_1, \mathtt{Q}'_1)$ and $\vartheta'_2 = (\varsigma'_2, \mathtt{Q}'_2)$,

$$((\varsigma'_1, \text{extractState}(\mathtt{Q}'_1, \imath)) \vdash \Theta'_1(\imath)) \sim ((\varsigma'_2, \text{extractState}(\mathtt{Q}'_2, \imath)) \vdash \Theta'_2(\imath))$$

Equation (B.7) implies:

- $V'_1 = V'_2$, $W'_1 = W'_2$ and $L'_1 = L'_2$,
- $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$,
- $\forall l \notin L'_1 : x \in \lambda'_1 \Leftrightarrow x \in \lambda'_2$.

For every $\jmath$, let $w_1^{\jmath}$ be the forth element of extractState$(\mathtt{Q}_1, \jmath)$. extractState$(\mathtt{Q}_1, \jmath)$ is equal to $(V_1, W_1, L_1, w_1^{\jmath})$. Let $w_2^{\jmath}$ be similarly defined. For every $\jmath$ — different from $\imath$ — in the domain of Θ'_1, (B.6) implies $\Theta'_1(\jmath) = \Theta_1(\jmath)$; and similarly for $\Theta'_2(\jmath)$. As well, (B.5) implies that extractState$(\mathtt{Q}'_1, \jmath)$ is equal to $(V'_1, W'_1, L'_1, w_1^{\jmath})$; and similarly with extractState$(\mathtt{Q}'_2, \jmath)$. Therefore, lemma B.2.4 applies and, for every $\jmath$ in the domain of Θ_1 and different from $\imath$:

$$((\varsigma'_1, \text{extractState}(\mathtt{Q}'_1, \jmath)) \vdash \Theta'_1(\jmath)) \sim ((\varsigma'_2, \text{extractState}(\mathtt{Q}_2, \jmath)) \vdash \Theta'_2(\jmath))$$

Consequently,

$$(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$$

Sub-case 1.b: The global conclusion ${}^{\text{B.2.7}}\triangleleft_2$ of lemma B.2.7 holds.

This conclusion states that there exists a monitored execution state ζ'_2 such that $\zeta'_1 \cong \zeta'_2$ and:

1. there exist an evaluation context E, a value v belonging to the set $\{\mathtt{true}, \mathtt{false}\}$ and three monitored statements S^b, $P^{\mathtt{true}}$ and $P^{\mathtt{false}}$ such that S_1 — and S_2 because $S_1 = S_2$ — is equal to $E(S^b)$, S'_1 is equal to $E(\otimes(S_v, S_{\neg v})[S_v])$ and $w_1 = (w_1 \downarrow_E)$,
2. $w'_2 = w_2 \top$,
3. $o^p = \epsilon$,
4. $\langle\!\langle \zeta_2 \vdash S_2 \rangle\!\rangle \xrightarrow{\epsilon}_{\mathcal{M}(O)} \langle\!\langle \zeta'_2 \vdash E(\otimes(S_{\neg v}, S_v)[S_{\neg v}]) \rangle\!\rangle$.

Let ϑ'_2 be equal to $(\varsigma'_2, \text{updateStates}(\mathtt{Q}_2, \imath, q'_2))$ and Θ'_2 be equal to $\Theta_2[\imath \mapsto E(\otimes(S_{\neg v}, S_v)[S_{\neg v}])]$. The previous evaluation of S_2 with the monitoring execution state ζ_2, the rule (E$_{\mathcal{M}(O)}$-CONCUR) and the equality between o^p and ϵ imply that $\langle\!\langle \vartheta_2 \Vdash \Theta_2 \rangle\!\rangle \xrightarrow{o^p}_{\mathcal{M}(O)} \langle\!\langle \vartheta'_2 \Vdash \Theta'_2 \rangle\!\rangle$.

Lemma B.2.3, the global hypothesis $\star_1$ and the evaluations of S_1 and S_2 imply that $\zeta'_1 \vdash S'_1$ and $\zeta'_2 \vdash S'_2$.

With $P_1^e = S_1^b = P_2^u = S_v$ and $P_2^e = S_2^b = P_1^u = S_{\neg v}$, S'_1 is equal to $E(\otimes(P_1^e, P_1^u)[S_1^b])$ and S'_2 is equal to $E(\otimes(P_2^e, P_2^u)[S_2^b])$. This is the point (a) of the second definition ${}^{\text{B.1.9}}\triangleleft_2$ of bisimilarity.

The fact that $\zeta'_1 \cong \zeta'_2$ implies that $V'_1 = V'_2$, $W'_1 = W'_2$, $L'_1 = L'_2$ and $(w'_1 \downarrow_E) = (w'_2 \downarrow_E)$. This is the point (b) of the second definition ${}^{\text{B.1.9}}\triangleleft_2$ of bisimilarity.

From the global conclusion $^{B.2.7}\lhd_2$ of lemma B.2.7, $w_1 = (w_1 \downarrow_E)$ and $w'_2 = w_2 \top$. The facts that $\zeta_1 \cong \zeta_2$ and $\zeta'_1 \cong \zeta'_2$ imply that $w_1 = w_2$ and $w'_1 = w'_2$. Therefore, $w'_1 = w_1 \top$ and $w'_2 = w_2 \top$. This is the point (c) of the second definition $^{B.1.9}\lhd_2$ of bisimilarity.

As $\zeta'_1 \cong \zeta'_2$, we have that $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$ and $\forall l \notin L'_1 : l \in \lambda'_1 \Leftrightarrow l \in \lambda'_2$. Those properties are points (d) and (e) of the second definition $^{B.1.9}\lhd_2$ of bisimilarity.

As $P^e_1 = P^u_2$ and $P^e_2 = P^u_1$, it is trivial to show that point (f) of the second definition $^{B.1.9}\lhd_2$ of bisimilarity holds.

Therefore,

$$(\zeta'_1 \vdash \Theta'_1(\iota)) \sim (\zeta'_2 \vdash \Theta'_2(\iota))$$

and, following a proof similar to the one used in the sub-case 1.a,

$$(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$$

Case 2: The conclusion $^{B.1.9}\lhd_2$ of definition B.1.9 holds.

This conclusion states that there exist an evaluation context E, four programs $P^{\mathrm{e}}_1, P^{\mathrm{u}}_1, P^{\mathrm{e}}_2, P^{\mathrm{u}}_2$ and two monitored statements S^{b}_1 and S^{b}_2 such that all the following is true:

1. $S_1 = E(\otimes(P^{\mathrm{e}}_1, P^{\mathrm{u}}_1)[S^{\mathrm{b}}_1])$ and $S_2 = E(\otimes(P^{\mathrm{e}}_2, P^{\mathrm{u}}_2)[S^{\mathrm{b}}_2])$,
2. $V_1 = V_2$, $W_1 = W_2$, $L_1 = L_2$ and $(w_1 \downarrow_E) = (w_2 \downarrow_E)$,
3. $\forall i \in \{1, 2\} : \exists w^s_i$ s.t. $w_i = (w_i \downarrow_E) \top w^s_i$,
4. $\forall x \notin V_1 : \sigma_1(x) = \sigma_2(x)$,
5. $\forall l \notin L_1 : l \in \lambda_1 \Leftrightarrow l \in \lambda_2$,
6. $\forall L : \neg\mathit{stoppable}(P^{\mathrm{e}}_1) \wedge \neg\mathit{stoppable}(P^{\mathrm{u}}_1) \Leftrightarrow \neg\mathit{stoppable}(P^{\mathrm{e}}_2) \wedge \neg\mathit{stoppable}(P^{\mathrm{u}}_2)$.

The point 3 stated above and lemma B.2.5 imply that o^p equals ϵ.

Sub-case 2.a: $S^b_1 \neq \emptyset$.

Equation (B.4), the conclusion $^{B.1.9}\lhd_2$ of definition B.1.9 and lemma B.2.1 imply that

$$\langle\!\langle \zeta_1 \Vdash \otimes(P^{\mathrm{e}}_1, P^{\mathrm{u}}_1)[S^b_1] \rangle\!\rangle \xrightarrow{\epsilon}_{\mathcal{M}(O)} \langle\!\langle (\varsigma'_1, q'_1) \Vdash E^{-1}(S'_1) \rangle\!\rangle$$

Because of the sub-case hypothesis, the only possible rule for this evaluation step is ($\mathrm{E}_{\mathrm{M(O)}}$-$\mathrm{BRANCH}_{\mathrm{step}}$). Therefore,

$$\langle\!\langle \zeta_1 \Vdash S^b_1 \rangle\!\rangle \xrightarrow{\epsilon}_{\mathcal{M}(O)} \langle\!\langle (\varsigma'_1, q'_1) \Vdash S^n_1 \rangle\!\rangle \quad \text{and} \quad S'_1 = E(\otimes(P^{\mathrm{e}}_1, P^{\mathrm{u}}_1)[S^n_1])$$

The following steps consist in proving that $(\zeta'_1 \vdash S'_1) \sim (\zeta_2 \vdash S_2)$.

The case hypothesis, definition B.1.4 and lemma B.2.5 imply:

$$V_1' = V_1, W_1' = W_1 \text{ and } L_1' = L_1 \tag{B.9}$$
$$\forall x \notin \mathit{modified}(S_1) : \sigma_1'(x) = \sigma_1(x) \tag{B.10}$$
$$\forall l \notin \mathit{NLocks}(S_1) \cup \mathit{OLocks}(S_1) : l \in \lambda_1' \Leftrightarrow l \in \lambda_1 \tag{B.11}$$

The first point of the definition of bisimilarity is trivial as $S_1' = E(\otimes(P_1^{\mathrm{e}}, P_1^{\mathrm{u}})[S_1^n])$ and $S_2 = E(\otimes(P_2^{\mathrm{e}}, P_2^{\mathrm{u}})[S_2^b])$.

The case hypothesis combined with (B.9) imply

$$V_1' = V_2, W_1' = W_2 \text{ and } L_1' = L_2$$

Lemma B.2.2 imply $(w_1' \downarrow_E) = (w_1 \downarrow_E)$. Therefore, the case 2 hypothesis "$(w_1 \downarrow_E) = (w_2 \downarrow_E)$" implies $(w_1' \downarrow_E) = (w_2 \downarrow_E)$. This is the point (b) of the bisimilarity definition.

Let E' be $E(\otimes(P_1^{\mathrm{e}}, P_1^{\mathrm{u}})[\square])$. S_1 is equal to $E'(S_1^b)$. As the case hypothesis imply $w_1 = (w_1 \downarrow_E) \top w_1^s$, by definition $(w_1 \downarrow_{E'}) = (w_1 \downarrow_E) \top$. Lemma B.2.2 imply $(w_1 \downarrow_{E'}) = (w_1' \downarrow_{E'})$. By definition of branching context of evaluation context, there exists $w_1^{s'}$ such that $w_1' = (w_1' \downarrow_{E'}) \, w_1^{s'}$. Therefore, there exists $w_1^{s'}$ such that $w_1' = (w_1' \downarrow_E) \top w_1^{s'}$. This is sufficient to trivially prove the point (c) of the bisimilarity definition.

Equation (B.9) implies that all x not in V_1' is not in V_1. Definition B.1.4 implies that all x not belonging to V_1 does not belong to $\mathit{modified}(S_1)$. Therefore, (B.10) implies the point (d) of the bisimilarity definition.

Similarly, (B.11) implies the point (e) of the bisimilarity definition.

Point (f) of the bisimilarity definition follows directly from the case hypothesis.

Therefore,

$$(\zeta_1' \vdash \Theta_1'(\iota)) \sim (\zeta_2 \vdash \Theta_2(\iota))$$

and, following a proof similar to the one used in the sub-case 1.a,

$$(\vartheta_1' \vdash \Theta_1') \sim (\vartheta_2' \vdash \Theta_2')$$

Sub-case 2.b: $S_1^b = \emptyset$.

Equation (B.4), the conclusion ${}^{B.1.9}\triangleleft_2$ of definition B.1.9 and lemma B.2.1 imply that

$$\langle\!| \zeta_1 \Vdash \otimes(P_1^{\mathrm{e}}, P_1^{\mathrm{u}})[S_1^b] |\!\rangle \xrightarrow{\epsilon}_{M(O)} \langle\!| (\varsigma_1', q_1') \Vdash E^{-1}(S_1') |\!\rangle$$

Because of the sub-case hypothesis, the only possible rule for this evaluation step is ($\mathrm{E}_{\mathrm{M(O)}}$-BRANCH$_{\mathrm{exit}}$). Therefore, with $\tilde{v} = \mathit{modified}(P_1^{\mathrm{e}}) \cup \mathit{modified}(P_1^{\mathrm{u}})$ and $\tilde{l} = \mathit{NLocks}(P_1^{\mathrm{e}}) \cup$

$NLocks(P_1^{\mathrm{u}})$,

$$S_1' = E(\emptyset) \quad \text{and} \quad \varsigma_1' = \varsigma_1 \tag{B.12}$$
$$\neg stoppable(P_1^{\mathrm{e}}) \quad \text{and} \quad \neg stoppable(P_1^{\mathrm{u}}) \tag{B.13}$$
$$V_1' = V_1\,, \quad W_1' = W_1 \setminus \tilde{v}\,, \quad L_1' = L_1 \setminus \tilde{l} \quad \text{and} \quad w_1' = (w_1 \downarrow_E) \tag{B.14}$$

Equation (B.13), the case hypothesis and definition B.1.4 imply $\neg stoppable(S_2^{\mathrm{b}})$. The case hypothesis and definition B.1.4 imply $\forall l \in (NLocks(S_2^{\mathrm{b}}) \setminus OLocks(S_2^{\mathrm{b}})) : l \notin \lambda_2$. Therefore, lemma B.2.6 implies that there exists a monitored execution state ζ_2^{t} such that:

$$\langle\!| \zeta_2 \vdash S_2 |\!\rangle \xrightarrow{\epsilon}{}^{\star}_{\mathcal{M}(O)} \langle\!| \zeta_2^{\mathrm{t}} \vdash E(\otimes(P_2^{\mathrm{e}}, P_2^{\mathrm{u}})[\emptyset]) |\!\rangle$$

Let E' be $E(\otimes(P_2^{\mathrm{e}}, P_2^{\mathrm{u}})[\square])$. The previous derivation, the case hypothesis and lemma B.2.5 imply

$$\begin{array}{l}
V_2^{\mathrm{t}} = V_2\,, \quad W_2^{\mathrm{t}} = W_2\,, \quad L_2^{\mathrm{t}} = L_2 \quad \text{and} \quad (w_2^{\mathrm{t}} \downarrow_{E'}) = (w_2 \downarrow_{E'}),\\
\forall x \notin modified(S_2) : \sigma_2^{\mathrm{t}}(x) = \sigma_2(x),\\
\forall l \notin NLocks(S_2) \cup OLocks(S_2) : l \in \lambda_2^{\mathrm{t}} \Leftrightarrow l \in \lambda_2.
\end{array}$$

Therefore, the case hypothesis implies:

$$\begin{array}{l}
V_1^{\mathrm{t}} = V_2^{\mathrm{t}}\,, \quad W_1^{\mathrm{t}} = W_2^{\mathrm{t}}\,, \quad L_1^{\mathrm{t}} = L_2^{\mathrm{t}} \quad \text{and} \quad (w_1^{\mathrm{t}} \downarrow_E) = (w_2^{\mathrm{t}} \downarrow_E)\\
\forall i \in \{1,2\} : \exists\, w_i^{\mathrm{t}} \text{ s.t. } w_i^{\mathrm{t}} = (w_i^{\mathrm{t}} \downarrow_E) \top w_i^{\mathrm{t}}\\
\forall x \notin V_1^{\mathrm{t}} : \sigma_1^{\mathrm{t}}(x) = \sigma_2^{\mathrm{t}}(x)\\
\forall l \notin L_1^{\mathrm{t}} : l \in \lambda_1^{\mathrm{t}} \Leftrightarrow l \in \lambda_2^{\mathrm{t}}
\end{array}$$

Then, by applying the rule ($\mathrm{E}_{\mathcal{M}(O)}$-BRANCH$_{\mathrm{exit}}$) to $E(\otimes(P_2^{\mathrm{e}}, P_2^{\mathrm{u}})[\emptyset])$, we get that there exists ζ_2' such that

$$\langle\!| \zeta_2 \vdash S_2 |\!\rangle \xrightarrow{\epsilon}{}^{\star}_{\mathcal{M}(O)} \langle\!| \zeta_2' \vdash S_1' |\!\rangle$$

and

$$\begin{array}{l}
V_1' = V_2'\,, \quad W_1' = W_2'\,, \quad L_1' = L_2' \quad \text{and} \quad w_1' = w_2'\\
\forall x \notin V_1' : \sigma_1'(x) = \sigma_2'(x)\\
\forall l \notin L_1' : l \in \lambda_1' \Leftrightarrow l \in \lambda_2'
\end{array}$$

Therefore $S_1' = S_2'$ and $\zeta_1' \cong \zeta_2'$.

Following a proof similar to the one used in the sub-case 1.a, we can conclude that:

$$(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$$

B.3 Transparency

Lemma B.3.1 (Confinement).
For all monitored statements S and typing environments γ, if "$\gamma \vdash_E S : H\,cmd$" then $\forall x \in modified(S).\ \gamma(x) = H$, $NLocks(S) = \emptyset$ and $\neg stoppable(S)$.

Proof. The proof goes by induction on the derivation tree of the typing judgment and follows directly from the definitions of the different functions.

Lemma B.3.2 (To an unmonitored execution step of a well-typed program corresponds a monitored step).

For all monitored statements S, execution statements S'_u, execution states (σ, λ) and (σ', λ'), automaton states (V, W, L, w), typing environments γ, and types τ if:

$\star_1$ $((\sigma, \lambda), (V, W, L, w)) \vdash S$,

$\star_2$ $\forall x \in V, \gamma(x) = H$,

$\star_3$ $L = \emptyset$,

$\star_4$ $w \notin \{\bot\}^\star \Rightarrow \tau = H$,

$\star_5$ $\gamma \vdash_E S : \tau\ cmd$,

$\star_6$ $\langle\!\!| (\sigma, \lambda) \vdash strip(S)^1 |\!\!\rangle \xrightarrow{o}_O \langle\!\!| (\sigma', \lambda') \vdash S'_u |\!\!\rangle$,

then there exists an automaton state (V', W', L', w'), a monitored statement S'_m, and a type τ' such that:

- $\langle\!\!| ((\sigma, \lambda), (V, W, L, w)) \Vdash S |\!\!\rangle \overset{o\ \star}{\longrightarrow}_{M(O)} \langle\!\!| ((\sigma', \lambda'), (V', W', L', w')) \Vdash S'_m |\!\!\rangle$,
- $\forall x \in V', \gamma(x) = H$,
- $L' = \emptyset$,
- $w' \notin \{\bot\}^\star \Rightarrow \tau' = H$,
- $\gamma \vdash_E S'_m : \tau'\ cmd$,
- $S'_u = strip(S'_m)$.

Proof. The proof goes by induction on the derivation tree of the global hypothesis $\star_5$. If the last typing rule used is:

[1] strip(S) is defined on page 151.

(T$_E$-NOC) then we can conclude that the lemma is vacuously true as the global hypothesis $\star_6$ does not hold.

(T$_E$-SKIP) then we can conclude that the lemma is true as the monitored or unmonitored execution of **skip** only replace it by $\emptyset$.

(T$_E$-ASSIGN) then we can conclude that :

(1) S is "x **:=** e".

This follows directly from the typing rule.

(2) there exists an automaton state (V', W', L', w') such that $W' = W$, $L' = L$, $w' = w$, and:

$$\langle\!|\, ((\sigma, \lambda), (V, W, L, w)) \Vdash S \,|\!\rangle \xrightarrow{\epsilon}{}^{\star}_{\mathcal{M}(O)} \langle\!|\, ((\sigma', \lambda'), (V', W', L', w')) \Vdash \emptyset \,|\!\rangle$$

This follows directly from the rule (E$_{\mathcal{M}(O)}$-OK) as the security automaton always validate the execution of assignments.

(3) $\forall y \in V', \gamma(y) = H$.

Case 1: $x \notin W$ and $FV(e) \cap V = \emptyset$.

(a) $V' \subseteq V$.

This follows directly from the case hypothesis and the rule (T-ASSIGN-pub).

($\circ$) $\forall y \in V', \gamma(y) = H$.

This follows from the local conclusion (a) and the global hypothesis $\star_2$.

Case 2: $x \in W$.

(a) $x \in V$.

This follows directly from the case hypothesis and the global hypothesis $\star_1$.

($\circ$) $\forall y \in V', \gamma(y) = H$.

This follows from the local conclusion (a) and the global hypothesis $\star_2$.

Case 3: $FV(e) \cap V \neq \emptyset$.

(a) $V' = V \cup \{x\}$.

This follows directly from the case hypothesis and the rule (T-ASSIGN-sec).

(b) $\gamma(x) = H$.

Because of the case hypothesis, there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis $\star_2$ implies that $\gamma(z) = H$. Then, the global hypothesis $\star_5$ and the typing rule (T$_E$-ASSIGN) imply that $\gamma(x) = H$.

($\circ$) $\forall y \in V', \gamma(y) = H$.

This follows from the global hypothesis $\star_2$ and the local conclusions (a) and (b).

($\bullet$) the lemma holds.

If follows directly from the local conclusions (2) and (3), the global hypotheses $\star_3$ and $\star_4$, and the typing rule (T$_E$-NOC).

(T_E-OUTPUT) then we can conclude that :

(1) S is "**output** e", $\gamma(e) = L$, and $\tau = L$.
This follows directly from the rule (T_E-OUTPUT).

(2) $w \in \{\bot\}^\star$ and $FV(e) \cap V = \emptyset$.
The global hypothesis $\star_4$ and the local conclusion (1) imply $w \in \{\bot\}^\star$. The global hypothesis $\star_2$ and the local conclusion (1) imply $FV(e) \cap V = \emptyset$.

(3) the following is true:

$$\langle\!\langle ((\sigma, \lambda), (V, W, L, w)) \Vdash S \rangle\!\rangle \overset{o}{\rightarrow}{}^{\star}_{\mathcal{M}(O)} \langle\!\langle ((\sigma', \lambda'), (V, W, L, w)) \Vdash \emptyset \rangle\!\rangle$$

This follows directly from the rule ($E_{\mathcal{M}(O)}$-OK) and the local conclusion (2).

(•) the lemma holds.
If follows directly from the local conclusions (3), the global hypotheses $\star_2$, $\star_3$ and $\star_4$, and the typing rule (T_E-NOC).

(T_E-SEQ) then we can conclude that :

(1) S is "S^{h} ; S^{t}" and:
- $\gamma \vdash S^{\mathrm{h}} : \tau$ cmd,
- $\gamma \vdash S^{\mathrm{t}} : \tau$ cmd.

This follows directly from the rule (T_E-SEQ).

(•) the lemma holds.
From the inductive hypothesis, the lemma holds for S^{h}. Therefore, from the rules for monitored and unmonitored executions, both executions evaluate the same way.

(T_E-IF) then we can conclude that :

(1) S is "**if** e **then** S^{true} **else** S^{false} **end**" and there exists a type τ' such that $\gamma(e) \leq \tau'$ and:
- $\gamma \vdash S^{true} : \tau'$ cmd,
- $\gamma \vdash S^{false} : \tau'$ cmd.

This follows directly from the rule (T_E-IF).

(2) $FV(e) \cap V = \emptyset$ or $NLocks(S^{\mathrm{true}}) \cup NLocks(S^{\mathrm{false}}) = \emptyset$.
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis $\star_2$ implies that $\gamma(z) = H$. Then $\gamma(e) = H$. Therefore, lemma B.3.1 and the local conclusion (1) imply $NLocks(S^{\mathrm{true}}) = NLocks(S^{\mathrm{false}}) = \emptyset$

(3) $FV(e) \cap V \neq \emptyset$ implies that both S^{true} and S^{false} can be typed as H cmd with the typing environment γ.
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis $\star_2$ implies that $\gamma(z) = H$. Then $\gamma(e) = H$ and the desired result follows from the local conclusion (1).

(•) the lemma holds.
From the local conclusion (2), one of the two transitions (T-BRANCH-low) or (T-BRANCH-high) applies. Therefore, the monitored execution evaluates similarly to the unmonitored execution. Additionally, lemma B.3.1, the global hypotheses and the local conclusion (3) imply the remaining global conclusions.

($\mathbf{T_E}$-WHILE) then we can conclude that :

(1) S is "**while** e **do** S^1 **done**", $\gamma(e) = L$, $\tau = L$ and $\gamma \vdash S^1 : L\,\mathrm{cmd}$. This follows directly from the rule ($\mathrm{T_E}$-WHILE).

(2) $FV(e) \cap V = \emptyset$.
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis $\star_2$ implies that $\gamma(z) = H$. Then $\gamma(e) = H$, which is in contradiction with the local conclusion (1).

(•) the lemma holds.
From the local conclusion (2), the transition (T-BRANCH-low) applies. Therefore, the monitored execution evaluates similarly to the unmonitored execution. Additionally, the global hypotheses and local conclusion (1) imply the remaining global conclusions.

($\mathbf{T_E}$-SYNC) then we can conclude that :

(1) S is "**with** $\bar{x}$ **when** e **do** S^s **done**", $\gamma(e) = L$, $\tau = L$ and $\gamma \vdash S^s : L\,\mathrm{cmd}$. This follows directly from the rule ($\mathrm{T_E}$-SYNC).

(2) $FV(e) \cap V = \emptyset$.
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis $\star_2$ implies that $\gamma(z) = H$. Then $\gamma(e) = H$, which is in contradiction with the local conclusion (1).

(•) the lemma holds.
From the local conclusion (2), the transition (T-SYNC) applies. Therefore, the monitored execution evaluates similarly to the unmonitored execution. Additionally, the global hypotheses and local conclusion (1) imply the remaining global conclusions.

($\mathbf{T_E}$-LOCKED) then we can conclude that :

(1) S is "$\odot\bar{x}[S^s]$", $\tau = L$ and $\gamma \vdash S^s : L\,\mathrm{cmd}$. This follows directly from the rule ($\mathrm{T_E}$-SYNC).

(•) the lemma holds.
From the rules of both semantics for locked statements, the monitored execution evaluates similarly to the unmonitored execution. Additionally, the global hypotheses and local conclusion (1) imply the remaining global conclusions.

($\mathbf{T_E}$-BRANCHED) then we can conclude that :

(1) S is "$\otimes(P^e, P^u)[S^b]$" and there exist a type τ' such that:

- $\tau \leq \tau'$

- $\gamma \vdash S^{\mathrm{b}} : \tau'$ cmd.

  This follows directly from the rule ($\mathrm{T_E}$-SYNC).

(2) strip(S) = strip(S^{b}).

This follows directly from the definition of strip().

(3) $S^{\mathrm{b}} \neq \emptyset$.

Otherwise, the global hypothesis $\star_6$ does not hold.

(•) the lemma holds.

From the local conclusions (1) and (2), the lemma holds for S^{b}, and from the local conclusion (3), rule ($\mathrm{E}_{\mathcal{M}(O)}$-BRANCH$_{\mathrm{step}}$) applies to S. Therefore, the lemma holds.

Lemma B.3.3 (To a monitored execution step of a well-typed program corresponds an unmonitored step).

For all monitored statements S and S', execution states (σ, λ) and (σ', λ'), automaton states (V, W, L, w) and (V', W', L', w'), typing environments γ, and types τ if:

$\star_1$ $((\sigma, \lambda), (V, W, L, w)) \vdash S$,

$\star_2$ $\forall x \in V, \gamma(x) = H$,

$\star_3$ $L = \emptyset$,

$\star_4$ $w \notin \{\bot\}^\star \;\Rightarrow\; \tau = H$,

$\star_5$ $\gamma \;\vdash_E\; S \;:\; \tau$ *cmd,*

$\star_6$ $\Lbag ((\sigma, \lambda), (V, W, L, w)) \Vdash S \Rbag \xrightarrow{o}_{\mathcal{M}(O)} \Lbag ((\sigma', \lambda'), (V', W', L', w')) \Vdash S' \Rbag$,

then there exists a type τ' such that:

- $\Lbag (\sigma, \lambda) \vdash strip(S) \Rbag \overset{o}{\rightarrow}{}^{\star}_{O} \Lbag (\sigma', \lambda') \vdash strip(S') \Rbag$,
- $\forall x \in V', \gamma(x) = H$,
- $L' = \emptyset$,
- $w' \notin \{\bot\}^\star \;\Rightarrow\; \tau' = H$,
- $\gamma \;\vdash_E\; S' \;:\; \tau'$ *cmd.*

Proof. The proof is similar to lemma B.3.2; however more simple.

Appendix C

Proofs of Chapter 5's Theorems

Some of the proofs in this chapter use a semantic rule for the evaluation of expressions. This semantic rule is defined below.

Definition C.0.1 (Semantics rule for expression evaluation). *For all value store σ, tag store ρ, and expression e:*

$$\sigma;\rho \;\vdash\; e \;\Downarrow\; \sigma(e) \;:\; \rho(e)$$

C.1 Usable Analyses

C.1.1 Hypothesis 5.2.1

Lemma C.1.1 (Constraints in Figure 5.5 enforce the Hypothesis 5.2.1).
For all value stores σ, σ_i and σ_o, tag stores ρ, ρ_i and ρ_o, program counter tag t^{pc}, statement S, and analysis result $(\mathfrak{D}, \mathfrak{X})$ such that:

$\star_1$ $(\mathfrak{D}, \mathfrak{X}) \models (\sigma, \rho \vdash S)$,

$\star_2$ $\forall x : (\rho(x) = \bot) \Rightarrow \sigma_i(x) = \sigma(x)$,

$\star_3$ $(\sigma_i,\ \rho_i),\ t^{pc}\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$

it is true that:

- $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$

Proof. The proof goes by induction on the derivation tree of " $(\sigma_i,\ \rho_i),\ t^{\mathrm{pc}}\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ ". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

$(\mathbf{E-SKIP})$ then we can conclude that :

(1) S is "**skip**" and $\sigma o = \sigma_i$.
This follows directly from the definition of the rule $(\mathbf{E-SKIP})$.

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.
This follows directly from the local conclusion (1).

(E − ASSIGN) then we can conclude that :

(1) S is "id **:=** e" and $\forall x \neq id.\sigma o(x) = \sigma_i(x)$.
This follows directly from the definition of the rule (**E − ASSIGN**).

(2) $id \in \mathfrak{X}$.
If follows from the global hypothesis $\star_1$ and the local conclusion (1).

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.
This follows directly from the local conclusions (1) and (2).

(E − SEQUENCE) then we can conclude that :

(1) S is "S_1 ; S_2" and there exist a value store σ_1, a tag store ρ_1, and two identifiers sets X_1 and X_2 such that:

- $(\sigma_i,\ \rho_i),\ t^{\text{pc}}\ \vdash\ S_1\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_1,\ \rho_1)$ is a sub-derivation tree of "S_1 ; S_2"
- $(\sigma_1,\ \rho_1),\ t^{\text{pc}}\ \vdash\ S_2\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ is a sub-derivation tree of "S_1 ; S_2"

This follows directly from the definition of the rule (**E − SEQUENCE**).

(2) There exists two analysis results $(\mathfrak{D}_1, \mathfrak{X}_1)$ and $(\mathfrak{D}_2, \mathfrak{X}_2)$ such that:

- $(\mathfrak{D}_1, \mathfrak{X}_1) \models (\sigma, \rho \vdash S_1)$
- $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho' \vdash S_2)$ with $\rho' = \rho\ \sqcup\ ((\mathfrak{X}_1 \times \top)\ \cup\ ((\mathfrak{X}_1)^c \times \bot))$
- $\mathfrak{X}_1 \cup \mathfrak{X}_2 \subseteq \mathfrak{X}$

This follows from the global hypothesis $\star_1$, the local conclusion (1) and the definition of the relation $\models$.

(3) $\forall x \notin \mathfrak{X}_1.\ \sigma_1(x) = \sigma_i(x)$.
This result is obtained by applying the inductive hypothesis on the derivation "$(\sigma_i,\ \rho_i),\ t^{\text{pc}}\ \vdash\ S_1\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_1,\ \rho_1)$" (sub-derivation tree of "$(\sigma_i,\ \rho_i),\ t^{\text{pc}}\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$") using the fact that "$(\mathfrak{D}_1, \mathfrak{X}_1) \models (\sigma, \rho \vdash S_1)$" (from the local conclusion (2)).

(4) $\forall x : (\rho'(x) = \bot) \Rightarrow \sigma_1(x) = \sigma(x)$.
For all variable x, from the definition of ρ' in the local conclusion (2) and the local conclusion (3), we can conclude that "$\rho'(x) = \bot$" implies "$\rho(x) = \bot$" and "$\sigma_1(x) = \sigma_i(x)$". Using the global hypothesis $\star_2$ and the fact that "$\rho(x) = \bot$", it is possible to show that "$\sigma_i(x) = \sigma(x)$". Hence, we can conclude that "$\rho'(x) = \bot$" implies "$\sigma_1(x) = \sigma(x)$".

(5) $\forall x \notin \mathfrak{X}_2.\ \sigma_o(x) = \sigma_1(x)$.
This result is obtained by applying the inductive hypothesis on the derivation "$(\sigma_1,\ \rho_1),\ t^{\text{pc}}\ \vdash\ S_2\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$" (sub-derivation tree of "$(\sigma_i,\ \rho_i),\ t^{\text{pc}}\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$") using the fact that "$(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho' \vdash S_2)$" (from the local conclusion (2)). The inductive hypothesis can be applied thanks to the local conclusion (4).

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.

This follows directly from the local conclusions (2), (3) and (5).

$(\mathbf{E-IF_\perp})$ then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exists a value $v \in \{true, false\}$ such that:

- $\sigma_i(e) = v$ and $\rho_i(e) = \perp$,
- " $(\sigma_i,\ \rho_i),\ t^{\mathrm{pc}}\ \vdash\ S_v\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ " is a sub-derivation tree of " $(\sigma_i,\ \rho_i),\ t^{\mathrm{pc}}\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ "

This follows directly from the definition of the rule $(\mathbf{E-IF_\perp})$.

(2) There exist $\mathfrak{X}_{true}$ and $\mathfrak{X}_{false}$ such that:

- $(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$,
- $\mathfrak{X} \supseteq \llbracket \mathfrak{X}_{true}, \mathfrak{X}_{false} \rrbracket^{\rho(e)}_{\sigma(e)}$

This follows from the global hypothesis $\star_1$, the local conclusions (1), and the definition of $\models$.

(3) $\forall x \notin \mathfrak{X}_v.\ \sigma_o(x) = \sigma_i(x)$.

This result is obtained by applying the inductive hypothesis on "$(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$" and "$(\sigma_i,\ \rho_i),\ t^{\mathrm{pc}}\ \vdash\ S_v\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$".

(4) $\mathfrak{X}_v \subseteq \mathfrak{X}$.

This follows directly from the local conclusions (2) and (1).

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.

This follows directly from the local conclusions (3) and (4).

$(\mathbf{E-IF_\top})$ then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exist a value $v \in \{true, false\}$ and a tag store ρ' such that:

- $\sigma_i(e) = v$ and $\rho_i(e) = \top$,
- " $(\sigma_i,\ \rho_i),\ \top\ \vdash\ S_v\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho')$ " is a sub-derivation tree of " $(\sigma_i,\ \rho_i),\ t^{\mathrm{pc}}\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ "

This follows directly from the definition of the rule $(\mathbf{E-IF_\top})$.

(2) There exist $\mathfrak{X}_{true}$ and $\mathfrak{X}_{false}$ such that:

- $(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$,
- $\mathfrak{X} \supseteq \llbracket \mathfrak{X}_{true}, \mathfrak{X}_{false} \rrbracket^{\rho(e)}_{\sigma(e)}$

This follows from the global hypothesis $\star_1$, the local conclusions (1), and the definition of $\models$.

(3) $\forall x \notin \mathfrak{X}_v.\ \sigma_o(x) = \sigma_i(x)$.

This result is obtained by applying the inductive hypothesis on "$(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$" and "$(\sigma_i,\ \rho_i),\ \top\ \vdash\ S_v\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho')$".

(4) $\mathfrak{X}_v \subseteq \mathfrak{X}$.
This follows directly from the local conclusions (2) and (1).

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.
This follows directly from the local conclusions (3) and (4).

(**E − WHILE$_{\text{skip}}$**) then we can conclude that :

(1) $\sigma_o = \sigma_i$.
This follows directly from the definition of the rule (**E − WHILE$_{\text{skip}}$**).

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.
This follows directly from the local conclusion (1).

(**E − WHILE$_{\text{true}_\perp}$**) then we can conclude that :

(1) S is "**while** e **do** S_l **done**" and there exist a tag t_e and a tag store ρ_t:

- $\sigma_i(e) = \texttt{true}$ and $\rho_i(e) = t_e$
- " $(\sigma_i,\ \rho_i),\ t^{\text{pc}} \sqcup t_e\ \vdash\ S_l$; **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_t)$ " is a sub-derivation tree of " $(\sigma_i,\ \rho_i),\ t^{\text{pc}}\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ "

This follows directly from the definition of the rule (**E − WHILE$_{\text{true}}$**).

(2) There exist an analysis result $(\mathfrak{D}_l, \mathfrak{X}_l)$ such that:

- $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho' \vdash S_l)$ with $\rho' = \rho\ \sqcup\ ((\mathfrak{X}_l \times \top)\ \cup\ ((\mathfrak{X}_l)^c \times \perp))$,
- $\mathfrak{X} = \mathfrak{X}_l$,

This follows from the global hypotheses $\star_1$ and $\star_2$, the local conclusions (1) (S is "**while** e **do** S_l **done**" and $\sigma_i(e) = \texttt{true}$), and the definition of $\models$.

(3) There exist a value store σ_1 and a tag store ρ_1 such that:

- " $(\sigma_i,\ \rho_i),\ t^{\text{pc}} \sqcup t_e\ \vdash\ S_l\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_1,\ \rho_1)$ " is a sub-derivation tree of " $(\sigma_i,\ \rho_i),\ t^{\text{pc}} \sqcup t_e\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ ",
- " $(\sigma_1,\ \rho_1),\ t^{\text{pc}} \sqcup t_e\ \vdash$ **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_t)$ " is a sub-derivation tree of " $(\sigma_i,\ \rho_i),\ t^{\text{pc}} \sqcup t_e\ \vdash\ S\ \Downarrow_{\mathcal{M}(O)}\ (\sigma_o,\ \rho_o)$ ",

This follows directly from the local conclusion (1) and the definition of the rule (**E − SEQUENCE**).

(4) $\forall x : (\rho'(x) = \perp) \Rightarrow (\sigma_i(x) = \sigma(x))$.
From the definition of ρ' (given in the local conclusion (2)), $\rho'(x) = \perp$ implies $\rho(x) = \perp$. Hence, the global hypothesis $\star_2$ implies the above result.

(5) $\forall x \notin \mathfrak{X}_l.\ \sigma_1(x) = \sigma_i(x)$.
This result follows from the inductive hypothesis applied on "$(\sigma_i,\ \rho_i),\ t^{\text{pc}} \sqcup t_e\ \vdash\ S_l\ \Downarrow_{\mathcal{M}(O)}$ $(\sigma_1,\ \rho_1)$" (given in the local conclusion (3)) and $(\mathfrak{D}_l, \mathfrak{X}_l)\ \models\ (\sigma, \rho'\ \vdash\ S_l)$ (given in the local conclusion (2)). The inductive hypothesis can be applied because of the local conclusion (4).

Let us define ρ_l and ρ'_l such that:

$\diamond_1$ $\forall x \in \mathfrak{X}_l, \rho_l(x) = \top$,

$\diamond_2$ $\forall x \notin \mathfrak{X}_l, \rho_l(x) = \rho(x)$,

$\diamond_3$ $\rho'_l = \rho_l \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$.

Let us define $\mathfrak{D}_t$ to be $\{\!\![\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d), \mathcal{I}d]\!\!\}^{\rho_l(e)}_{\sigma(e)}$.

(6) $\rho'_l = \rho'$.
This follows from the definitions of ρ'_l and ρ' (given in $\diamond_3$ and (2)). For all x in $\mathfrak{X}_l$, $\rho'_l(x) = \top = \rho'(x)$. For all x not in $\mathfrak{X}_l$, from $\diamond_2$, $\rho_l(x) = \rho(x)$; hence, $\rho'_l(x) = \rho_l(x) = \rho(x) = \rho'(x)$.

(7) $(\mathfrak{D}_t, \mathfrak{X}_l) \models (\sigma, \rho_l \vdash$ **while** e **do** S_l **done**$)$.
From the local conclusions (2) and (6), we can conclude that $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho'_l \vdash S_l)$. From the local definition $\diamond_3$, $\rho'_l = \rho_l \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$. The local definitions $\diamond_1$ and $\diamond_2$ and the global hypothesis $\star_2$ imply $\rho_l(e) \neq \top \Rightarrow \rho(e) \neq \top \Rightarrow \sigma(e) = \sigma_i(e)$; therefore, as $\sigma_i(e) = \texttt{true}$, $\mathfrak{X}_l = \{\!\![\mathfrak{X}_l, \emptyset]\!\!\}^{\rho(e)}_{\sigma(e)}$. Hence, from the definition of $\models$ for while-statements, $(\mathfrak{D}_t, \mathfrak{X}_l) \models (\sigma, \rho_l \vdash$ **while** e **do** S_l **done**$)$.

(8) $\forall x : (\rho(x) = \bot) \Rightarrow (\sigma_1(x) = \sigma(x))$.
From $\diamond_1$, $\rho_l(x) = \bot$ implies $x \notin \mathfrak{X}_l$. Hence, local conclusion (5) implies $\sigma_1(x) = \sigma_i(x)$. From the property "$\rho_l(x) = \bot$ implies $x \notin \mathfrak{X}_l$", the local definition $\diamond_2$, and the global hypothesis $\star_2$, it is possible to show that $\rho_l(x) = \bot$ implies $\sigma_i(x) = \sigma(x)$. Hence, $\rho_l(x) = \bot$ implies $\sigma_1(x) = \sigma(x)$.

(9) $\forall x \notin \mathfrak{X}_l.\ \sigma_o(x) = \sigma_1(x)$.
The inductive hypothesis, using the local conclusions (7), (8), and (3), implies the desired result.

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.
From the local conclusions (9) and (5), $\forall x \notin \mathfrak{X}_l.\ \sigma_o(x) = \sigma_i(x)$. The fact $\mathfrak{X} = \mathfrak{X}_l$ of the local conclusion (2) completes the proof.

(E − WHILE$_{\mathbf{true}_\top}$) then we can conclude that :

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.
The proof goes exactly the same way as for **(E − WHILE$_{\mathbf{true}_\top}$)**.

(E − WHILE$_{\mathbf{false}_\top}$) then we can conclude that :

(•) $\forall x \notin \mathfrak{X}.\ \sigma_o(x) = \sigma_i(x)$.
The proof goes exactly the same way as for **(E − WHILE$_{\mathbf{skip}}$)**.

C.1.2 Hypothesis 5.2.2

Lemma C.1.2 (Analysis is tag store monotone for assigned variables).
For all value store σ, tag stores ρ_1 and ρ_2, statement S, and analysis result $(\mathfrak{D}_1, \mathfrak{X}_1)$ such that:

$\star_1$ $(\mathfrak{D}_1, \mathfrak{X}_1) \models (\sigma, \rho_1 \vdash S)$,

$\star_2$ $\rho_2 \sqsubseteq \rho_1$

then there exists an analysis result $(\mathfrak{D}_2, \mathfrak{X}_2)$ *such that:*

- $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho_2 \vdash S)$ *and*
- $\mathfrak{X}_2 \subseteq \mathfrak{X}_1$

Proof. The proof, which goes by structural induction on the statement S, is direct. It mainly relies on the following property. For all tag stores ρ_1, ρ'_1, ρ_2 and ρ'_2 and variable sets $\mathfrak{X}_1$ and $\mathfrak{X}_2$ such that $\rho'_1 = \rho_1 \sqcup ((\mathfrak{X}_1 \times \top) \cup ((\mathfrak{X}_1)^c \times \bot))$ and $\rho'_2 = \rho_2 \sqcup ((\mathfrak{X}_2 \times \top) \cup ((\mathfrak{X}_2)^c \times \bot))$, if $\rho_2 \sqsubseteq \rho_1$ and $\mathfrak{X}_2 \subseteq \mathfrak{X}_1$ then $\rho'_2 \sqsubseteq \rho'_1$.

Lemma C.1.3 (Constraints are high-values independent).
For all value stores σ and σ', tag store ρ, statement S, and analysis result $(\mathfrak{D}, \mathfrak{X})$ *such that:*

$\star_1$ $(\mathfrak{D}, \mathfrak{X}) \models (\sigma, \rho \vdash S)$,

$\star_2$ $\forall x : (\rho(x) = \bot) \Rightarrow (\sigma'(x) = \sigma(x))$

it is true that:

- $(\mathfrak{D}, \mathfrak{X}) \models (\sigma', \rho \vdash S)$

Proof. The proof, which goes by structural induction on the statement S, is direct. It mainly relies on the following property. For all value stores σ and σ' and tag stores ρ, such that $\forall x : (\rho(x) = \bot) \Rightarrow (\sigma'(x) = \sigma(x))$, the following holds: $\llbracket X, Y \rrbracket^{\rho(e)}_{\sigma(e)} = \llbracket X, Y \rrbracket^{\rho(e)}_{\sigma'(e)}$.

Lemma C.1.4 (Constraints in Figure 5.5 enforce the Hypothesis 5.2.2).
For all value store σ, tag store ρ, statement S, analysis result $(\mathfrak{D}, \mathfrak{X})$*, and variable x such that:*

$\star_1$ $\sigma; \rho; \top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$,

$\star_2$ $(\mathfrak{D}, \mathfrak{X}) \models (\sigma, \rho \vdash S)$,

it is true that:

$$\rho_o(x) = \quad \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y) \quad \sqcup \quad ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x)$$

Proof. The proof goes by induction on the derivation tree of " $\sigma; \rho; \top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

(**E − SKIP**) then we can conclude that :

(1) S is "**skip**" and $\rho_o = \rho$.
This follows directly from the definition of the rule (**E − SKIP**).

(2) $\mathfrak{D} = \mathcal{I}d$ and $\mathfrak{X} = \emptyset$.
This follows directly from the global hypothesis $\star_2$ and the definition of $\models$.

(•) $\rho_o = (\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.
This follows from the local conclusions (1) and (2).

(**E − ASSIGN**) then we can conclude that :

(1) S is "id **:=** e" and $\rho_o = \rho[id \mapsto \top]$.
This follows directly from the definition of the rule (**E − ASSIGN**).

(2) $\mathfrak{D} = \mathcal{I}d[id \mapsto FV(e)]$ and $\mathfrak{X} = \{id\}$.
This follows directly from the global hypothesis $\star_2$ and the definition of $\models$.

(•) $\rho_o = (\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.
This follows from the local conclusions (1) and (2).

(**E − SEQUENCE**) then we can conclude that :

(1) S is "S_1 ; S_2" and there exist a value store σ_1 and a tag store ρ_1 such that:

- $\sigma;\rho;\top \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ is a sub-derivation tree of " $\sigma;\rho;\top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- $\sigma_1;\rho_1;\top \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ is a sub-derivation tree of " $\sigma;\rho;\top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule (**E − SEQUENCE**).

(2) There exist $(\mathfrak{D}_1, \mathfrak{X}_1)$ and $(\mathfrak{D}_2, \mathfrak{X}_2)$ such that:

- $(\mathfrak{D}_1, \mathfrak{X}_1) \models (\sigma, \rho \vdash S_1)$
- $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho' \vdash S_2)$ with $\rho' = \rho \ \sqcup \ ((\mathfrak{X}_1 \times \top) \ \cup \ ((\mathfrak{X}_1)^c \times \bot))$
- $\mathfrak{D} = \mathfrak{D}_1 \circ \mathfrak{D}_2$
- $\mathfrak{X} = \mathfrak{X}_1 \cup \mathfrak{X}_2$

This follows from the global hypothesis $\star_2$, the local conclusion (1) and the definition of the relation $\models$.

(3) $\rho_1 = (\lambda x. \bigsqcup_{y \in \mathfrak{D}_1(x)} \rho(y)) \sqcup ((\mathfrak{X}_1 \times \{\top\}) \cup (\mathfrak{X}_1^c \times \{\bot\}))$.
This result is obtained by applying the inductive hypothesis on the derivation " $\sigma;\rho;\top \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " (sub-derivation tree of " $\sigma;\rho;\top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") using the fact that "$(\mathfrak{D}_1, \mathfrak{X}_1) \models (\sigma, \rho \vdash S_1)$" (from the local conclusion (2)).

(4) $\rho' = \rho_1$.
For all x in $\mathfrak{X}_1$, $\rho'(x) = \top = \rho_1(x)$ (it follows from the definitions of ρ' and ρ_1 in the local conclusions (2) and (3)). From the local conclusion (2) and hypothesis 5.2.4, if a variable x is not in $\mathfrak{X}_1$ then $\mathfrak{D}_1(x) = \{x\}$. Hence, from the local conclusion (3), if a variable x is not in $\mathfrak{X}_1$ then $\rho_1(x) = \rho(x)$. For all x not in $\mathfrak{X}_1$, $\rho'(x) = \rho(x)$ (it follows from the definition of ρ' in the local conclusion (2)).

(5) $\forall x : (\rho'(x) = \bot) \Rightarrow (\sigma_1(x) = \sigma(x))$.
From the definition of ρ' in the local conclusion (2), for all variable x if $\rho'(x) = \bot$ then x is not in $\mathfrak{X}_1$. Hence, from the local conclusions (1) and (2) and lemma C.1.1, for all variable x $\rho'(x) = \bot$ implies $\sigma_1(x) = \sigma(x)$.

(6) $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho' \vdash S_2) \Rightarrow (\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma_1, \rho_1 \vdash S_2)$.

This follows directly from the local conclusions (4) and (5) and lemma C.1.3.

(7) $\rho_o = (\lambda x. \bigsqcup_{y \in \mathfrak{D}_2(x)} \rho_1(y)) \sqcup ((\mathfrak{X}_2 \times \{\top\}) \cup (\mathfrak{X}_2^c \times \{\bot\}))$.

This result is obtained by applying the inductive hypothesis on the derivation " $\sigma_1; \rho_1; \top \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (sub-derivation tree of " $\sigma; \rho; \top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") using the fact that "$(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma_1, \rho_1 \vdash S_2)$" (from the local conclusion (2)).

(8) $((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\})) = ((\mathfrak{X}_1 \times \{\top\}) \cup (\mathfrak{X}_1^c \times \{\bot\})) \sqcup ((\mathfrak{X}_2 \times \{\top\}) \cup (\mathfrak{X}_2^c \times \{\bot\}))$.

This follows from the local conclusion (2) and the fact that $\bot \sqsubseteq \top$.

(9) $\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y) = \lambda x. \bigsqcup_{z \in \mathfrak{D}_2(x)} (\bigsqcup_{y \in \mathfrak{D}_1(z)} \rho(y))$.

This follows from the local conclusion (2).

(•) $(\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\})) = \rho_o$.

For all x, let $t(x) = \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x)$, then:

$$\begin{aligned} t(x) = & \bigsqcup_{z \in \mathfrak{D}_2(x)} (\bigsqcup_{y \in \mathfrak{D}_1(z)} \rho(y)) \\ & \sqcup \quad ((\mathfrak{X}_1 \times \{\top\}) \cup (\mathfrak{X}_1^c \times \{\bot\}))(x) \sqcup ((\mathfrak{X}_2 \times \{\top\}) \cup (\mathfrak{X}_2^c \times \{\bot\}))(x) \\ = & \bigsqcup_{z \in \mathfrak{D}_2(x)} (\quad (\bigsqcup_{y \in \mathfrak{D}_1(z)} \rho(y)) \quad \sqcup \quad ((\mathfrak{X}_1 \times \{\top\}) \cup (\mathfrak{X}_1^c \times \{\bot\}))(x) \quad) \\ & \sqcup \quad ((\mathfrak{X}_2 \times \{\top\}) \cup (\mathfrak{X}_2^c \times \{\bot\}))(x) \\ = & \bigsqcup_{z \in \mathfrak{D}_2(x)} (\quad \rho_1(z) \quad) \quad \sqcup \quad ((\mathfrak{X}_2 \times \{\top\}) \cup (\mathfrak{X}_2^c \times \{\bot\}))(x) \\ = & \rho_o(x) \end{aligned} \tag{C.1}$$

The equation (C.1) follows from the local conclusions (9), (8), (3), and (7).

(E − IF$_\bot$) then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exist a value $v \in \{true, false\}$ such that:

- " $\sigma; \rho \vdash e \Downarrow_{\mathcal{M}(O)} v : \bot$ ",
- " $\sigma; \rho; t^{\text{pc}} \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma; \rho; t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule (**E − IF**$_\bot$).

(2) There exist $(\mathfrak{D}_{true}, \mathfrak{X}_{true})$ and $(\mathfrak{D}_{false}, \mathfrak{X}_{false})$ such that:

- $(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$,
- $\mathfrak{D} = [\![\mathfrak{D}_{true}, \mathfrak{D}_{false}]\!]_{\sigma(e)}^{\rho(e)} \cup ([\![\mathfrak{X}_{true}, \mathfrak{X}_{false}]\!]_{\sigma(e)}^{\rho(e)} \times \mathcal{FV}(e))$,
- $\mathfrak{X} = [\![\mathfrak{X}_{true}, \mathfrak{X}_{false}]\!]_{\sigma(e)}^{\rho(e)}$

This follows from the global hypothesis $\star_2$, the local conclusions (1), and the definition of $\models$.

(3) $\rho_o = (\lambda x. \bigsqcup_{y \in \mathfrak{D}_v(x)} \rho(y)) \sqcup ((\mathfrak{X}_v \times \{\top\}) \cup (\mathfrak{X}_v^c \times \{\bot\}))$.

This result is obtained by applying the inductive hypothesis on "$(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$" and "$\sigma; \rho; t^{\text{pc}} \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$".

(4) It is true that:

- $\mathfrak{D} = \mathfrak{D}_v \cup (\mathfrak{X}_v \times \mathcal{FV}(e))$,
- $\mathfrak{X} = \mathfrak{X}_v$

This follows from the local conclusion (2) and the facts that $\rho(e) = \bot$ and $\sigma(e) = v$ (from the local conclusion (1)).

(5) $\forall y \in \mathcal{FV}(e), \rho(y) = \bot$.
This follows from the fact that $\rho(e) = \bot$ (from the local conclusion (1)) and the semantic rules for expressions.

(•) $\rho_o = (\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.
This follows from the local conclusions (3), (4), and (5)

($\mathbf{E-IF}_\top$) then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exist a value $v \in \{true, false\}$ such that:

- " $\sigma;\rho \vdash e \Downarrow_{\mathcal{M}(O)} v : \top$ ",
- " $\sigma;\rho;\top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ " is a sub-derivation tree of " $\sigma;\rho;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ",
- $(\mathfrak{D}_{\neg v}, \mathfrak{X}_{\neg v}) \models (\sigma, \rho \vdash S_{\neg v})$,
- $\rho_o = \rho_v \sqcup (\lambda x. \bigsqcup_{y \in \mathfrak{D}_{\neg v}(x)} \rho(y)) \sqcup ((\mathfrak{X}_{\neg v} \times \{\top\}) \cup (\mathfrak{X}^c_{\neg v} \times \{\bot\}))$,

This follows directly from the definition of the rule ($\mathbf{E-IF}_\top$).

(2) There exist $(\mathfrak{D}_{true}, \mathfrak{X}_{true})$ and $(\mathfrak{D}_{false}, \mathfrak{X}_{false})$ such that:

- $(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$,
- $(\mathfrak{D}_{\neg v}, \mathfrak{X}_{\neg v}) \models (\sigma, \rho \vdash S_{\neg v})$,
- $\mathfrak{D} = (\mathfrak{D}_v \cup \mathfrak{D}_{\neg v}) \cup ((\mathfrak{X}_v \cup \mathfrak{X}_{\neg v}) \times \mathcal{FV}(e))$,
- $\mathfrak{X} = \mathfrak{X}_v \cup \mathfrak{X}_{\neg v}$

This follows from the global hypothesis $\star_2$, the local conclusions (1), the definition of $\models$, and the fact that $\rho(e) = \top$ (from the local conclusion (1)).

(3) $\rho_v = (\lambda x. \bigsqcup_{y \in \mathfrak{D}_v(x)} \rho(y)) \sqcup ((\mathfrak{X}_v \times \{\top\}) \cup (\mathfrak{X}^c_v \times \{\bot\}))$.
This result is obtained by applying the inductive hypothesis on the derivation " $\sigma;\rho;\top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ " (sub-derivation tree of " $\sigma;\rho;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") using the fact that "$(\mathfrak{D}_v, \mathfrak{X}_v) \models (\sigma, \rho \vdash S_v)$" (from the local conclusion (2)).

(4) $((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\})) = ((\mathfrak{X}_v \times \{\top\}) \cup (\mathfrak{X}^c_v \times \{\bot\})) \sqcup ((\mathfrak{X}_{\neg v} \times \{\top\}) \cup (\mathfrak{X}^c_{\neg v} \times \{\bot\}))$.
This follows from the local conclusion (2) and the fact that $\bot \sqsubseteq \top$.

(5) $\exists y \in \mathcal{FV}(e) : \rho(y) = \top$.
This follows directly from the fact that $\rho(e) = \top$ (given in the local conclusion (1)) and the semantics for expressions.

(6) $\forall x, \quad \bigsqcup_{y \in (\mathfrak{X} \times \mathcal{FV}(e))(x)} \rho(y) = ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x)$.
This follows from the local conclusion (2) and (5).

(•) $(\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\})) = \rho_o$.
For all x, let $t(x) = \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x)$, then:

$$\begin{aligned}
t(x) &= \bigsqcup_{y \in (\mathfrak{D}_v \cup \mathfrak{D}_{\neg v} \cup (\mathfrak{X} \times \mathcal{FV}(e)))(x)} \rho(y) \quad \sqcup \quad ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) \\
&= \bigsqcup_{y \in \mathfrak{D}_v(x)} \rho(y) \quad \sqcup \quad \bigsqcup_{y \in \mathfrak{D}_{\neg v}(x)} \rho(y) \quad \sqcup \quad \bigsqcup_{y \in (\mathfrak{X} \times \mathcal{FV}(e))(x)} \rho(y) \\
&\quad \sqcup \quad ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) \\
&= \bigsqcup_{y \in \mathfrak{D}_v(x)} \rho(y) \quad \sqcup \quad \bigsqcup_{y \in \mathfrak{D}_{\neg v}(x)} \rho(y) \\
&\quad \sqcup \quad ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) \quad \sqcup \quad ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) \\
&= \bigsqcup_{y \in \mathfrak{D}_v(x)} \rho(y) \quad \sqcup \quad \bigsqcup_{y \in \mathfrak{D}_{\neg v}(x)} \rho(y) \qquad\qquad \text{(C.2)} \\
&\quad \sqcup \quad ((\mathfrak{X}_v \times \{\top\}) \cup (\mathfrak{X}_v^c \times \{\bot\}))(x) \quad \sqcup \quad ((\mathfrak{X}_{\neg v} \times \{\top\}) \cup (\mathfrak{X}_{\neg v}^c \times \{\bot\}))(x) \\
&= \bigsqcup_{y \in \mathfrak{D}_v(x)} \rho(y) \quad \sqcup \quad ((\mathfrak{X}_v \times \{\top\}) \cup (\mathfrak{X}_v^c \times \{\bot\}))(x) \\
&\quad \sqcup \quad \bigsqcup_{y \in \mathfrak{D}_{\neg v}(x)} \rho(y) \quad \sqcup \quad ((\mathfrak{X}_{\neg v} \times \{\top\}) \cup (\mathfrak{X}_{\neg v}^c \times \{\bot\}))(x) \\
&= \rho_v(x) \quad \sqcup \quad \bigsqcup_{y \in \mathfrak{D}_{\neg v}(x)} \rho(y) \quad \sqcup \quad ((\mathfrak{X}_{\neg v} \times \{\top\}) \cup (\mathfrak{X}_{\neg v}^c \times \{\bot\}))(x) \\
&= \rho_o(x)
\end{aligned}$$

The equation (C.2) follows from the local conclusions (2), (6), (4), (3), and (1).

(**E − WHILE$_{\mathbf{skip}}$**) then we can conclude that :

(1) S is "e ; S_l" and:

- "$\sigma; \rho \vdash e \Downarrow_{\mathcal{M}(O)}$ `false` $: \bot$",
- $\rho_o = \rho$

This follows directly from the definition of the rule (**E − WHILE$_{\mathbf{skip}}$**).

(2) It is true that:

- $\mathfrak{D} = \mathcal{I}d$,
- $\mathfrak{X} = \emptyset$

This follows from the global hypothesis $\star_2$, the local conclusion (1), the definition of $\models$, and the facts that $\rho(e) = \bot$ and $\sigma(e) =$ `false` (from the local conclusion (1)).

(•) $\rho_o = (\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.
This follows directly from the local conclusions (1) and (2).

(**E − WHILE$_{\mathbf{true}_\bot}$**) then we can conclude that :

(1) S is "**while** e **do** S_l **done**" and:

- " $\sigma;\rho \vdash e \Downarrow_{\mathcal{M}(O)}$ `true` $: \bot$ "
- " $\sigma;\rho;\top \vdash S_l$; **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma;\rho;\top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule ($\mathbf{E-WHILE_{true_\bot}}$).

(2) There exist $(\mathfrak{D}_l, \mathfrak{X}_l)$ such that:

- $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho' \vdash S_l)$ with $\rho' = \rho \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$,
- $\mathfrak{D} = \mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d)$,
- $\mathfrak{X} = \mathfrak{X}_l$

This follows from the global hypothesis $\star_2$, the local conclusion (1), the definition of $\models$, and the facts that $\rho(e) = \bot$ and $\sigma(e) =$ `true` (from the local conclusion (1)).

(3) There exist a value store σ_1 and a tag store ρ_1:

- " $\sigma;\rho;\top \vdash S_l \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " is a sub-derivation tree of " $\sigma;\rho;\top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ",
- " $\sigma_1;\rho_1;\top \vdash$ **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma;\rho;\top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the local conclusion (1) and the definition of the rule (**E – SEQUENCE**).

(4) There exists an analysis result $(\mathfrak{D}_{S_l}, \mathfrak{X}_{S_l})$ such that:

- $(\mathfrak{D}_{S_l}, \mathfrak{X}_{S_l}) \models (\sigma, \rho \vdash S_l)$,
- $\mathfrak{X}_{S_l} \subseteq \mathfrak{X}_l$

This follows from the lemma C.1.2, the fact that $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho' \vdash S_l)$ (from the local conclusion (2)), and the fact that $\rho \sqsubseteq \rho'$ (from the definition of ρ' in the local conclusion (2)).

(5) $\rho_1 = (\lambda x. \bigsqcup_{y \in \mathfrak{D}_{S_l}(x)} \rho(y)) \sqcup ((\mathfrak{X}_{S_l} \times \{\top\}) \cup (\mathfrak{X}^c_{S_l} \times \{\bot\}))$.
This result is obtained by applying the inductive hypothesis on " $\sigma;\rho;\top \vdash S_l \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " (from the local conclusion (3)) and "$(\mathfrak{D}_{S_l}, \mathfrak{X}_{S_l}) \models (\sigma, \rho \vdash S_l)$" (from the local conclusion (4)).

(6) $\forall x \in \mathfrak{X}^c_l, \rho(x) = \rho_1(x)$.
From the local conclusion (4), $x \in \mathfrak{X}^c_{S_l}$. Hence, from the hypothesis 5.2.4, $\mathfrak{D}_{S_l}(x) = \{x\}$. Then, using the local conclusion (5), we can conclude that $\rho_1(x) = \rho(x)$.

Let us define ρ'_1 such that:

$\diamond_1$ $\rho'_1 = \rho_1 \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$.

(7) $\rho'_1 = \rho'$.
This follows from the local conclusion (6) and the definitions of ρ'_1 and ρ' (given in $\diamond_1$ and the local conclusion (2)). For all x in $\mathfrak{X}_l$, $\rho'_1(x) = \top = \rho'(x)$. For all x not in $\mathfrak{X}_l$, the local conclusion (6) implies $\rho'_1(x) = \rho_1(x) = \rho(x) = \rho'(x)$.

(8) $\forall x,\ \rho_1(x) = \bot \Rightarrow \sigma_1(x) = \sigma(x)$.
This follows from lemma C.2.2 and the local conclusion (3).

(•) $\rho_o = (\lambda x. \bigsqcup_{y\in\mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

Case 1: $\rho_1(e) = \bot$

(a) $(\mathfrak{D}, \mathfrak{X}) \models (\sigma, \rho_1 \vdash$ **while** e **do** S_l **done**$)$.

From the local conclusions (2) and (7), we can conclude that $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho_1' \vdash S_l)$ with $\rho_1' = \rho_1 \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$. From the local conclusion (2), $\mathfrak{D} = \mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d)$ and $\mathfrak{X} = \mathfrak{X}_l$, from the local conclusion (1), $\sigma(e) = \texttt{true}$, and, from the case hypothesis, $\rho_1(e) = \bot$, then we can conclude that $\mathfrak{D} = \{\!\![\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d), \mathcal{I}d]\!\!\}_{\sigma(e)}^{\rho_1(e)}$ and $\mathfrak{X} = \{\!\![\mathfrak{X}_l, \emptyset]\!\!\}_{\sigma(e)}^{\rho_1(e)}$. Hence, from the definition of $\models$ for while-statements, $(\mathfrak{D}, \mathfrak{X}) \models (\sigma, \rho_1 \vdash$ **while** e **do** S_l **done**$)$.

(b) $(\mathfrak{D}, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**$)$.

This follows from hypothesis 5.2.3 and the local conclusions (8) and (a).

(c) $\rho_o = (\lambda x. \bigsqcup_{y\in\mathfrak{D}(x)} \rho_1(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

This result is obtained by applying the inductive hypothesis on " $\sigma_1; \rho_1; \top \vdash$ **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (from the local conclusion (3)) and "$(\mathfrak{D}, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**$)$" (from the local conclusion (b)).

Let us define $\mathcal{F}_1^{\rho_o}$ and $\mathcal{F}^{\rho_o}$ such that:

$\diamond_1$ $\mathcal{F}_1^{\rho_o} = (\lambda x. \bigsqcup_{y\in\mathfrak{D}(x)} \rho_1(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$,

$\diamond_2$ $\mathcal{F}^{\rho_o} = (\lambda x. \bigsqcup_{y\in\mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

(•) $\rho_o(x) = (\bigsqcup_{y\in\mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) = \mathcal{F}^{\rho_o}(x)$.

From the local conclusion (c), $\rho_o(x) = \mathcal{F}_1^{\rho_o}(x)$, and, from the local conclusion (2), $\mathfrak{X} = \mathfrak{X}_l$. If x belongs to $\mathfrak{X}_l$, then $\mathcal{F}_1^{\rho_o}(x) = \top = \mathcal{F}^{\rho_o}(x)$. If x does not belong to $\mathfrak{X}_l$, then x does not belong to $\mathfrak{X}$ (from the local conclusion (2)) and, from the hypothesis 5.2.4, $\mathfrak{D}(x) = \{x\}$. Hence, if x does not belong to $\mathfrak{X}_l$, $\mathcal{F}_1^{\rho_o}(x) = \bigsqcup_{y\in\mathfrak{D}(x)} \rho_1(y) = \rho_1(x) = \rho(x)$ (the last equality comes from the local conclusion (6)). For similar reasons, if x does not belong to $\mathfrak{X}_l$ then $\mathcal{F}^{\rho_o}(x) = \bigsqcup_{y\in\mathfrak{D}(x)} \rho(y) = \rho(x)$.

Case 2: $\rho_1(e) \neq \bot$ (which implies that $\rho_1(e) = \top$)

(a) $\mathfrak{D} \cup \mathcal{I}d = (\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d \cup \mathcal{I}d)) \cup \mathcal{I}d$.

This follows directly from the fact $\mathfrak{D} = \mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d)$ in local conclusion (2).

(b) $(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**$)$.

From the local conclusions (2) and (7), we can conclude that $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho_1' \vdash S_l)$ with $\rho_1' = \rho_1 \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$. From the local conclusion (a), $\mathfrak{D} \cup \mathcal{I}d = (\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d \cup \mathcal{I}d)) \cup \mathcal{I}d$, from the local conclusion (2), $\mathfrak{X} = \mathfrak{X}_l$, and from the case hypothesis, $\rho_1(e) = \top$, then we can conclude that $\mathfrak{D} \cup \mathcal{I}d = \{\!\![\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d \cup \mathcal{I}d), \mathcal{I}d]\!\!\}_{\sigma(e)}^{\rho_1(e)}$ and $\mathfrak{X} = \{\!\![\mathfrak{X}_l, \emptyset]\!\!\}_{\sigma(e)}^{\rho_1(e)}$. Hence, from the definition of $\models$ for while-statements, $(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma, \rho_1 \vdash$ **while** e **do** S_l **done**$)$. Finally, the hypothesis 5.2.3 and the local conclusion (8) imply $(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**$)$.

(c) $\rho_o = (\lambda x. \bigsqcup_{y\in(\mathfrak{D}\cup\mathcal{I}d)(x)} \rho_1(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

This result is obtained by applying the inductive hypothesis on " $\sigma_1; \rho_1; \top \vdash$

while e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (from the local conclusion (3)) and "$(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**)" (from the local conclusion (b)).

Let us define $\mathcal{F}_1^{\rho_o}$ and $\mathcal{F}^{\rho_o}$ such that:

$\diamond_1$ $\mathcal{F}_1^{\rho_o} = (\lambda x. \bigsqcup_{y \in (\mathfrak{D} \cup \mathcal{I}d)(x)} \rho_1(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$,

$\diamond_2$ $\mathcal{F}^{\rho_o} = (\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

(•) $\rho_o(x) = (\bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) = \mathcal{F}^{\rho_o}(x)$.
From the local conclusion (c), $\rho_o(x) = \mathcal{F}_1^{\rho_o}(x)$, and, from the local conclusion (2), $\mathfrak{X} = \mathfrak{X}_l$. If x belongs to $\mathfrak{X}_l$, then $\mathcal{F}_1^{\rho_o}(x) = \top = \mathcal{F}^{\rho_o}(x)$. If x does not belong to $\mathfrak{X}_l$, then x does not belong to $\mathfrak{X}$ (from the local conclusion (2)) and, from the hypothesis 5.2.4, $\mathfrak{D}(x) = \{x\}$. Hence, if x does not belong to $\mathfrak{X}_l$, $\mathcal{F}_1^{\rho_o}(x) = \bigsqcup_{y \in (\mathfrak{D} \cup \mathcal{I}d)(x)} \rho_1(y) = \rho_1(x) = \rho(x)$ (the last equality comes from the local conclusion (6)). For similar reasons, if x does not belong to $\mathfrak{X}_l$ then $\mathcal{F}^{\rho_o}(x) = \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y) = \rho(x)$.

($\mathbf{E - WHILE_{true_\top}}$) then we can conclude that :

(1) S is "**while** e **do** S_l **done**" and:

- " $\sigma; \rho \vdash e \Downarrow_{\mathcal{M}(O)}$ `true` $: \top$ ",
- " $\sigma; \rho; \top \vdash S_l$; **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho'_o$ " is a sub-derivation tree of " $\sigma; \rho; \top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ",
- $\rho_o = \rho \sqcup \rho'_o$.

This follows directly from the definition of the rule ($\mathbf{E - WHILE_{true_\top}}$).

(2) There exist $(\mathfrak{D}_l, \mathfrak{X}_l)$ such that:

- $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho' \vdash S_l)$ with $\rho' = \rho \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$,
- $\mathfrak{D} = (\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d)) \cup \mathcal{I}d$,
- $\mathfrak{X} = \mathfrak{X}_l$

This follows from the global hypothesis $\star_2$, the local conclusion (1), the definition of $\models$, and the facts that $\rho(e) = \bot$ and $\sigma(e) =$ `true` (from the local conclusion (1)).

(3) There exist a value store σ_1 and a tag store ρ_1:

- " $\sigma; \rho; \top \vdash S_l \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " is a sub-derivation tree of " $\sigma; \rho; \top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ",
- " $\sigma_1; \rho_1; \top \vdash$ **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho'_o$ " is a sub-derivation tree of " $\sigma; \rho; \top \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the local conclusion (1) and the definition of the rule (**E – SEQUENCE**).

(4) There exists an analysis result $(\mathfrak{D}_{S_l}, \mathfrak{X}_{S_l})$ such that:

- $(\mathfrak{D}_{S_l}, \mathfrak{X}_{S_l}) \models (\sigma, \rho \vdash S_l)$,
- $\mathfrak{X}_{S_l} \subseteq \mathfrak{X}_l$

This follows from the lemma C.1.2, the fact that $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho' \vdash S_l)$ (from the local conclusion (2)), and the fact that $\rho \sqsubseteq \rho'$ (from the definition of ρ' in the local conclusion (2)).

(5) $\forall x \in \mathfrak{X}_l^c, \rho(x) = \rho_1(x)$.

By applying the inductive hypothesis on " $\sigma; \rho; \top \vdash S_l \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " (from the local conclusion (3)) and "$(\mathfrak{D}_{S_l}, \mathfrak{X}_{S_l}) \models (\sigma, \rho \vdash S_l)$" (from the local conclusion (4)), we get that $\rho_1 = (\lambda x. \bigsqcup_{y \in \mathfrak{D}_{S_l}(x)} \rho(y)) \sqcup ((\mathfrak{X}_{S_l} \times \{\top\}) \cup (\mathfrak{X}_{S_l}^c \times \{\bot\}))$. From the local conclusion (4), $x \in \mathfrak{X}_{S_l}^c$. Hence, from the hypothesis 5.2.4, $\mathfrak{D}_{S_l}(x) = \{x\}$. Then, we can conclude that $\rho_1(x) = \rho(x)$.

Let us define ρ'_1 such that:

$\diamond_1$ $\rho'_1 = \rho_1 \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$.

(6) $\rho'_1 = \rho'$.

This follows from the local conclusion (5) and the definitions of ρ'_1 and ρ' (given in $\diamond_1$ and the local conclusion (2)). For all x in $\mathfrak{X}_l$, $\rho'_1(x) = \top = \rho'(x)$. For all x not in $\mathfrak{X}_l$, the local conclusion (5) implies $\rho'_1(x) = \rho_1(x) = \rho(x) = \rho'(x)$.

(7) $\forall x,\ \rho_1(x) = \bot \Rightarrow \sigma_1(x) = \sigma(x)$.

This follows from lemma C.2.2 and the local conclusion (3).

(8) $\mathfrak{D} \cup \mathcal{I}d = (\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d \cup \mathcal{I}d)) \cup \mathcal{I}d$.

This follows directly from the fact $\mathfrak{D} = (\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d)) \cup \mathcal{I}d$ in local conclusion (2).

(9) $(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**$)$.

From the local conclusions (2) and (6), we can conclude that $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho'_1 \vdash S_l)$ with $\rho'_1 = \rho_1 \sqcup ((\mathfrak{X}_l \times \top) \cup ((\mathfrak{X}_l)^c \times \bot))$. From the local conclusion (8), $\mathfrak{D} \cup \mathcal{I}d = (\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d \cup \mathcal{I}d)) \cup \mathcal{I}d$, from the local conclusion (2), $\mathfrak{X} = \mathfrak{X}_l$, and , from the local conclusion (1), $\sigma(e) = \texttt{true}$, then we can conclude that $\mathfrak{D} \cup \mathcal{I}d = \{\!\!\{(\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d \cup \mathcal{I}d)) \cup \mathcal{I}d, \mathcal{I}d\}\!\!\}_{\sigma(e)}^{\rho_1(e)}$ and $\mathfrak{X} = \{\!\!\{\mathfrak{X}_l, \emptyset\}\!\!\}_{\sigma(e)}^{\rho_1(e)}$. Hence, from the definition of $\models$ for while-statements, $(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma, \rho_1 \vdash$ **while** e **do** S_l **done**$)$. Finally, the hypothesis 5.2.3 and the local conclusion (7) imply $(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**$)$.

(10) $\rho'_o = (\lambda x. \bigsqcup_{y \in (\mathfrak{D} \cup \mathcal{I}d)(x)} \rho_1(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

This result is obtained by applying the inductive hypothesis on " $\sigma_1; \rho_1; \top \vdash$ **while** e **do** S_l **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho'_o$ " (from the local conclusion (3)) and "$(\mathfrak{D} \cup \mathcal{I}d, \mathfrak{X}) \models (\sigma_1, \rho_1 \vdash$ **while** e **do** S_l **done**$)$" (from the local conclusion (9)).

Let us define $\mathcal{F}_1^{\rho'_o}$ and $\mathcal{F}^{\rho_o}$ such that:

$\diamond_1$ $\mathcal{F}_1^{\rho'_o} = (\lambda x. \bigsqcup_{y \in (\mathfrak{D} \cup \mathcal{I}d)(x)} \rho_1(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$,

$\diamond_2$ $\mathcal{F}^{\rho_o} = (\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

(•) $\rho_o(x) = (\bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \ \sqcup\ ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) \ = \mathcal{F}^{\rho_o}(x)$.

From the local conclusion (10), $\rho'_o(x) = \mathcal{F}_1^{\rho'_o}(x)$, and, from the local conclusion (2), $\mathfrak{X} = \mathfrak{X}_l$. If x belongs to $\mathfrak{X}_l$, then $\rho(x) \sqcup \mathcal{F}_1^{\rho'_o}(x) = \top = \mathcal{F}^{\rho_o}(x)$. If x does not belong to $\mathfrak{X}_l$, then x does not belong to $\mathfrak{X}$ (from the local conclusion (2)) and, from the hypothesis 5.2.4, $\mathfrak{D}(x) = \{x\}$. Hence, if x does not belong to $\mathfrak{X}_l$, $\rho(x) \sqcup \mathcal{F}_1^{\rho'_o}(x) = \rho(x) \sqcup \bigsqcup_{y \in (\mathfrak{D} \cup \mathcal{I}d)(x)} \rho_1(y) = \rho(x) \sqcup \rho_1(x) = \rho(x)$ (the last equality comes from the local conclusion (5)). For similar reasons, if x does not belong to $\mathfrak{X}_l$ then $\mathcal{F}^{\rho_o}(x) = \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y) = \rho(x)$.

(E – WHILE$_{\mathbf{false}_\top}$) then we can conclude that :

(1) S is "**while** e **do** S_l **done**" and there exists $(\mathfrak{D}_S, \mathfrak{X}_S)$ such that:

- "$\sigma;\rho \vdash e \Downarrow_{\mathcal{M}(O)}$ `false` $: \top$"
- $(\mathfrak{D}_S, \mathfrak{X}_S) \models (\sigma, \rho \vdash S_l$; **while** e **do** S_l **done**),
- $\rho_o = \rho \sqcup (\lambda x. \bigsqcup_{y \in \mathfrak{D}_S(x)} \rho(y)) \sqcup ((\mathfrak{X}_S \times \{\top\}) \cup (\mathfrak{X}_S^c \times \{\bot\}))$,

This follows directly from the definition of the rule **(E – WHILE$_{\mathbf{false}_\top}$)**.

(2) There exist $(\mathfrak{D}_l, \mathfrak{X}_l)$ such that:

- $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho' \vdash S_l)$ with $\rho' = \rho \sqcup ((\mathfrak{X}_l \times \{\top\}) \cup ((\mathfrak{X}_l)^c \times \{\bot\}))$,
- $\mathfrak{D} = (\mathfrak{D}_l \circ (\mathfrak{D} \cup \mathcal{I}d)) \cup \mathcal{I}d$,
- $\mathfrak{X} = \mathfrak{X}_l$

This follows from the global hypothesis $\star_2$, the local conclusion (1), the definition of $\models$, and the fact that $\rho(e) = \top$ (from the local conclusion (1)).

(3) There exist $(\mathfrak{D}_1, \mathfrak{X}_1)$ and $(\mathfrak{D}_2, \mathfrak{X}_2)$ such that:

- $(\mathfrak{D}_1, \mathfrak{X}_1) \models (\sigma, \rho \vdash S_l)$
- $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho'' \vdash$ **while** e **do** S_l **done**) with $\rho'' = \rho \sqcup ((\mathfrak{X}_1 \times \{\top\}) \cup ((\mathfrak{X}_1)^c \times \{\bot\}))$
- $\mathfrak{D}_S = \mathfrak{D}_1 \circ \mathfrak{D}_2$
- $\mathfrak{X}_S = \mathfrak{X}_1 \cup \mathfrak{X}_2$

This follows from the fact that $(\mathfrak{D}_S, \mathfrak{X}_S) \models (\sigma, \rho \vdash S_l$; **while** e **do** S_l **done**) (given in the local conclusion (1)) and the definition of $\models$.

(4) There exist $(\mathfrak{D}_3, \mathfrak{X}_3)$ such that:

- $(\mathfrak{D}_3, \mathfrak{X}_3) \models (\sigma, \rho''' \vdash S_l)$ with $\rho''' = \rho'' \sqcup ((\mathfrak{X}_3 \times \{\top\}) \cup ((\mathfrak{X}_3)^c \times \{\bot\}))$,
- $\mathfrak{D}_2 = (\mathfrak{D}_3 \circ (\mathfrak{D}_2 \cup \mathcal{I}d)) \cup \mathcal{I}d$,
- $\mathfrak{X}_2 = \mathfrak{X}_3$

This follows from the fact that $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho'' \vdash$ **while** e **do** S_l **done**) (given in the local conclusion (3)), the definition of $\models$, and the fact that $\rho(e) = \top$ (from the local conclusion (1)).

(5) $\mathfrak{X}_1 \subseteq \mathfrak{X}$ and $\forall x \in \mathfrak{X}_1^c,\ \mathfrak{D}_1(x) = \{x\}$.
The lemma C.1.2, the facts that $(\mathfrak{D}_l, \mathfrak{X}_l) \models (\sigma, \rho' \vdash S_l)$ (from the local conclusion (2)) and $(\mathfrak{D}_1, \mathfrak{X}_1) \models (\sigma, \rho \vdash S_l)$ (from the local conclusion (3)), and the fact that $\rho \sqsubseteq \rho'$ (from the definition of ρ' in the local conclusion (2)) imply $\mathfrak{X}_1 \subseteq \mathfrak{X}_l$. Hence, from the local conclusion (2), $\mathfrak{X}_1 \subseteq \mathfrak{X}$. Finally, the hypothesis 5.2.4 and the local conclusion (3) imply $\forall x \in \mathfrak{X}_1^c, \mathfrak{D}_1(x) = \{x\}$.

(6) $\mathfrak{X} \subseteq \mathfrak{X}_2$.

This follows from the lemma C.1.2, the facts that $(\mathfrak{D}, \mathfrak{X}) \models (\sigma, \rho \vdash$ **while** e **do** S_l **done**$)$ (from the global hypothesis $\star_2$ and the local conclusion (1)) and $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho'' \vdash$ **while** e **do** S_l **done**$)$ (from the local conclusion (3)), and the fact that $\rho \sqsubseteq \rho''$ (from the definition of ρ'' in the local conclusion (3)).

(7) $\rho''' = \rho \sqcup ((\mathfrak{X}_3 \times \{\top\}) \cup ((\mathfrak{X}_3)^c \times \{\bot\}))$.

The definitions of ρ'' and ρ''' (given in the local conclusions (3) and (4)) imply $\rho''' = \rho \sqcup ((\mathfrak{X}_1 \times \{\top\}) \cup ((\mathfrak{X}_1)^c \times \{\bot\})) \sqcup ((\mathfrak{X}_3 \times \{\top\}) \cup ((\mathfrak{X}_3)^c \times \{\bot\}))$. Additionally, the local conclusions (5), (6) and (4) imply $\mathfrak{X}_1 \subseteq \mathfrak{X}_3$. Hence, $((\mathfrak{X}_1 \times \{\top\}) \cup ((\mathfrak{X}_1)^c \times \{\bot\})) \sqcup ((\mathfrak{X}_3 \times \{\top\}) \cup ((\mathfrak{X}_3)^c \times \{\bot\}))$ equals $((\mathfrak{X}_3 \times \{\top\}) \cup ((\mathfrak{X}_3)^c \times \{\bot\}))$.

(8) $\mathfrak{D} = \mathfrak{D}_2$ and $\mathfrak{X} = \mathfrak{X}_2$.

The local conclusions (4) and (7) and the fact that $\rho(e) = \top$ (from the local conclusion (1)) imply $(\mathfrak{D}_2, \mathfrak{X}_2) \models (\sigma, \rho \vdash$ **while** e **do** S_l **done**$)$. Hence, the hypothesis 5.2.3 implies $(\mathfrak{D}, \mathfrak{X})$ equals $(\mathfrak{D}_2, \mathfrak{X}_2)$.

(9) $\mathfrak{D}_S = \mathfrak{D}_l \circ \mathfrak{D}$ and $\mathfrak{X}_S = \mathfrak{X}$.

The local conclusions (3) and (8) imply $\mathfrak{D}_S = \mathfrak{D}_l \circ \mathfrak{D}$. Finally, the local conclusions (5), (6), and (8) imply $\mathfrak{X}_1 \subseteq \mathfrak{X}$; hence, the local conclusion (3) implies $\mathfrak{X}_S = \mathfrak{X}$.

(10) $\rho_o = (\lambda x. \bigsqcup_{y \in (\mathfrak{D}_S \cup \mathcal{I}d)(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

This follows directly from the local conclusions (1) and (9).

Let us define $\mathcal{F}_S^{\rho_o}$ and $\mathcal{F}^{\rho_o}$ such that:

$\diamond_1$ $\mathcal{F}_S^{\rho_o} = (\lambda x. \bigsqcup_{y \in (\mathfrak{D}_S \cup \mathcal{I}d)(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$,

$\diamond_2$ $\mathcal{F}^{\rho_o} = (\lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))$.

(•) $\rho_o(x) = (\bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)) \sqcup ((\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\}))(x) = \mathcal{F}^{\rho_o}(x)$.

From the local conclusion (10), $\rho_o(x) = \mathcal{F}_S^{\rho_o}(x)$ and, from the local conclusion (9), $\mathfrak{X}_S = \mathfrak{X}$. If x belongs to $\mathfrak{X}$, then $\mathcal{F}_S^{\rho_o}(x) = \top = \mathcal{F}^{\rho_o}(x)$. If x does not belong to $\mathfrak{X}$, then , from the hypothesis 5.2.4, $\mathfrak{D}_S(x) = \{x\}$. Hence, if x does not belong to $\mathfrak{X}$, $\mathcal{F}_S^{\rho_o}(x) = \bigsqcup_{y \in (\mathfrak{D}_S \cup \mathcal{I}d)(x)} \rho(y) = \rho(x)$. For similar reasons, if x does not belong to $\mathfrak{X}$ then $\mathcal{F}^{\rho_o}(x) = \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y) = \rho(x)$.

C.2 Soundness

C.2.1 Detection

Axiom C.2.1 (Deterministic Expressions).
For all expressions e, value stores σ_1 and σ_2, the property "$\forall x \in FV(e)\ (\sigma_1(x) = \sigma_2(x))$" implies $\sigma_1(e) = \sigma_2(e)$.

Lemma C.2.1 (Public expressions are stable).
For all tag stores ρ, value stores σ_1 and σ_2, and expression e such that for all variable x the following holds $(\rho(x) = \bot) \Rightarrow \sigma_1(x) = \sigma_2(x)$, if $\rho(e) = \bot$ then $\sigma_1(e) = \sigma_2(e)$.

Proof. The proof is straightforward. It follows directly from the facts that expression evaluation is deterministic (Axiom C.2.1), the tag of an expression is the least upper bound of the tags of its free variables (definition on page 83) and that public ($\bot$) variables have the same value in σ_1 and σ_2.

Lemma C.2.2 (Tag of assigned variables contains t^{pc}).
For all value stores σ_i, tag stores ρ_i, and statement S such that:

- $\sigma_i; \rho_i; t^{pc} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$

it is true that:

- $\forall x : t^{pc} \not\sqsubseteq \rho_o(x) \;\Rightarrow\; (\sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x))$

Proof. The proof goes by induction on the derivation tree of " $\sigma_i; \rho_i; t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

$(\mathbf{E}_{\mathcal{M}(O)} - \mathbf{IF}_\bot)$ then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and " $\sigma_i; \rho_i; t^{\mathrm{pc}} \sqcup \bot \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i; \rho_i; t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ".
This follows directly from the definition of the rule $(\mathbf{E}_{\mathcal{M}(O)} - \mathbf{IF}_\bot)$.

(•) $\forall x : t^{\mathrm{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
This is obtained by a simple induction because $t^{\mathrm{pc}} \sqcup \bot = t^{\mathrm{pc}}$.

$(\mathbf{E}_{\mathcal{M}(O)} - \mathbf{IF}_\top)$ then we can conclude that :

(1) S is "**if** e **then** S_1 **else** S_2 **end**" and there exist S_v and ρ_v such that:

- " $\sigma_i; \rho_i; t^{\mathrm{pc}} \sqcup \top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ " is a sub-derivation tree of " $\sigma_i; \rho_i; t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- $\forall x : \rho_v(x) \sqsubseteq \rho_o(x)$

This follows directly from the definition of the rule $(\mathbf{E}_{\mathcal{M}(O)} - \mathbf{IF}_\top)$.

(2) $\forall x : t^{\mathrm{pc}} \sqcup \top \sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_v(x)$.
This follows directly from the inductive hypothesis and the local conclusion (1).

(•) $\forall x : t^{\mathrm{pc}} \sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
As $t^{\mathrm{pc}} \sqsubseteq \rho_o(x)$ implies $t^{\mathrm{pc}} \sqcup \top \sqsubseteq \rho_o(x)$, the above result follows directly from the local conclusions (1) and (2).

$(\mathbf{E}_{\mathcal{M}(O)} - \mathbf{WHILE}_{\mathbf{skip}})$ then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and $\sigma_o = \sigma_i \wedge \rho_i = \rho_o$.
This follows directly from the definition of the rule $(\mathbf{E}_{\mathcal{M}(O)} - \mathbf{WHILE}_{\mathbf{skip}})$.

(•) $\forall x : t^{\text{pc}} \sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
This follows directly from the local conclusion (1).

($\mathbf{E}_{\mathcal{M}(O)}$ − WHILE$_{\mathbf{true}_\bot}$) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and there exist S', t'_e and ρ' such that:

- " $\sigma_i;\rho_i;t^{\text{pc}}\sqcup\bot \vdash S' \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule ($\mathbf{E}_{\mathcal{M}(O)}$ − **WHILE$_{\mathbf{true}_\bot}$**).

(•) $\forall x : t^{\text{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
This follows directly from the inductive hypothesis and the local conclusion (1).

($\mathbf{E}_{\mathcal{M}(O)}$ − WHILE$_{\mathbf{true}_\top}$) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and there exist S', t'_e and ρ' such that:

- " $\sigma_i;\rho_i;t^{\text{pc}}\sqcup t'_e \vdash S' \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho'$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- $\forall x : \rho'(x) \sqsubseteq \rho_o(x)$

This follows directly from the definition of the rule ($\mathbf{E}_{\mathcal{M}(O)}$ − **WHILE$_{\mathbf{true}_\top}$**).

(2) $\forall x : t^{\text{pc}} \sqcup t'_e \not\sqsubseteq \rho'(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho'(x)$.
This follows directly from the inductive hypothesis and the local conclusion (1).

(•) $\forall x : t^{\text{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
As $t^{\text{pc}} \not\sqsubseteq \rho_o(x)$ and $\rho'(x) \sqsubseteq \rho_o(x)$ imply $t^{\text{pc}} \sqcup t'_e \not\sqsubseteq \rho'(x)$, the above result follows from the local conclusions (1) and (2).

($\mathbf{E}_{\mathcal{M}(O)}$ − WHILE$_{\mathbf{false}_\top}$) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**", $\sigma_o = \sigma_i$ and $\forall x : \rho_i(x) \sqsubseteq \rho_o(x)$.
This follows directly from the definition of the rule ($\mathbf{E}_{\mathcal{M}(O)}$ − **WHILE$_{\mathbf{false}_\top}$**).

(•) $\forall x : t^{\text{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
This follows directly from the local conclusion (1).

($\mathbf{E}_{\mathcal{M}(O)}$ − SEQUENCE) then we can conclude that :

(1) S is "S_1 ; S_2" and there exist σ_1 and ρ_1 such that:

- " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- " $\sigma_1;\rho_1;t^{\text{pc}} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule ($\mathbf{E}_{\mathcal{M}(O)}$ − **SEQUENCE**).

(2) $\forall x : t^{\text{pc}} \not\sqsubseteq \rho_1(x) \Rightarrow \sigma_1(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_1(x)$ and $\forall x : t^{\text{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_1(x) \wedge \rho_1(x) \sqsubseteq \rho_o(x)$.
This follows directly from the inductive hypothesis and the local conclusion (1).

(•) $\forall x : t^{\mathrm{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
From the local conclusion (1), if $t^{\mathrm{pc}} \not\sqsubseteq \rho_o(x)$ then $\rho_1(x) \sqsubseteq \rho_o(x)$. Hence, $t^{\mathrm{pc}} \not\sqsubseteq \rho_1(x)$. Then, combining the results of the local conclusion (2), "$\sigma_o(x) = \sigma_1(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_1(x) \sqsubseteq \rho_o(x)$"

($\mathbf{E}_{\mathcal{M}(O)}$ − **ASSIGN**) then we can conclude that :

(1) S is "id **:=** e" and there exist v_e, and t_e such that $\sigma_o = \sigma_i[id \mapsto v_e]$ and $\rho_o = \rho_i[id \mapsto t_e \sqcup t^{\mathrm{pc}}]$.
This follows directly from the definition of the rule ($\mathbf{E}_{\mathcal{M}(O)}$ − **ASSIGN**).

(•) $\forall x : t^{\mathrm{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
From (1), if $x = id$ then $t^{\mathrm{pc}} \sqsubseteq \rho_o(x)$. Otherwise, $x \neq id$ and $\sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) = \rho_o(x)$

($\mathbf{E}_{\mathcal{M}(O)}$ − **SKIP**) then we can conclude that :

(1) S is "**skip**" and $\sigma_o = \sigma_i \wedge \rho_i = \rho_o$.
This follows directly from the definition of the rule ($\mathbf{E}_{\mathcal{M}(O)}$ − **SKIP**).

(•) $\forall x : t^{\mathrm{pc}} \not\sqsubseteq \rho_o(x) \Rightarrow \sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.
This follows directly from the local conclusion (1).

Lemma C.2.3 (Correctness of semantics $\Downarrow_{\mathcal{M}(O)}$ for information flow detection).
For all:

- *variable x,*
- *value stores σ_i, σ_o, σ'_i, and σ'_o,*
- *tag stores ρ_i, ρ_o, ρ'_i, and ρ'_o,*
- *tags t^{pc} and $t^{pc'}$,*
- *and statement S*

such that:

$\star_1$ $\forall y : (\rho_i(y) = \bot) \Rightarrow \sigma_i(y) = \sigma'_i(y)$,

$\star_2$ $\sigma_i; \rho_i; t^{pc} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$,

$\star_3$ $\sigma'_i; \rho'_i; t^{pc'} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$,

$\star_4$ $\rho_o(x) = \bot$

it is true that:

- $\sigma_o(x) = \sigma'_o(x)$.

Proof. The proof goes by induction on the derivation tree of " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

$(\mathbf{E-IF}_\bot)$ then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exists $v \in \{true, false\}$ such that:

- " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : \bot$ "
- " $\sigma_i;\rho_i;t^{\mathrm{pc}} \sqcup \bot \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule $(\mathbf{E-IF}_\bot)$.

(2) There exist a value v', a tag t', and a tag store ρ' such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'$ "
- " $\sigma'_i;\rho'_i;t^{\mathrm{pc}'} \sqcup t' \vdash S_{v'} \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'$ "

This follows from the global hypothesis $\star_3$, the local conclusion (1), and the definitions of the rules applying to "**if** e **then** S_{true} **else** S_{false} **end**" ($(\mathbf{E-IF}_\bot)$ and $(\mathbf{E-IF}_\top)$).

(3) $v = v'$.

This result comes from lemma C.2.1 applied to the derivations " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : \bot$ " and " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'$ ". It is possible to apply it because of the global hypothesis $\star_1$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

The conclusion is obtained by applying the inductive hypothesis on " $\sigma_i;\rho_i;t^{\mathrm{pc}} \sqcup \bot \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (sub-derivation tree of " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_i;\rho'_i;t^{\mathrm{pc}'} \sqcup t' \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'$ " (because $v = v'$).

$(\mathbf{E-IF}_\top)$ then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exist $v \in \{true, false\}$, two tag store ρ_v and ρ_e, and an analysis result $(\mathfrak{D}, \mathfrak{X})$ such that:

- " $\sigma_i;\rho_i;t^{\mathrm{pc}} \sqcup \top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- $[\![\sigma_i;\rho_i \vdash S_{\neg v}]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X})$
- $\rho_v(x) \sqsubseteq \rho_o(x)$
- $\rho_e = (\mathfrak{X} \times \{\top\}) \cup ((\mathbb{I}d - \mathfrak{X}) \times \{\bot\})$
- $\rho_e(x) \sqsubseteq \rho_o(x)$

This follows directly from the definition of the rule $(\mathbf{E-IF}_\bot)$.

(2) There exist a value $v' \in \{true, false\}$, a tag store $\rho'_{v'}$, and a tag t'_e such that:

- " $\sigma'_i;\rho'_i;t^{\mathrm{pc}'} \sqcup t'_e \vdash S_{v'} \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_{v'}$ "

This follows from the global hypothesis $\star_3$, the local conclusion (1), and the definitions of the rules applying to "**if** e **then** S_{true} **else** S_{false} **end**" ($(\mathbf{E-IF}_\bot)$ and $(\mathbf{E-IF}_\top)$).

Case 1: $v = v'$

(a) $\rho_v(x) = \bot$.

This follows from the local conclusion (1) and the global hypothesis $\star_4$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

The conclusion is obtained by applying the inductive hypothesis on " $\sigma_i; \rho_i; t^{\text{pc}} \sqcup \top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ " (sub-derivation tree of " $\sigma_i; \rho_i; t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_i; \rho'_i; t^{\text{pc}'} \sqcup t'_e \vdash S_{v'} \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_{v'}$ ". It is possible to apply the inductive hypothesis because of the local conclusion (a) and the fact that, by the case hypothesis, $v = v'$).

Case 2: $v \neq v'$ (which implies that $v' = \neg v$)

(a) $\sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.

From the global hypothesis $\star_4$, $\rho_o(x) = \bot$. Hence, from lemma C.2.2 applied to " $\sigma_i; \rho_i; t^{\text{pc}} \sqcup \top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ ", $\sigma_o(x) = \sigma_i(x) \wedge \rho_i(x) \sqsubseteq \rho_o(x)$.

(b) $\sigma'_o(x) = \sigma'_i(x)$.

From the local conclusion (1), the global hypothesis $\star_4$, and the definition of ρ_e in the local conclusion (1), $x \notin \mathfrak{X}$. Hence, from the local conclusion (1) and the hypothesis 5.2.1 applied to " $\sigma'_i; \rho'_i; t^{\text{pc}'} \sqcup t'_e \vdash S_{v'} \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_{v'}$ ", $\sigma'_o(x) = \sigma'_i(x)$.

(c) $\sigma_i(x) = \sigma'_i(x)$.

From the local conclusion (a) and the global hypothesis $\star_4$, we get that $\rho_i(x) = \bot$. Hence, from the global hypothesis $\star_1$, we get that $\sigma_i(x) = \sigma'_i(x)$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

This follows directly from the local conclusions (a), (b), and (c).

(**E – WHILE$_{\textbf{skip}}$**) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and:

- " $\sigma_i; \rho_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{false} : \bot$ "
- $\sigma_i = \sigma_o$ and $\rho_i = \rho_o$

This follows directly from the definition of the rule (**E – IF$_\bot$**).

(2) There exist a value v' and a tag t' such that:

- " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'$ "

This follows from the global hypothesis $\star_3$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" ((**E – WHILE$_{\textbf{skip}}$**), (**E – WHILE$_{\textbf{true}_\bot}$**), (**E – WHILE$_{\textbf{true}_\top}$**) and (**E – WHILE$_{\textbf{false}_\top}$**)).

(3) $\sigma'_i = \sigma'_o$.

From lemma C.2.1 and the local conclusions (1) and (2), we get $v' = false$. Hence, from the global hypothesis $\star_3$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{false} : t'$ " ((**E – WHILE$_{\textbf{skip}}$**) and (**E – WHILE$_{\textbf{false}_\top}$**)), $\sigma'_i = \sigma'_o$.

(4) $\sigma_i(x) = \sigma'_i(x)$.

From the local conclusion (1) and the global hypothesis $\star_4$, we get that $\rho_i(x) = \bot$. Hence, from the global hypothesis $\star_1$, we get that $\sigma_i(x) = \sigma'_i(x)$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

This follows directly from the local conclusions (1), (3), and (4).

(**E – WHILE$_{\mathbf{true}_\bot}$**) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and:

- " $\sigma_i; \rho_i \vdash e \Downarrow_{\mathcal{M}(O)}$ `true` $: \bot$ "
- " $\sigma_i; \rho_i; t^{\text{pc}} \sqcup \bot \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i; \rho_i; t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule (**E – WHILE$_{\mathbf{true}_\bot}$**).

(2) There exist a value v' and a tag t'_e such that:

- " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

This follows from the global hypothesis $\star_3$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" ((**E – WHILE$_{\mathbf{skip}}$**), (**E – WHILE$_{\mathbf{true}_\bot}$**), (**E – WHILE$_{\mathbf{true}_\top}$**) and (**E – WHILE$_{\mathbf{false}_\top}$**)).

(3) $v' =$ `true`.

This follows directly from the local conclusions (1) and (2), and lemma C.2.1.

(4) There exists a tag store ρ'_l such that:

- " $\sigma'_i; \rho'_i; t^{\text{pc}'} \sqcup t'_e \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_l$ "

This follows from the global hypothesis $\star_3$, the local conclusions (1), (2) and (3), and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)}$ `true` $: t'_e$ " ((**E – WHILE$_{\mathbf{true}_\bot}$**) and (**E – WHILE$_{\mathbf{true}_\top}$**)).

(•) $\sigma_o(x) = \sigma'_o(x)$.

By applying the inductive hypothesis on the derivations " $\sigma_i; \rho_i; t^{\text{pc}} \sqcup \bot \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " and " $\sigma'_i; \rho'_i; t^{\text{pc}'} \sqcup t'_e \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_l$ ", it is possible to deduce that $\sigma_o(x) = \sigma'_o(x)$.

(**E – WHILE$_{\mathbf{true}_\top}$**) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and there exist a tag t_e and a tag store ρ_l such that:

- " $\sigma_i; \rho_i \vdash e \Downarrow_{\mathcal{M}(O)}$ `true` $: \top$ "
- " $\sigma_i; \rho_i; t^{\text{pc}} \sqcup \top \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_l$ " is a sub-derivation tree of " $\sigma_i; \rho_i; t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- $\rho_l \sqsubseteq \rho_o$

This follows directly from the definition of the rule (**E – WHILE$_{\mathbf{true}_\top}$**).

(2) $\rho_l(x) = \bot$.

This follows from the local conclusion (1) and the global hypothesis $\star_4$.

(3) There exist a value v' and a tag t'_e such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

This follows from the global hypothesis $\star_3$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" ((**E – WHILE$_{\text{skip}}$**), (**E – WHILE$_{\text{true}_\bot}$**), (**E – WHILE$_{\text{true}_\top}$**) and (**E – WHILE$_{\text{false}_\top}$**)).

Case 1: $v' = \texttt{true}$

(a) There exists a tag store ρ'_l such that:

- " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup t'_e \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_l$ "

This follows from the global hypothesis $\star_3$, the local conclusions (1) and (3), the case hypothesis, and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{true} : t'_e$ " ((**E – WHILE$_{\text{true}_\bot}$**) and (**E – WHILE$_{\text{true}_\top}$**)).

(•) $\sigma_o(x) = \sigma'_o(x)$.

By applying the inductive hypothesis on the derivations " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \top \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_l$ " and " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup t'_e \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_l$ ", it is possible to deduce, using the local conclusion (2), that $\sigma_o(x) = \sigma'_o(x)$.

Case 2: $v' = \texttt{false}$

(a) $\sigma_o(x) = \sigma_i(x)$.

This follows directly from the local conclusions (1), the fact that "$t^{\text{pc}} \sqcup \top \not\sqsubseteq \rho_l(x)$" (from the local conclusion (2)), and the lemma C.2.2 applied to " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \top \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_l$ ".

(b) $\sigma'_o(x) = \sigma'_i(x)$.

From the global hypothesis $\star_3$, the local conclusions (1) and (3), the case hypothesis, and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{false} : t'_e$ " ((**E – WHILE$_{\text{skip}}$**) and (**E – WHILE$_{\text{false}_\top}$**)), $\sigma'_i = \sigma'_o$.

(c) $\sigma_i(x) = \sigma'_i(x)$.

From the local conclusion (2) and lemma C.2.2 applied to " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \top \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_l$ ", $\rho_i(x) = \bot$. Hence, from the global hypothesis $\star_1$, we get that $\sigma_i(x) = \sigma'_i(x)$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

This follows directly from the local conclusions (a), (b), and (c).

(**E – WHILE$_{\text{false}_\top}$**) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and there exist two tag stores ρ_1 and ρ_e such that:

- " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{false} : \top$ "

- $[\![\sigma_i;\rho_i \vdash S_1 \; ; \; \textbf{while } e \textbf{ do } S_1 \textbf{ done}]\!]^{\sharp_\mathcal{G}} = (\mathfrak{D}, \mathfrak{X})$
- $\sigma_o = \sigma_i$
- $\rho_e = (\mathfrak{X} \times \{\top\}) \cup ((\mathbb{I}d - \mathfrak{X}) \times \{\bot\})$
- $\rho_e(x) \sqsubseteq \rho_o(x)$ and $\rho_i(x) \sqsubseteq \rho_o(x)$

This follows directly from the definition of the rule $(\textbf{E} - \textbf{WHILE}_\textbf{true})$.

(2) There exist a value $v' \in \{\texttt{true}, \texttt{false}\}$ and a tag t'_e such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

This follows from the global hypothesis $\star_3$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" $((\textbf{E} - \textbf{WHILE}_\textbf{skip})$, $(\textbf{E} - \textbf{WHILE}_\textbf{true})$ and $(\textbf{E} - \textbf{WHILE}_{\textbf{false}_\top}))$.

(3) $\sigma_i(x) = \sigma'_i(x)$.

From the local conclusion (1) $(\rho_i(x) \sqsubseteq \rho_o(x))$ and the global hypothesis $\star_4$, $\rho_i(x) = \bot$. Hence, from the global hypothesis $\star_1$, we get that $\sigma_i(x) = \sigma'_i(x)$.

Case 1: $v' = \texttt{false}$

(a) $\sigma'_o = \sigma'_i$.

From the global hypothesis $\star_3$, the local conclusions (1) and (2), the case hypothesis, and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)}$ $\texttt{false} : t'_e$ " $((\textbf{E} - \textbf{WHILE}_\textbf{skip})$ and $(\textbf{E} - \textbf{WHILE}_{\textbf{false}_\top}))$, $\sigma'_i = \sigma'_o$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

This follows directly from the local conclusions (1) $(\sigma_o = \sigma_i)$, (3), and (a).

Case 2: $v' = \texttt{true}$

(a) $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup t'_e \;\vdash\; S_1 \; ; \; \textbf{while } e \textbf{ do } S_1 \textbf{ done} \;\Downarrow_{\mathcal{M}(O)}\; \sigma'_o \;:\; \rho'_o$

This follows from the global hypothesis $\star_3$, the local conclusions (1) and (2), the case hypothesis, and the definition of the rule applying to "**while** e **do** S_1 **done**" whenever e evaluates to $\texttt{true}$ $((\textbf{E} - \textbf{WHILE}_\textbf{true}))$.

(b) $x \notin \mathfrak{X}$.

From the local conclusion (1) $(\rho_e(x) \sqsubseteq \rho_o(x))$ and the global hypothesis $\star_4$, we get that $\rho_e(x) = \bot$. Hence, from the definition of ρ_e given in the local conclusion (1), $x \notin \mathfrak{X}$.

(c) $\sigma'_o(x) = \sigma'_i(x)$.

From the local conclusion (b), the hypothesis 5.2.1 applied to the analysis from the local conclusion (1) and the derivation from the local conclusion (a), $\sigma'_o(x) = \sigma'_i(x)$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

This follows directly from the local conclusions (1), (3), and (c).

$(\textbf{E} - \textbf{SEQUENCE})$ then we can conclude that :

(1) S is "$S_1 \; ; \; S_2$" and there exist σ_1 and ρ_1 such that:

- " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

- " $\sigma_1; \rho_1; t^{pc} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i; \rho_i; t^{pc} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

  This follows directly from the definition of the rule ($\mathbf{E}_{\mathcal{M}(O)}$ **− SEQUENCE**).

(2) There exist σ'_1 and ρ'_1 such that:

- " $\sigma'_i; \rho'_i; t^{pc'} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma'_1 : \rho'_1$ "
- " $\sigma'_1; \rho'_1; t^{pc'} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ "

This follows directly from the global hypothesis $\star_3$, the local conclusion (1), and the definition of the only rule applying to "$S_1\ ;\ S_2$" (($\mathbf{E}_{\mathcal{M}(O)}$ **− SEQUENCE**)).

(3) $\forall y : (\rho_1(y) = \bot) \Rightarrow \sigma_1(y) = \sigma'_1(y)$.
This result is obtained by applying the inductive hypothesis on " $\sigma_i; \rho_i; t^{pc} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " (sub-derivation tree of " $\sigma_i; \rho_i; t^{pc} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_i; \rho'_i; t^{pc'} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma'_1 : \rho'_1$ " for any variable y such that $\rho_1(y) = \bot$.

(•) $\sigma_o(x) = \sigma'_o(x)$.
The conclusion is obtained by applying the inductive hypothesis on " $\sigma_1; \rho_1; t^{pc} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (sub-derivation tree of " $\sigma_i; \rho_i; t^{pc} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_1; \rho'_1; t^{pc'} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ " using the local conclusion (3).

(**E − ASSIGN**) then we can conclude that :

(1) S is "*id* **:=** *e*".

Case 1: $x \neq id$

(a) $\sigma_o(x) = \sigma_i(x)$ and $\rho_o(x) = \rho_i(x)$.
This follows directly from the case hypothesis and the definition of the rule (**E − ASSIGN**).

(b) $\sigma'_o(x) = \sigma'_i(x)$.
This follows directly from the global hypothesis $\star_3$, the local conclusion (1), the definition of the only rule applying to "*id* **:=** *e*" ((**E − ASSIGN**)), and the case hypothesis.

(•) $\sigma_o(x) = \sigma'_o(x)$.
As $\rho_o(x) = \bot$, from the global hypothesis $\star_4$, the local conclusion (a) implies $\rho_i(x) = \bot$. Then, from the global hypothesis $\star_1$, $\sigma_i(x) = \sigma'_i(x)$. Hence, from the local conclusions (a) and (b), $\sigma_o(x) = \sigma'_o(x)$.

Case 2: $x = id$

(a) There exist v and t_e such that:

- " $\sigma_i; \rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : t_e$ "
- $\sigma_o(x) = v$
- $t_e \sqsubseteq \rho_o(x)$

This follows directly from the case hypothesis and the definition of the rule (**E − ASSIGN**).

(b) There exist v' and t'_e such that:

- " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

- $\sigma'_o(x) = v'$

  This follows directly from the global hypothesis $\star_3$, the local conclusion (1), the definition of the only rule applying to "*id* **:=** *e*" ((**E − ASSIGN**)), and the case hypothesis.

(c) $v = v'$.

This result comes from lemma C.2.1 applied to the derivations " $\sigma_i; \rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : t_e$ " and " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ ". It is possible to apply it because of the global hypothesis $\star_1$ and the fact that, from the global hypothesis $\star_4$ and the local conclusion (a), $t_e = \bot$.

(•) $\sigma_o(x) = \sigma'_o(x)$.

This follows directly from the local conclusions (a), (b), and (c).

(**E − SKIP**) then we can conclude that :

(1) S is "**skip**", $\sigma_o = \sigma_i$ and $\rho_o = \rho_i$.

This follows directly from the definition of the rule (**E − SKIP**).

(2) $\sigma'_o = \sigma'_i$.

This follows directly from the global hypothesis $\star_3$, the local conclusion (1), and the definition of the only rule applying to "**skip**" ((**E − SKIP**)).

(•) $\sigma_o(x) = \sigma'_o(x)$.

From the local conclusion (1) and the global hypothesis $\star_4$, $\rho_i(x) = \bot$. Then, from the global hypothesis $\star_1$, $\sigma_i(x) = \sigma'_i(x)$. Hence, from the local conclusions (1) and (2), $\sigma_o(x) = \sigma'_o(x)$.

C.2.2 Correction

Lemma C.2.4 (Expression tag is deterministic).
For all expressions e and tag stores ρ and ρ', if $\rho = \rho'$ then $\rho(e) = \rho'(e)$.

Proof. The proof is straightforward. It follows directly from the facts that the evaluation of an expression's tag is deterministic. It is the least upper bound of the tags of its free variables.

Lemma C.2.5 (Correctness of semantics $\Downarrow_{\mathcal{M}(O)}$ for information flow correction).
For all:

- *variable x,*
- *value stores σ_i, σ_o, σ'_i, and σ'_o,*
- *tag stores ρ_i, ρ_o, ρ'_i, and ρ'_o,*
- *tags t^{pc} and $t^{pc'}$,*
- *and statement S*

such that:

$\star_1$ $\forall y : (\rho_i(y) = \bot) \Rightarrow \sigma_i(y) = \sigma'_i(y)$,

$\star_2$ $\rho_i = \rho'_i$,

$\star_3$ $t^{pc} = t^{pc'}$,

$\star_4$ $\sigma_i;\rho_i;t^{pc} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$,

$\star_5$ $\sigma'_i;\rho'_i;t^{pc'} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$

it is true that:

- $\rho_o(x) = \rho'_o(x)$.

Proof. The proof goes by induction on the derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

$(\mathbf{E-IF}_\bot)$ then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exists $v \in \{true, false\}$ such that:

- " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : \bot$ "
- " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \bot \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule $(\mathbf{E-IF}_\bot)$.

(2) There exist a value v' and a tag t' such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'$ "

This follows from the global hypothesis $\star_5$, the local conclusion (1), and the definitions of the rules applying to "**if** e **then** S_{true} **else** S_{false} **end**" ($(\mathbf{E-IF}_\bot)$ and $(\mathbf{E-IF}_\top)$).

(3) $t' = \bot$ and $v = v'$.

This follows directly from the local conclusion (1) and (2), lemma C.2.4, and lemma C.2.1.

(4) " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup \bot \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ " This follows from the global hypothesis $\star_5$, the local conclusions (1), (2), and (3), and the definition of the rule applying to "**if** e **then** S_{true} **else** S_{false} **end**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v : \bot$ " ($(\mathbf{E-IF}_\bot)$).

(•) $\rho_o(x) = \rho'_o(x)$.

The conclusion is obtained by applying the inductive hypothesis on " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \bot \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup \bot \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ ".

$(\mathbf{E-IF}_\top)$ then we can conclude that :

(1) S is "**if** e **then** S_{true} **else** S_{false} **end**" and there exist $v \in \{true, false\}$, two tag stores ρ_v and ρ_e, and an analysis result $(\mathfrak{D}, \mathfrak{X})$ such that:

- " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : \top$ "

- " $\sigma_i;\rho_i;t^{\text{pc}}\sqcup\top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- $[\![\sigma_i;\rho_i \vdash S_{\neg v}]\!]^{\sharp\mathcal{G}} = (\mathfrak{D},\mathfrak{X})$
- $\rho_{\neg v} = \lambda x.\bigsqcup_{y\in\mathfrak{D}(x)}\rho_i(y)$
- $\rho_e = (\mathfrak{X}\times\{\top\})\cup((\mathbb{I}d-\mathfrak{X})\times\{\bot\})$
- $\rho_o = \rho_v \sqcup \rho_{\neg v} \sqcup \rho_e$

This follows directly from the definition of the rule ($\mathbf{E-IF}_\top$).

(2) There exist a value $v' \in \{true, false\}$ and a tag t'_e such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

This follows from the global hypothesis $\star_5$, the local conclusion (1), and the definitions of the rules applying to "**if** e **then** S_{true} **else** S_{false} **end**" (($\mathbf{E-IF}_\bot$) and ($\mathbf{E-IF}_\top$)).

(3) $t'_e = \top$.

This follows directly from the local conclusions (1) and (2), and the lemma C.2.4.

(4) There exists an analysis result $(\mathfrak{D}',\mathfrak{X}')$ and three tag stores $\rho'_{v'}$, $\rho'_{\neg v'}$ and ρ'_e such that:

- " $\sigma'_i;\rho'_i;t^{\text{pc}'}\sqcup t'_e \vdash S_{v'} \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_{v'}$ "
- $[\![\sigma'_i;\rho'_i \vdash S_{\neg v'}]\!]^{\sharp\mathcal{G}} = (\mathfrak{D}',\mathfrak{X}')$
- $\rho'_{\neg v'} = \lambda x.\bigsqcup_{y\in\mathfrak{D}'(x)}\rho'_i(y)$
- $\rho'_e = (\mathfrak{X}'\times\{\top\})\cup((\mathbb{I}d-\mathfrak{X}')\times\{\bot\})$
- $\rho'_o = \rho'_{v'} \sqcup \rho'_{\neg v'} \sqcup \rho'_e$

This follows from the global hypothesis $\star_5$, the local conclusions (1), (2), and (3), and the definition of the rule applying to "**if** e **then** S_{true} **else** S_{false} **end**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : \top$ " (($\mathbf{E-IF}_\top$)).

Case 1: $v = v'$

(a) $\rho_{\neg v}(x) = \rho'_{\neg v'}(x)$ and $\rho_e(x) = \rho'_e(x)$.

This follows from the definitions of $\rho_{\neg v}$, $\rho'_{\neg v'}$, ρ_e, and ρ'_e (given in the local conclusions (1) and (2)), the case hypothesis, the global hypothesis $\star_2$, and the hypothesis 5.2.3.

(b) $\rho_v(x) = \rho'_{v'}(x)$.

This result is obtained by applying the inductive hypothesis on " $\sigma_i;\rho_i;t^{\text{pc}}\sqcup\top \vdash S_v \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_v$ " (sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_i;\rho'_i;t^{\text{pc}'}\sqcup t'_e \vdash S_{v'} \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_{v'}$ " using the result of the local conclusion (3) and the case hypothesis.

(•) $\rho_o(x) = \rho'_o(x)$.

This follows directly from the definitions of ρ_o and ρ'_o (given in the local conclusions (1) and (4)) and the local conclusions (a) and (b).

Case 2: $v \neq v'$ (which implies that $v' = \neg v$)

(a) $\rho_v = \rho'_{\neg v'} \sqcup \rho'_e$.

This follows from the definitions of ρ_v, $\rho'_{\neg v'}$, and ρ'_e (given in the local conclusions (1) and (4)), the case hypothesis, and the hypotheses 5.2.3 and 5.2.2.

(b) $\rho'_{v'} = \rho_{\neg v} \sqcup \rho'_e$.
This follows from the definitions of $\rho'_{v'}$, $\rho_{\neg v}$, and ρ_e (given in the local conclusions (1) and (4)), the case hypothesis, the local conclusion (3) and the hypotheses 5.2.3 and 5.2.2.

(•) $\rho_o(x) = \rho'_o(x)$.
This follows directly from the definitions of ρ_o and ρ'_o (given in the local conclusions (1) and (4)) and the local conclusions (a) and (b).

(**E − WHILE**$_{\mathbf{skip}}$) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and:

- " $\sigma_i; \rho_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{false} : \bot$ "
- $\rho_i = \rho_o$

This follows directly from the definition of the rule (**E − IF**$_\bot$).

(2) There exist a value v' and a tag t' such that:

- " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'$ "

This follows from the global hypothesis $\star_5$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" ((**E − WHILE**$_{\mathbf{skip}}$), (**E − WHILE**$_{\mathbf{true}_\bot}$), (**E − WHILE**$_{\mathbf{true}_\top}$) and (**E − WHILE**$_{\mathbf{false}_\top}$)).

(3) $v' = \texttt{false}$ and $t' = \bot$.
This follows directly from the local conclusions (1) and (2), lemma C.2.1 and lemma C.2.4.

(4) $\rho'_o = \rho'_i$.
This follows from the global hypothesis $\star_5$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{false} : \bot$ " ((**E − WHILE**$_{\mathbf{skip}}$)).

(5) $\rho_i(x) = \rho'_i(x)$.
This follows directly from the global hypothesis $\star_2$.

(•) $\rho_o(x) = \rho'_o(x)$.
This follows directly from the local conclusions (1), (4), and (5).

(**E − WHILE**$_{\mathbf{true}_\bot}$) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and:

- " $\sigma_i; \rho_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{true} : \bot$ "
- " $\sigma_i; \rho_i; t^{\text{pc}} \sqcup \bot \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i; \rho_i; t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule (**E − WHILE**$_{\mathbf{true}_\bot}$).

(2) There exist a value v' and a tag t'_e such that:

- " $\sigma'_i; \rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

This follows from the global hypothesis $\star_5$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" ((**E − WHILE$_{\text{skip}}$**), (**E − WHILE$_{\text{true}_\perp}$**), (**E − WHILE$_{\text{true}_\top}$**) and (**E − WHILE$_{\text{false}_\top}$**)).

(3) $t'_e = \perp$ and $v' = true$.
This follows directly from the local conclusions (1) and (2), lemma C.2.4, and lemma C.2.1.

(4) " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup \perp \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ " This follows from the global hypothesis $\star_5$, the local conclusions (1), (2), and (3), and the definition of the rule applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} true : \perp$ " ((**E − WHILE$_{\text{true}_\perp}$**)).

(•) $\rho_o(x) = \rho'_o(x)$.
The conclusion is obtained by applying the inductive hypothesis on " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \perp \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup \perp \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ ".

(**E − WHILE$_{\text{true}_\top}$**) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and:

- " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)}$ `true` $: \top$ "
- " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \top \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_l$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- $\rho_o = \rho_i \sqcup \rho_l$

This follows directly from the definition of the rule (**E − WHILE$_{\text{true}_\top}$**).

(2) There exist a value v' and a tag t'_e such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

This follows from the global hypothesis $\star_5$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" ((**E − WHILE$_{\text{skip}}$**), (**E − WHILE$_{\text{true}_\perp}$**), (**E − WHILE$_{\text{true}_\top}$**) and (**E − WHILE$_{\text{false}_\top}$**)).

(3) $t'_e = \top$.
This follows directly from the local conclusions (1) and (2), and lemma C.2.4.

Case 1: $v' =$ `true`

(a) There exists a tag store ρ'_l such that:

- " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup \top \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_l$ "
- $\rho'_o = \rho'_i \sqcup \rho'_l$

This follows from the global hypothesis $\star_5$, the local conclusions (1), (2), and (3), the case hypothesis, and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)}$ `true` $: \top$ " ((**E − WHILE$_{\text{true}_\top}$**)).

(b) $\rho_l = \rho'_l$.
This is obtained by applying the inductive hypothesis on " $\sigma_i;\rho_i;t^{\text{pc}} \sqcup \top \vdash$

S_1 ; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)}$ σ_o : ρ_l " (sub-derivation tree of " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)}$ $\sigma_o : \rho_o$ ") and " $\sigma'_i;\rho'_i;t^{\mathrm{pc}'} \sqcup \top \vdash S_1$; **while** e **do** S_1 **done** $\Downarrow_{\mathcal{M}(O)}$ $\sigma'_o : \rho'_l$ ".

(•) $\rho_o = \rho'_o$.

This follows directly from the local conclusions (1), (a), and (b), and the global hypothesis $\star_2$.

Case 2: v' = `false`

(a) There exists an analysis result $(\mathfrak{D}', \mathfrak{X}')$ and two tag stores ρ'_l and ρ'_e such that:

- $[\![\sigma'_i;\rho'_i \vdash S_1$; **while** e **do** S_1 **done**$]\!]^{\sharp\mathcal{G}} = (\mathfrak{D}',\mathfrak{X}')$
- $\rho'_l = \lambda x. \bigsqcup_{y\in\mathfrak{D}'(x)} \rho'_i(y)$
- $\rho'_e = (\mathfrak{X}' \times \{\top\}) \cup ((\mathbb{I}d - \mathfrak{X}') \times \{\bot\})$
- $\rho'_o = \rho'_i \sqcup \rho'_l \sqcup \rho'_e$

This follows from the global hypothesis $\star_5$, the local conclusions (1), (2), and (3), the case hypothesis, and the definition of the rule applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)}$ `false` : $\top$ " ((**E – WHILE$_{\text{false}_\top}$**)).

(b) $\rho_l = \rho'_l \sqcup \rho'_e$.

This follows from the definitions of ρ_l, ρ'_l, and ρ'_e (given in the local conclusions (1) and (a)), and the hypotheses 5.2.3 and 5.2.2.

(•) $\rho_o = \rho'_o$.

This follows directly from the local conclusions (1), (a), and (b).

(**E – WHILE$_{\text{false}_\top}$**) then we can conclude that :

(1) S is "**while** e **do** S_1 **done**" and there exist two tag stores ρ_l and ρ_e such that:

- " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)}$ `false` : $\top$ "
- $[\![\sigma_i;\rho_i \vdash S_1$; **while** e **do** S_1 **done**$]\!]^{\sharp\mathcal{G}} = (\mathfrak{D}, \mathfrak{X})$
- $\rho_l = \lambda x. \bigsqcup_{y\in\mathfrak{D}(x)} \rho_i(y)$
- $\rho_e = (\mathfrak{X} \times \{\top\}) \cup ((\mathbb{I}d - \mathfrak{X}) \times \{\bot\})$
- $\rho_o = \rho_i \sqcup \rho_l \sqcup \rho_e$

This follows directly from the definition of the rule (**E – WHILE$_{\text{true}}$**).

(2) There exist a value $v' \in \{$`true`, `false`$\}$ and a tag t'_e such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "

This follows from the global hypothesis $\star_5$, the local conclusion (1), and the definitions of the rules applying to "**while** e **do** S_1 **done**" ((**E – WHILE$_{\text{skip}}$**), (**E – WHILE$_{\text{true}_\bot}$**), (**E – WHILE$_{\text{true}_\top}$**) and (**E – WHILE$_{\text{false}_\top}$**)).

(3) $t'_e = \top$.

This follows directly from the local conclusions (1) and (2), and lemma C.2.4.

Case 1: v' = `false`

(a) there exist two tag stores ρ'_l and ρ'_e such that:

- $[\![\sigma'_i;\rho'_i \vdash S_1 \text{ ; } \textbf{while } e \textbf{ do } S_1 \textbf{ done}]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}',\mathfrak{X}')$
- $\rho'_l = \lambda x. \bigsqcup_{y \in \mathfrak{D}'(x)} \rho'_i(y)$
- $\rho'_e = (\mathfrak{X}' \times \{\top\}) \cup ((\mathbb{I}d - \mathfrak{X}') \times \{\bot\})$
- $\rho'_o = \rho'_i \sqcup \rho'_l \sqcup \rho'_e$

This follows from the global hypothesis $\star_5$, the local conclusions (1), (2), and (3), the case hypothesis, and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{false} : \top$ " ((**E – WHILE**$_{\textbf{false}_\top}$)).

(b) $\rho_l = \rho'_l$ and $\rho_e = \rho'_e$.

This follows from the definitions of ρ_l, ρ'_l, ρ_e, and ρ'_e (given in the local conclusions (1) and (a)), the global hypothesis $\star_2$, and the hypothesis 5.2.3.

(•) $\rho_o = \rho'_o$.

This follows directly from the definitions of ρ_o and ρ'_o (given in the local conclusions (1) and (a)), the local conclusions (b), and the global hypothesis $\star_2$.

Case 2: $v' = \texttt{true}$

(a) There exists a tag store ρ'_l such that:

- " $\sigma'_i;\rho'_i;t^{\text{pc}'} \sqcup \top \vdash S_1 \text{ ; } \textbf{while } e \textbf{ do } S_1 \textbf{ done} \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_l$ "
- $\rho'_o = \rho'_i \sqcup \rho'_l$

This follows from the global hypothesis $\star_5$, the local conclusions (1), (2), and (3), the case hypothesis, and the definitions of the rules applying to "**while** e **do** S_1 **done**" whenever " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} \texttt{true} : \top$ " ((**E – WHILE**$_{\textbf{true}_\top}$)).

(b) $\rho'_l = \rho_l \sqcup \rho_e$.

This follows from the definitions of ρ'_l, ρ_l, and ρ_e (given in the local conclusions (1) and (a)), and the hypotheses 5.2.3 and 5.2.2.

(•) $\rho_o = \rho'_o$.

This follows directly from the local conclusions (1), (a), and (b), and the global hypothesis $\star_2$

(**E – SEQUENCE**) then we can conclude that :

(1) S is "S_1 ; S_2" and there exist σ_1 and ρ_1 such that:

- " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "
- " $\sigma_1;\rho_1;t^{\text{pc}} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " is a sub-derivation tree of " $\sigma_i;\rho_i;t^{\text{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ "

This follows directly from the definition of the rule (**E**$_{\mathcal{M}(O)}$ **– SEQUENCE**).

(2) There exist σ'_1 and ρ'_1 such that:

- " $\sigma'_i;\rho'_i;t^{\text{pc}'} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma'_1 : \rho'_1$ "
- " $\sigma'_1;\rho'_1;t^{\text{pc}'} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ "

This follows directly from the global hypothesis $\star_5$, the local conclusion (1), and the definition of the only rule applying to "S_1 ; S_2" (($\mathbf{E}_{\mathcal{M}(O)}$ **– SEQUENCE**)).

(3) $\forall y : (\rho_1(y) \sqsubseteq \bot) \Rightarrow \sigma_1(y) = \sigma'_1(y)$.

This result is obtained by applying lemma C.2.3 on " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " and " $\sigma'_i;\rho'_i;t^{\mathrm{pc}'} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma'_1 : \rho'_1$ " for any variable y such that $\rho_1(y) \sqsubseteq \bot$.

(4) $\rho_1 = \rho'_1$.

This result is obtained by applying the inductive hypothesis on " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma_1 : \rho_1$ " (sub-derivation tree of " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_i;\rho'_i;t^{\mathrm{pc}'} \vdash S_1 \Downarrow_{\mathcal{M}(O)} \sigma'_1 : \rho'_1$ " for any variable.

(•) $\rho_o(x) = \rho'_o(x)$.

The conclusion is obtained by applying the inductive hypothesis on " $\sigma_1;\rho_1;t^{\mathrm{pc}} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ " (sub-derivation tree of " $\sigma_i;\rho_i;t^{\mathrm{pc}} \vdash S \Downarrow_{\mathcal{M}(O)} \sigma_o : \rho_o$ ") and " $\sigma'_1;\rho'_1;t^{\mathrm{pc}'} \vdash S_2 \Downarrow_{\mathcal{M}(O)} \sigma'_o : \rho'_o$ " using the local conclusions (3) and (4).

(**E – ASSIGN**) then we can conclude that :

(1) S is "*id* **:=** *e*".

Case 1: $x \neq id$

(a) $\rho_o(x) = \rho_i(x)$.

This follows directly from the case hypothesis and the definition of the rule (**E – ASSIGN**).

(b) $\rho'_o(x) = \rho'_i(x)$.

This follows directly from the global hypothesis $\star_5$, the local conclusion (1), the definition of the only rule applying to "*id* **:=** *e*" ((**E – ASSIGN**)), and the case hypothesis.

(•) $\rho_o(x) = \rho'_o(x)$.

From the global hypothesis $\star_2$, $\rho_i(x) = \rho'_i(x)$. Hence, from the local conclusions (a) and (b), $\rho_o(x) = \rho'_o(x)$.

Case 2: $x = id$

(a) There exist v and t_e such that:

- " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : t_e$ "
- $\rho_o(x) = t_e \sqcup t^{\mathrm{pc}}$

This follows directly from the case hypothesis and the definition of the rule (**E – ASSIGN**).

(b) There exist v' and t'_e such that:

- " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ "
- $\rho'_o(x) = t'_e \sqcup t^{\mathrm{pc}'}$

This follows directly from the global hypothesis $\star_5$, the local conclusion (1), the definition of the only rule applying to "*id* **:=** *e*" ((**E – ASSIGN**)), and the case hypothesis.

(c) $t_e = t'_e$.

This result comes from lemma C.2.4 applied to the derivations " $\sigma_i;\rho_i \vdash e \Downarrow_{\mathcal{M}(O)} v : t_e$ " and " $\sigma'_i;\rho'_i \vdash e \Downarrow_{\mathcal{M}(O)} v' : t'_e$ ".

(•) $\rho_o(x) = \rho'_o(x)$.
From the global hypothesis $\star_3$, $t^{\mathrm{pc}'} = t^{\mathrm{pc}}$. Then, the above result follows directly from the local conclusions (a), (b), and (c).

(**E − SKIP**) then we can conclude that :

(1) S is "**skip**" and $\rho_o = \rho_i$.
This follows directly from the definition of the rule (**E − SKIP**).

(2) $\rho'_o = \rho'_i$.
This follows directly from the global hypothesis $\star_5$, the local conclusion (1), and the definition of the only rule applying to "**skip**" ((**E − SKIP**)).

(•) $\rho_o(x) = \rho'_o(x)$.
From the global hypothesis $\star_2$, $\rho_i(x) = \rho'_i(x)$. Hence, from the local conclusions (1) and (2), $\rho_o(x) = \rho'_o(x)$.

C.3 Transparency

Lemma C.3.1 (More Precise Static Analysis). *For all command C, value store σ, tag store ρ, and analysis result $(\mathfrak{D}, \mathfrak{X})$, if $(\mathfrak{D}, \mathfrak{X}) \models (\sigma, \rho \vdash C)$ then, for all variable x in $\mathfrak{X}$, x belongs to modified(C).*

Proof. The proof goes by structural induction on C and follows directly from the rules given in Figure 5.5 and the definition of *modified*(C).

Lemma C.3.2 (Existence of a Static Analysis Result). *For all command C, value store σ, and tag store ρ, there exists an analysis result $(\mathfrak{D}, \mathfrak{X})$ such that $[\![\sigma, \rho \vdash C]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X})$ and Hypotheses 5.2.3 and 5.2.4 hold.*

Proof. The proof goes easily by structural induction on C by constructing straightforwardly $(\mathfrak{D}, \mathfrak{X})$ from the constraints. The only difficult case is for while-statements to show that there exists an adequate solution to $\mathfrak{D} = \{\![\mathfrak{D}^1 \circ (\mathfrak{D} \cup \mathcal{I}d), \mathcal{I}d]\!\}^{\rho(e)}_{\sigma(e)}$.

Let $\mathcal{G}_l$, $\mathcal{G}_l^+$ and $\mathcal{G}_l^*$ be graphs whose nodes are $\mathbb{X}$. There exists an edge from x to y in $\mathcal{G}_l$ if and only if (x,y) belongs to $\mathfrak{D}^1$. $\mathcal{G}_l^+$ is the transitive closure of $\mathcal{G}_l$; and $\mathcal{G}_l^*$ is the reflexive closure of $\mathcal{G}_l^+$.

If $\rho(e) = \bot$ and $\sigma(e) = \texttt{false}$ then $\mathcal{I}d$is a solution for $\mathfrak{D}$. It is straightforward to see that it complies with all the requirements.

If $\rho(e) = \bot$ and $\sigma(e) = \texttt{true}$ then $\mathfrak{D}$ is the set of pairs (x,y) such that there exists an edge from x to y in $\mathcal{G}_l^+$. It is easy to see that (x,y) belongs to $\mathfrak{D}^1 \circ (\mathfrak{D} \cup \mathcal{I}d)$ if and only if there exists a node z such that there exists an edge (x,z) in $\mathcal{G}_l^*$ and an edge (z,y) in $\mathcal{G}_l$. If this is the case then, as $\mathcal{G}_l^+$ is the transitive closure of $\mathcal{G}_l$, there exists an edge (x,y) in $\mathcal{G}_l^+$; and therefore, (x,y) belongs to $\mathfrak{D}$. Therefore, $\mathfrak{D}^1 \circ (\mathfrak{D} \cup \mathcal{I}d)$ is a solution for $\mathfrak{D}$. From the construction mechanism, it is obvious that $(\mathfrak{D}, \mathfrak{X})$ complies with Hypothesis 5.2.3. If x does not belongs to $\mathfrak{X}^1$ then $\mathfrak{D}^1 = \{x\}$ (by induction). Therefore the sole edge in $\mathcal{G}_l$ whose origin is x has for destination x; and hence, there exists a sole edge whose origin is x in $\mathcal{G}_l^+$ (the destination of this edge being x itself). This allows to conclude that, as $\mathfrak{X} = \mathfrak{X}^1$, the Hypothesis 5.2.4 holds.

If $\rho(e) = \top$ then the set of pairs (x,y) such that there exists an edge from x to y in $\mathcal{G}_l^*$ is a solution for $\mathfrak{D}$. The proof then ends similarly to the preceding case.

Lemma C.3.3 (No Downgrading Under Secret Context). *For all command C, value store σ and σ', and automaton state (V, w) and (V', w') such that $w \notin \{\bot\}^\star$, if $((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V', w'), \sigma')$ then $V \subseteq V'$.*

Proof. This follows directly from lemma A.1.3 and the only transition rules applying to "`not` C" ((T-NOT-high) and (T-NOT-low)).

Lemma C.3.4 (WDKL (while-dilemma killer lemma) bis). *For all command C, expression e, automaton state $q = (V, w)$, and value store σ, if*

$\star_1$ *$FV(e) \cap V \neq \emptyset$,*

$\star_2$ $((V, w), \sigma) \vdash$ **while** e **do** C **done** $\overset{o}{\Rightarrow}_{M(s)} ((V_f, w), \sigma_f)$

then "$((V, w\top), \sigma) \vdash$ **while** e **do** C **done** $\overset{o}{\Rightarrow}_{M(s)} ((V_f, w\top), \sigma_f)$".

Proof. $\star_1$ implies that the automaton transition — Figure 3.3 — on `branch` e is (T-BRANCH-high). The definition of this transition in turn implies that the transition on `not` C, if it occurs, is (T-NOT-high).

Lemma C.3.5 (Context Sensitive Dynamic Analysis is More Precise). *Assumes that the dynamic information flow analysis uses an* acceptable *— Figure 5.5 — static analysis for which Hypotheses 5.2.3 and 5.2.4 hold. For all command C, value stores σ and σ', automaton states (V, w) and (V', w), outputs o, tag stores ρ, and tags t^{pc}, if:*

$\star_1$ $((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V', w), \sigma')$,

$\star_2$ $\forall x, (\rho(x) = \top) \Rightarrow x \in V$,

$\star_3$ $t^{pc} = \top \Rightarrow w \notin \{\bot\}^\star$,

then there exists a tag store ρ' such that:

- $\sigma; \rho; t^{pc} \vdash C \Downarrow \sigma' : \rho'$,
- $\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$.

Proof. The proof is done by induction on the size of the derivation tree of "$((V, w), \sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V', w), \sigma')$". Assume the lemma holds for any sub-derivation tree. As we consider the language without print statements, the rules (E$_{M(O)}$-EDIT) and (E$_{M(O)}$-NO) can not be applied. If the last semantic rule used is:

(E$_{M(O)}$-OK) then we can conclude that :

(1) • $((V, w), C) \xrightarrow{OK} (V', w)$,
• $\sigma \vdash C \overset{o}{\Rightarrow}_O \sigma'$,

- C is "**skip**" or "x **:=** e".

This follows from the $M(O)$ semantic rule ($\mathrm{E}_{M(O)}$-OK). This rule applies only to actions (skip or assignment statements).

(•) There exists a tag store ρ' such that "$\sigma; \rho; t^{\mathrm{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

Case 1: C is "**skip**".

(a) $V' = V$ and $\sigma' = \sigma$.
This follows from the case hypothesis, the only rule applying to "**skip**" in the O semantics (E_O-SKIP), and from the only transition in the automaton applying to "**skip**" (T-SKIP).

(b) $\sigma; \rho; t^{\mathrm{pc}} \vdash C \Downarrow \sigma : \rho$.
This follows directly from the rule ($\mathrm{E}_{M(O)}$-SKIP).

(c) $\forall x, (\rho(x) = \top) \Rightarrow x \in V'$.
This follows directly from the global hypothesis $\star_2$ and from the local conclusion (a).

(○) There exists a tag store ρ' such that "$\sigma; \rho; t^{\mathrm{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".
This follows from the local conclusions (b), (a), and (c).

Case 2: C is "x **:=** e"; and $w \notin \{\bot\}^\star$ or $FV(e) \cap V \neq \emptyset$.

(a) $V' = V \cup \{x\}$ and $\sigma' = \sigma[x \mapsto \sigma(e)]$.
This follows from the case hypothesis, the only rule applying to "x **:=** e" in the O semantics (E_O-ASSIGN), and the only transition in the automaton applying to "x **:=** e" whenever $w \notin \{\bot\}^\star$ or $FV(e) \cap V \neq \emptyset$ (T-ASSIGN-sec).

(b) $\sigma; \rho; t^{\mathrm{pc}} \vdash C \Downarrow \sigma[x \mapsto \sigma(e)] : \rho[x \mapsto t^{\mathrm{pc}} \sqcup \rho(e)]$.
This follows directly from the rule ($\mathrm{E}_{M(O)}$-ASSIGN) and Definition C.0.1.

(c) $\forall y,\ (\rho[x \mapsto t^{\mathrm{pc}} \sqcup \rho(e)](y) = \top) \Rightarrow y \in V'$.
From the local conclusion (a), x belongs to V'. For all variable y, different from x, $\rho[x \mapsto t^{\mathrm{pc}} \sqcup \rho(e)](y) = \top$ implies $\rho(y) = \top$. Hence, the global hypothesis $\star_2$ and the local conclusion (a) imply that y belongs to V'.

(○) There exists a tag store ρ' such that "$\sigma; \rho; t^{\mathrm{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".
This follows from the local conclusions (b), (a), and (c).

Case 3: C is "x **:=** e"; and $w \in \{\bot\}^\star$ and $FV(e) \cap V = \emptyset$.

(a) $V' = V - \{x\}$ and $\sigma' = \sigma[x \mapsto \sigma(e)]$.
This follows from the case hypothesis, the only rule applying to "x **:=** e" in the O semantics (E_O-ASSIGN), and the only transition in the automaton applying to "x **:=** e" whenever $w \in \{\bot\}^\star$ and $FV(e) \cap V = \emptyset$ (T-ASSIGN-sec).

(b) $\sigma; \rho; t^{\mathrm{pc}} \vdash C \Downarrow \sigma[x \mapsto \sigma(e)] : \rho[x \mapsto t^{\mathrm{pc}} \sqcup \rho(e)]$.
This follows directly from the rule ($\mathrm{E}_{M(O)}$-ASSIGN) and Definition C.0.1.

(c) $t^{\mathrm{pc}} \sqcup \rho(e) = \bot$.

From the case hypothesis and the global hypothesis $\star_3$, $t^{\mathrm{pc}} = \bot$. From the case hypothesis and the global hypothesis $\star_2$, $\rho(e) = \bot$. Those two properties implying the desired result.

(d) $\forall y,\ (\rho[x \mapsto t^{\mathrm{pc}} \sqcup \rho(e)](y) = \top) \Rightarrow y \in V'$.

From the local conclusion (c), the variable x respects the desired property. For all variable y, different from x, $\rho[x \mapsto t^{\mathrm{pc}} \sqcup \rho(e)](y) = \top$ implies $\rho(y) = \top$. Hence, the global hypothesis $\star_2$ and the local conclusion (a) imply that y belongs to V'.

($\circ$) There exists a tag store ρ' such that "$\sigma;\rho;t^{\mathrm{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

This follows from the local conclusions (b), (a), and (d).

($\mathbf{E}_{M(O)}$-IF) then we can conclude that :

(1) There exist automaton states (V_1, w_1), (V_2, w_1), and (V_3, w_2) such that:

- C is "**if** e **then** C_{true} **else** C_{false} **end**",
- $\sigma(e) = v$,
- $((V, w),\ \texttt{branch}\ e) \xrightarrow{OK} (V_1, w_1)$,
- $((V_1, w_1),\ \sigma) \vdash C_v \overset{o}{\Rightarrow}_{M(O)} ((V_2, w_1),\ \sigma')$,
- $((V_2, w_1),\ \texttt{not}\ C_{\neg v}) \xrightarrow{OK} (V_3, w_2)$,
- $((V_3, w_2),\ \texttt{exit}) \xrightarrow{OK} (V', w)$.

This follows directly from the rule ($\mathrm{E}_{M(O)}$-IF) and lemma A.1.1.

(2) $V_1 = V$ and $(w \notin \{\bot\}^\star \Rightarrow w_1 \notin \{\bot\}^\star)$.

This follows directly from the only two transitions applying to "$\texttt{branch}\ e$" ((T-BRANCH-high) and (T-BRANCH-low)).

(3) There exists a tag t_e such that $\rho(e) = t_e$.

This trivially follows from the local conclusion (1) considering that ρ and σ have the same domain.

($\bullet$) There exists a tag store ρ' such that "$\sigma;\rho;t^{\mathrm{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

Case 1: $t_e = \bot$.

(a) There exists a tag store ρ_v such that "$\sigma;\rho;t^{\mathrm{pc}} \vdash C_v \Downarrow \sigma' : \rho_v$" and "$\forall x, (\rho_v(x) = \top) \Rightarrow x \in V_2$".

This follows from the inductive hypothesis, the local conclusions (1) and (2), and the global hypotheses $\star_2$ and $\star_3$.

(b) $\sigma;\rho;t^{\mathrm{pc}} \vdash C \Downarrow \sigma' : \rho_v$.

This follows from the rule ($\mathrm{E}_{M(O)}$-IF$_\bot$), the case hypothesis, and the local conclusion (a).

(c) $\forall x, (\rho_v(x) = \top) \Rightarrow x \in V'$.

From the local conclusion (1) and the transition rules (T-NOT-high), (T-NOT-low), and (T-EXIT), $V_2 \subseteq V'$. Hence, the desired result follow from the local conclusion (a).

(○) There is a tag store ρ' such that "$\sigma;\rho;t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

This follows from the local conclusions (b) and (c).

Case 2: $t_e = \top$.

(a) $FV(e) \cap V \neq \emptyset$.

This follows from the local conclusion (3), Definition C.0.1, the case hypothesis, and the global hypothesis $\star_2$.

(b) $w_1 \notin \{\bot\}^\star$.

Transition rule (T-BRANCH-high) and the local conclusions (1) and (a) imply $w_1 \notin \{\bot\}^\star$.

(c) There is a tag store ρ_v such that "$\sigma;\rho;\top \vdash C_v \Downarrow \sigma' : \rho_v$" and "$\forall x, (\rho_v(x) = \top) \Rightarrow x \in V_2$".

This follows from the inductive hypothesis, the local conclusions (1), (2) and (b), and the global hypothesis $\star_2$.

(d) $V' = V_2 \cup \mathit{modified}(C_{\neg v})$.

The local conclusions (1) and (b) and the transition rules (T-NOT-high) and (T-EXIT) imply the desired result.

Let the analysis result $(\mathfrak{D}, \mathfrak{X})$ be such that $[\![\sigma, \rho \vdash C_{\neg v}]\!]^{\sharp\mathcal{G}} = (\mathfrak{D}, \mathfrak{X})$, and the tag stores $\rho_{\neg v}$ and ρ_e be such that $\rho_{\neg v} = \lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)$ and $\rho_e = (\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\})$.

(e) There exists a tag store ρ_f such that "$\sigma;\rho;t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho_f$" and "$\rho_f = \rho_v \sqcup \rho_{\neg v} \sqcup \rho_e$".

This follows from the rule ($\text{E}_{\mathcal{M}(O)}$-$\text{IF}_\top$), the case hypothesis, the local conclusions (3) and (c), and the lemma C.3.2.

(f) $\forall x, (\rho_f(x) = \top) \Rightarrow x \in V'$.

For all variable x such that $\rho_v(x) = \top$, the local conclusions (c) and (d) imply that x belongs to V'. For all variable x such that $\rho_e(x) = \top$, lemma C.3.1 implies that x belongs to $\mathit{modified}(C_{\neg v})$; and hence, the local conclusion (d) implies that x belongs to V'. For all variable x such that $\rho_{\neg v}(x) = \top$ and $\mathfrak{D}(x) \neq \{x\}$, Hypothesis 5.2.4 implies that x belongs to $\mathfrak{X}$; and hence, from what has been said before, x belongs to V'. For all variable x such that $\rho_{\neg v}(x) = \top$ and $\mathfrak{D}(x) = \{x\}$, from the definition of $\rho_{\neg v}$, $\rho(x) = \top$; which, combined with the global hypothesis $\star_2$, the local conclusions (2), (b), and (1), lemma C.3.3, and local conclusion (d), implies that x belongs to V'.

(○) There is a tag store ρ' such that "$\sigma;\rho;t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

This follows from the local conclusions (e) and (f).

($\text{E}_{\mathcal{M}(O)}$-$\text{WHILE}_{true}$) then we can conclude that :

(1) There exist a value store σ_1 and automaton states (V, w_1), (V_1, w_1), and (V_1, w) such that:

- C is "**while** e **do** C_l **done**",
- $\sigma(e) = \texttt{true}$,
- $((V, w),\ \texttt{branch}\ e) \xrightarrow{OK} (V, w_1)$,
- $((V, w_1),\ \sigma) \vdash C_l \overset{o_l}{\Longrightarrow}_{M(O)} ((V_1, w_1),\ \sigma_1)$,
- $((V_1, w_1),\ \texttt{exit}) \xrightarrow{OK} (V_1, w)$,
- $((V_1, w),\ \sigma_1) \vdash$ **while** e **do** C_l **done** $\overset{o_w}{\Longrightarrow}_{M(O)} ((V', w),\ \sigma')$.

This follows directly from the rule ($\text{E}_{M(O)}$-WHILE$_{true}$), the transitions of the monitoring automaton, and lemma A.1.1.

(2) There exists a tag t_e such that $\rho(e) = t_e$.
This trivially follows from the local conclusion (1) considering that ρ and σ have the same domain.

(3) $(t_e = \top) \Rightarrow FV(e) \cap V \neq \emptyset$.
This follows from the local conclusion (2), Definition C.0.1, and the global hypothesis $\star_2$.

(4) There exists a tag store ρ_1 such that "$\sigma; \rho; t^{\text{pc}} \sqcup t_e \vdash C_l \Downarrow \sigma_1 : \rho_1$" and "$\forall x, (\rho_1(x) = \top) \Rightarrow x \in V_1$".
From the local conclusion (1) and the transitions (T-BRANCH-high) and (T-BRANCH-low), there exists $a \in \{\top, \bot\}$ such that $w_1 = wa$. Thus, from the global hypothesis $\star_3$, $(t^{\text{pc}} = \top) \Rightarrow w_1 \notin \{\bot\}^\star$. From the transition (T-BRANCH-high) and the local conclusions (3) and (1), $(t_e = \top) \Rightarrow w_1 \notin \{\bot\}^\star$. Hence, $(t^{\text{pc}} \sqcup t_e = \top) \Rightarrow w_1 \notin \{\bot\}^\star$. Then, using the inductive hypothesis, the local conclusion (1) and the global hypothesis $\star_2$ imply the desired result.

(•) There exists a tag store ρ' such that "$\sigma; \rho; t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

Case 1: $t_e = \bot$.

(a) There is ρ_2 such that "$\sigma_1; \rho_1; t^{\text{pc}} \vdash$ **while** e **do** C_l **done** $\Downarrow \sigma' : \rho_2$" and "$\forall x, (\rho_2(x) = \top) \Rightarrow x \in V'$".
This result is implied by the inductive hypothesis, the local conclusions (1) and (4), and the global hypothesis $\star_3$.

(b) $\sigma; \rho; t^{\text{pc}} \vdash C_l$; **while** e **do** C_l **done** $\Downarrow \sigma' : \rho_2$.
This follows directly from the case hypothesis, the semantic rule ($\text{E}_{\mathcal{M}(O)}$-SEQUENCE) and the local conclusions (4) and (a).

(∘) There is a tag store ρ' such that "$\sigma; \rho; t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".
This follows from the semantic rule ($\text{E}_{\mathcal{M}(O)}$-WHILE$_{true_\bot}$), the case hypothesis, and the local conclusions (2), (b), and (a).

Case 2: $t_e = \top$.

(a) $((V_1, w\top),\ \sigma_1) \vdash$ **while** e **do** C_l **done** $\overset{o_w}{\Longrightarrow}_{M(O)} ((V', w\top),\ \sigma')$.
This follows directly from the local conclusion (1) and lemma C.3.4.

(b) There is ρ_2 such that "$\sigma_1; \rho_1; t^{\text{pc}} \sqcup t_e \vdash$ **while** e **do** C_l **done** $\Downarrow \sigma' : \rho_2$" and "$\forall x, (\rho_2(x) = \top) \Rightarrow x \in V'$".

This result is implied by the inductive hypothesis, the local conclusions (a) and (4), and the fact that $w\top$ does not belong to $\{\bot\}^\star$.

(c) $\sigma; \rho; t^{\text{pc}} \sqcup t_e \vdash C_l$; **while** e **do** C_l **done** $\Downarrow \sigma' : \rho_2$.

This follows directly from the case hypothesis, the semantic rule (E$_{\mathcal{M}(O)}$-SEQUENCE) and the local conclusions (4) and (b).

(d) $\forall x, (\rho(x) = \top) \Rightarrow x \in V'$.

From the case hypothesis, the local conclusions (3) and (1), and the transition (T-BRANCH-high), $w_1 \notin \{\bot\}^\star$. Hence, from lemma C.3.3, $V \subseteq V_1$. From lemma A.1.5, the case hypothesis, and the local conclusions (3) and (1), $V_1 \subseteq V'$. Thus, the desired result follows from the global hypothesis $\star_2$.

(○) There is a tag store ρ' such that "$\sigma; \rho; t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

This follows from the semantic rule (E$_{\mathcal{M}(O)}$-WHILE$_{true_\top}$), the case hypothesis, and the local conclusions (2), (c), (d) and (b).

(E$_{M(O)}$-WHILE$_{false}$) then we can conclude that :

(1) There exist automaton states (V, w_1) and (V', w_1) such that:

- C is "**while** e **do** C_l **done**",
- $\sigma(e) = \texttt{false}$,
- $((V, w), \texttt{branch}\ e) \xrightarrow{OK} (V, w_1)$,
- $((V, w_1), \texttt{not}\ C_l) \xrightarrow{OK} (V', w_1)$,
- $((V', w_1), \texttt{exit}) \xrightarrow{OK} (V', w)$,
- $\sigma' = \sigma$.

This follows directly from the rule (E$_{M(O)}$-WHILE$_{false}$) and the transitions of the monitoring automaton.

(2) There exists a tag t_e such that $\rho(e) = t_e$.

This trivially follows from the local conclusion (1) considering that ρ and σ have the same domain.

(•) There exists a tag store ρ' such that "$\sigma; \rho; t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

Case 1: $t_e = \bot$.

(a) $\sigma; \rho; t^{\text{pc}} \vdash C \Downarrow \sigma : \rho$.

This follows from the rule (E$_{\mathcal{M}(O)}$-WHILE$_{skip}$), the case hypothesis, and the local conclusion (2).

(b) $\forall x, (\rho(x) = \top) \Rightarrow x \in V'$.

From the local conclusion (1) and the transition rules (T-NOT-high) and (T-NOT-low), $V \subseteq V'$. Hence, the desired result follows from the global hypothesis $\star_2$.

($\circ$) There is a tag store ρ' such that "$\sigma;\rho;t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

This follows from the local conclusions (a), (1) and (b).

Case 2: $t_e = \top$.

(a) $FV(e) \cap V \neq \emptyset$.

This follows from the local conclusion (2), Definition C.0.1, the case hypothesis, and the global hypothesis $\star_2$.

(b) $V' = V \cup \mathit{modified}(C_l)$.

Transition rule (T-BRANCH-high) and the local conclusions (1) and (a) imply $w_1 \notin \{\bot\}^\star$. Hence, the local conclusion (1) and transition (T-NOT-high) imply the desired result.

Let the analysis result $(\mathfrak{D}, \mathfrak{X})$ be such that $[\![\sigma, \rho \vdash C_l \text{ ; } \textbf{while } e \textbf{ do } C_l \textbf{ done}]\!]^{\sharp_\mathcal{G}} = (\mathfrak{D}, \mathfrak{X})$, and the tag stores ρ_l and ρ_e be such that $\rho_l = \lambda x. \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)$ and $\rho_e = (\mathfrak{X} \times \{\top\}) \cup (\mathfrak{X}^c \times \{\bot\})$.

(c) There exists a tag store ρ_f such that "$\sigma;\rho;t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho_f$" and "$\rho_f = \rho \sqcup \rho_l \sqcup \rho_e$".

This follows from the rule ($\text{E}_{\mathcal{M}(O)}$-WHILE$_{false_\top}$), the case hypothesis, the local conclusions (2) and (1), and the Lemma C.3.2.

(d) $\forall x, (\rho_f(x) = \top) \Rightarrow x \in V'$.

For all variable x such that $\rho(x) = \top$, the global hypothesis $\star_2$ and the local conclusion (b) imply that x belongs to V'. For all variable x such that $\rho_e(x) = \top$, lemma C.3.1 implies that x belongs to $\mathit{modified}(C_l)$; and hence, the local conclusion (b) implies that x belongs to V'. For all variable x such that $\rho_l(x) = \top$ and $\mathfrak{D}(x) \neq \{x\}$, Hypothesis 5.2.4 implies that x belongs to $\mathfrak{X}$; and hence, from what has been said before, x belongs to V'. For all variable x such that $\rho_l(x) = \top$ and $\mathfrak{D}(x) = \{x\}$, from the definition of ρ_l, $\rho(x) = \top$; which, combined with the global hypothesis $\star_2$ and the local conclusion (b), implies that x belongs to V'.

($\circ$) There is a tag store ρ' such that "$\sigma;\rho;t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".

This follows from the local conclusions (c) and (d).

($\text{E}_{M(O)}$-SEQ) then we can conclude that :

(1) There exists a value store σ_1 and an automaton state (V_1, w) such that:

- C is "C_1 ; C_2",
- $((V, w), \sigma) \vdash C_1 \overset{o_1}{\Longrightarrow}_{M(O)} ((V_1, w_1), \sigma_1)$,
- $((V_1, w), \sigma_1) \vdash C_2 \overset{o_2}{\Longrightarrow}_{M(O)} ((V', w), \sigma')$.

This follows directly from the rule ($\text{E}_{M(O)}$-SEQ) and lemma A.1.1.

(2) There exists a tag store ρ_1 such that "$\sigma;\rho;t^{\text{pc}} \vdash C_1 \Downarrow \sigma_1 : \rho_1$" and "$\forall x, (\rho_1(x) = \top) \Rightarrow x \in V_1$".

This follows from the inductive hypothesis, the global hypotheses $\star_2$ and $\star_3$, and the local conclusion (1).

(3) There exists a tag store ρ' such that "$\sigma_1; \rho_1; t^{\text{pc}} \vdash C_2 \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".
This follows from the inductive hypothesis, the global hypothesis $\star_3$, and the local conclusions (1) and (2).

(•) There exists a tag store ρ' such that "$\sigma; \rho; t^{\text{pc}} \vdash C \Downarrow \sigma' : \rho'$" and "$\forall x, (\rho'(x) = \top) \Rightarrow x \in V'$".
This follows from the rule (E$_{\mathcal{M}(O)}$-SEQUENCE) and the local conclusions (1), (2), and (3).

Index of Citations