A FORMAL APPROACH TO CONTRACT VERIFICATION FOR HIGH-INTEGRITY APPLICATIONS

by

ZHI ZHANG

B.A., Nanjing University of Posts and Telecommunications, China, 2007
M.S., Nanjing University, China, 2010

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

2016
Abstract

High-integrity applications are safety- and security-critical applications developed for a variety of critical tasks. The correctness of these applications must be thoroughly tested or formally verified to ensure their reliability and robustness. The major properties to be verified for the correctness of applications include: (1) functional properties, capturing the expected behaviors of a software, (2) dataflow property, tracking data dependency and preventing secret data from leaking to the public, and (3) robustness property, the ability of a program to deal with errors during execution.

This dissertation presents and explores formal verification and proof technique, a promising technique using rigorous mathematical methods, to verify critical applications from the above three aspects. Our research is carried out in the context of SPARK, a programming language designed for development of safety- and security-critical applications.

First, we have formalized in the Coq proof assistant the dynamic semantics for a significant subset of the SPARK 2014 language, which includes run-time checks as an integral part of the language, as any formal methods for program specification and verification depend on the unambiguous semantics of the language.

Second, we have formally defined and proved the correctness of run-time checks generation and optimization based on SPARK reference semantics, and have built the certifying tools within the mechanized proof infrastructure to certify the run-time checks inserted by the GNAT compiler frontend to guarantee the absence of run-time errors.

Third, we have proposed a language-based information security policy framework and the associated enforcement algorithm, which is proved to be sound with respect to the formalized program semantics. We have shown how the policy framework can be integrated into SPARK 2014 for more advanced information security analysis.
A FORMAL APPROACH TO CONTRACT VERIFICATION FOR HIGH-INTEGRITY APPLICATIONS

by

ZHI ZHANG

B.A., Nanjing University of Posts and Telecommunications, China, 2007
M.S., Nanjing University, China, 2010

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

2016

Approved by:

Major Professor
John Hatcliff
Abstract

High-integrity applications are safety- and security-critical applications developed for a variety of critical tasks. The correctness of these applications must be thoroughly tested or formally verified to ensure their reliability and robustness. The major properties to be verified for the correctness of applications include: (1) functional properties, capturing the expected behaviors of a software, (2) dataflow property, tracking data dependency and preventing secret data from leaking to the public, and (3) robustness property, the ability of a program to deal with errors during execution.

This dissertation presents and explores formal verification and proof technique, a promising technique using rigorous mathematical methods, to verify critical applications from the above three aspects. Our research is carried out in the context of SPARK, a programming language designed for development of safety- and security-critical applications.

First, we have formalized in the Coq proof assistant the dynamic semantics for a significant subset of the SPARK 2014 language, which includes run-time checks as an integral part of the language, as any formal methods for program specification and verification depend on the unambiguous semantics of the language.

Second, we have formally defined and proved the correctness of run-time checks generation and optimization based on SPARK reference semantics, and have built the certifying tools within the mechanized proof infrastructure to certify the run-time checks inserted by the GNAT compiler frontend to guarantee the absence of run-time errors.

Third, we have proposed a language-based information security policy framework and the associated enforcement algorithm, which is proved to be sound with respect to the formalized program semantics. We have shown how the policy framework can be integrated into SPARK 2014 for more advanced information security analysis.
## 3 Formal Semantics of SPARK 2014

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Syntax of Core SPARK 2014</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Run-Time Checks</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Formal Language Semantics</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1 SPARK AST Representation</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2 State/Value</td>
<td>38</td>
</tr>
<tr>
<td>3.3.3 Expression Semantics</td>
<td>38</td>
</tr>
<tr>
<td>3.3.4 Statement Semantics</td>
<td>41</td>
</tr>
<tr>
<td>3.3.5 Declaration Semantics</td>
<td>44</td>
</tr>
<tr>
<td>3.3.6 Evaluation</td>
<td>45</td>
</tr>
</tbody>
</table>

## 4 Run-Time Check Certification

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Architecture</td>
<td>46</td>
</tr>
<tr>
<td>4.2 Certified Run-Time Check Generator</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Certified Run-Time Check Optimizer</td>
<td>52</td>
</tr>
<tr>
<td>4.4 Certifying GNAT RT Generator</td>
<td>60</td>
</tr>
<tr>
<td>4.5 Evaluation: Certifying GNAT</td>
<td>61</td>
</tr>
</tbody>
</table>

## 5 Declassification Policy for SPARK

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Syntax of Security Policy</td>
<td>70</td>
</tr>
<tr>
<td>5.1.1 Policy Design Principles</td>
<td>70</td>
</tr>
<tr>
<td>5.1.2 Policy Aspects</td>
<td>70</td>
</tr>
<tr>
<td>5.2 Enforcement of Security Policy</td>
<td>74</td>
</tr>
<tr>
<td>5.2.1 Type Constraint Generation</td>
<td>75</td>
</tr>
<tr>
<td>5.2.2 Type Constraint Simplification</td>
<td>83</td>
</tr>
<tr>
<td>5.2.3 Type Constraint Satisfiability</td>
<td>85</td>
</tr>
<tr>
<td>5.3 Example</td>
<td>87</td>
</tr>
</tbody>
</table>
List of Figures

3.1 Core SPARK Syntax (Part 1) ............................................. 29
3.2 Core SPARK Syntax (Part 2) ............................................. 30
3.3 Core SPARK Syntax (Part 3) ............................................. 31
3.4 SPARK AST in Coq (Part 1) ............................................ 34
3.5 SPARK AST in Coq (Part 2) ............................................ 35

4.1 Architectural Overview ................................................. 47
4.2 Interval Bounds For Divide and Modulus .......................... 53
4.3 RT-OPT Specification for Expression (excerpts) ............... 54
4.4 RT-OPT Specification for Array and Record Access ........... 55
4.5 RT-OPT Specification for Statement ................................ 56
4.6 RT-OPT Specification for Arguments ............................... 57

5.1 Security Type Inference for Expression ............................ 75
5.2 Type Constraints Generation Rules ................................. 76
5.3 Type Constraints for While Loop ................................... 79
5.4 Illustration of Declassification Policy Framework ............... 88
5.5 Operational Semantics .................................................. 91
List of Tables

4.1 Experiment Data .............................................................. 62

5.1 Experiment Data .............................................................. 99
Acknowledgments

First, I really want to thank my advisor Dr. John Hatcliff for his time and patience for advising my research work and his support for my internships at different industrial companies and research institutes worldwide, opening the door to my future career path. As a distinguished professor, he will always be my role model in my future career, to be confident, diligent, perseverant and excellent.

Second, thanks to Dr. Torben Amtoft, a nice and well-respected teacher, for the regular meeting and discussion every week to make progress for my research topic, and Dr. Robby, a pursuit of perfection in research and a great software engineer, for giving me a lot of help and advice during my PhD research.

Thanks to all my committee members: Dr. Xinming Ou, Dr. Andrew G. Bennett, and Dr. Margaret Rys for your review and feedback for my PhD work.

Thanks to Dr. Yannick Moy from AdaCore, Dr. Pierre Courtieu, Dr. Maria Virginia Aponte and Dr. Tristan Crolard from CNAM for the cooperation work on SPARK.

Thanks to all the friends at K-State.

Finally, I want to express my sincere gratitude to my family. Wish you health and happiness!
Chapter 1

Introduction

Challenges  Today, more and more software-dependent applications are being developed and they are widely used to better our life. At the same time, they may also cause disaster, especially for those safety critical and security critical softwares used in the area of finance, healthcare, transportation, defence and so on. Examples include password checking systems, online shopping and banking systems, medical device coordinating systems, railway and aircraft control systems, military system and control systems of nuclear power plants. Any errors in these critical systems will endanger personal life safety, company financial security, national security and environment safety. How can we prevent such catastrophic events and trust that these critical softwares will always perform correctly? To solve this problem, stringent software certification standards have been defined for several different critical domains (e.g., DO-178C/DO-333 for avionics, ISO 2626 for automotive, and EN50128 for railways). In the case of avionics, DO-333, a supplement of DO-178C on formal methods, allows testing to be partly replaced by formal verification, which has been successfully applied in some projects at Dassault-Aviation and Airbus\(^1\). These standards only provide a guidance for software verification procedure, but concrete verification techniques completely satisfying these standards are still a long way to go.
**Correctness Properties** Critical softwares are usually developed for a variety of critical tasks, so their correctness must be thoroughly tested or formally verified to ensure their reliability and robustness. The major properties to be verified for the correctness of software include: functional property, dataflow property, and robustness property.

Functional Property. It is a property about a program’s intended behaviors, which are usually expressed with pre and post conditions. The precondition describes the condition that must always be true for the input variables when the program is called, and the postcondition is a condition that will always hold when the program returns. Therefore, pre and postconditions specify a contract between a program and its users, whenever the precondition is satisfied by the users, the program will promise to always return a result satisfying its postcondition.

Dataflow Property. It’s a property about data dependency between program input and output variables. It can be used for code optimization (e.g. dead code detection), ensure the property of noninterference (e.g. prevention of leaking secret information) and so on.

Robustness Property. It is a property about the ability of a program to deal with errors during execution. One major robustness property is to ensure the absence of runtime errors, such as divide by zero, overflow, out of range and so on. They should be handled correctly, otherwise the program may get into abnormal state and return unexpected result.

**Verification Methods** Automatic methods to ensure software correctness include static analysis, performed during compilation phase, dynamic analysis, performed during program execution, and their combination. Static analysis techniques range from simple type checks to sophisticated formal methods, such as data-flow analysis, Hoare logic, model checking and symbolic execution. Contrast to dynamic analysis, such as testing which is driven by test cases and is unlikely to cover and test all possible program behaviors, formal methods apply rigorous mathematical approach to prove the correctness of software by building some mathematic models for the software and using mathematic reasoning logic to reason about its possible behaviors against some correctness constraints. Therefore, verification techniques
based on formal methods are able to explore all possible program states and conclusively prove the absence of certain errors.

1.1 Motivation

Formal Language Semantics  Formal verification techniques applies formal methods of mathematics to prove the correctness of programs, and all assumptions related to each formal analysis method should be described and justified in order to make it sound. As any formal static analysis must rely on the behavior of the language being analyzed to explore all possible behaviors of a program, a precise and unambiguous definition of the semantics of the language becomes clearly a requirement in the certification process. Furthermore, the correctness of program static analysis itself and program translations can be also proved with respect to the language semantics.

The semantics of language is so important, for both correct program execution and sound program analysis, that all programming languages come with a reference manual defining semantics for each language structure. However, the semantics in this form (written in natural language descriptions) is quite informal and it may lead to semantic ambiguity (different implementor has different interpretation). To solve this problem, it’s required to precisely define the formal semantics for a language in a rigorous mathematical way. Approaches to defining formal semantics of programming language include denotational semantics, operational semantics and axiomatic semantics.

Certification of Static Analysis  There are various program static analysis and verification techniques that have been proposed to automatically check the correctness of the program with respect to the user supplied contracts. Usually, the reasoning logic and the implementation algorithms behind these static analysis methods are assumed to be correct and their claimed arguments for the analyzed program are trusted to be true. For critical applications with demand of high level reliability and robustness, these assumptions should
be justified to convince the correctness of the analysis methods themselves. More and more researchers have realized this problem, and a promising approach is to have their correctness formally proved within some mathematical proof tool, or even generate convincing evidence for their claimed verification results.

**Enforceable and Sound Information Security Checking Framework**  Information security is concerned with confidentiality, integrity, and availability of information\(^2\). Unlike functionality requirements defining what a software is supposed to accomplish, the requirements of information security is implicit. Current methodologies for software development provide little assurance that information security requirements are satisfied. A policy specification language for information security at the source code level is needed to explicitly and precisely specify the expected information security requirements. A security policy framework, with both expressive policy specification language and enforceable policy checking algorithm proved to be sound based on the interpretation of the programming language, can provide a high assurance that an implementation of a program satisfies the program’s security requirements.

**Formal Verification of Robustness**  Critical softwares are required to be robust to perform well not only under ordinary conditions but also under unusual conditions. The robustness of a software is almost as important as its functionality to achieve high levels of reliability. Sometimes, it can cause serious, even catastrophic, results. A well-known example is the explosion of the Ariane 5 launcher on its maiden flight in 1996 because of a software runtime error (out of range, overflow), which caused a loss of approximately DM 1200 million\(^3\). As the number and complexity of critical softwares are growing, the common testing based verification techniques will no longer be sufficient to ensure the robustness. While formal verification is a promising technique to enable proving the absence of errors for softwares executing in unusual conditions.
1.2 Related Work

In this section, we provide a survey of the work related to this dissertation, including the efforts made towards the formalization of language semantics, state-of-the-art work on formal verification methods, recent work on language-based information flow security policies, and the work on assurance of absence of run-time errors.

1.2.1 Formalization of Programming Language Semantics

Much work has been done on the formalization of programming language semantics, but most are for toy languages for academic purposes, and it’s rare to see the formal semantics for real programming languages. The following are the only examples that we know coming with formal semantics.

Clight\textsuperscript{4}, a large subset of the C programming language that have been formalized in the Coq proof assistant in CompCert project\textsuperscript{5} in INRIA. CompCert\textsuperscript{6} is a formally verified C compiler with proof of semantic preservation in program translation from source to target language. To achieve the correctness proof of the CompCert C compiler, the formal semantics of Clight and a series of intermediate languages during its translation to machine code have been formally defined in inductive small-step relations in Coq.

Tahina Ramananandro has formalized the semantics of a small subset of C++ using Coq, focusing on the C++ object model, including object construction and destruction\textsuperscript{7,8}. This is a work towards a formalization of a verified compiler for the object-oriented subset of C++ within the framework of the CompCert.

The Standard ML\textsuperscript{9} is a general-purpose functional language that comes with formal semantics in mathematical notation, where the rules of evaluation for each phrase of the language are formally specified.

C, C++ and Standard ML are all general-purpose programming language, none of them are designed for development of critical softwares. As far as we know, SPARK is the only
commercially supported programming language with a set of associated verification tool set targeted specially to the development of critical softwares. SPARK formal semantics were previously defined for SPARK 83\textsuperscript{10,11}. This definition includes both the static and the dynamic semantics of the language based on a precise notation inspired by the Z notation. But these semantics are manually defined on paper, no tool was used to check the soundness of these definition. What’s more, SPARK has evolved a lot since then, so a new and more rigorous semantic formalization for SPARK becomes imperative for machine-checkable proof of program correctness.

### 1.2.2 Certified Static Analysis

We have previously developed an information flow verification framework for machine-checked proofs of programs compliance to its information flow contract\textsuperscript{12}. The relational Hoare logic (called SIFL\textsuperscript{13}) for reasoning about conditional information flow is formalized in Coq and have been proved sound with respect to the operational semantics of programming language. In addition, SIFL is extended to emit formal certificate of correctness claims about information flow contracts that are checkable in Coq proof assistant.

Deng Xianghua and others have manually formalized the operational semantics of Kiasan, a framework for reasoning and checking program properties based on symbolic execution, and proved the relative soundness and completeness for the basic symbolic executions by hand\textsuperscript{14,15}. Besides, they have also manually formalized the test input generation algorithms of KUnit\textsuperscript{16}, an analysis feedback extension to Kiasan, and proved the path coverage of KUnit algorithms.

To assure the assertions claimed at the source-language program to be held in the machine-language program, a verified software toolchain VST\textsuperscript{17} has been developed in Andrew Appel’s research group. The software toolchain is a set of verification and translation tools for C programs built based on the framework of CompCert, which consists of a formally verified static analyzer to check assertions about programs, a formally verified compiler to
translate to machine language, and a runtime system to support context for programs.

All of the above work are carried out as an academic research and none of these prototype tools are actually used by the industry for the program development. A still challenging open problem is how to integrate these certification techniques into commercially available static analysis tools and enable them have the ability for industrial scale development and verification.

1.2.3 Verification of Run-Time Checks

Instead of merely detecting errors, formal methods can automatically verify the presence or absence of run-time errors. Model checking, abstract interpretation, and deductive verification techniques are the most well-known formal methods for the verification of run-time errors.

Model checking\ref{model-checking} builds a model for a system and then exhaustively check whether a given property holds in this model. Verification of run-time errors can be performed by an exhaustive analysis of the program states to check whether there exists a state leading to a violation of any of the run-time properties. But it suffers from state explosion problems. Some well-known existing model checkers includes SPIN\ref{spin}, Bandera\ref{bandera}, Java Path Finder (JPF)\ref{jpf}, Bogor\ref{bogor} and so on.

Abstract interpretation\ref{abstract-interpretation} is a technique of sound abstraction of the semantics of programs, which is mainly used for formal static analysis of program behaviors. It will never yield false negatives, but may produce false alarms (or false positives). Astree\ref{astree} is one of static analysis tools designed based on the theory of abstract interpretation, aiming at formally proving the absence of run-time errors. It consists of two phases, the analysis phase is to formally construct the set of execution traces of the program, and the verification phase is to check that none of the program’s execution traces can reach a state triggering a run-time error.

Deductive program verification\ref{deductive-program-verification} consists of generating a set of mathematical proof obli-
gations (also called verification conditions), and then discharging these obligations using either automatic or interactive theorem provers. GNATProve for SPARK programs is one of such tools based on deductive verification technique. For each possible run-time error, a verification condition will be generated and then discharged by either automatic theorem prover, e.g. Alt-Ergo\textsuperscript{26}, Z3\textsuperscript{27}, CVC4\textsuperscript{28} and others, or interactive theorem prover, e.g. Coq.

All of the above methods and tools are assuming the existence of certain frontend tool that can tell them what kinds of run-time checks and where are they needed to be checked under what conditions. This dependence on the frontend and the assumption of its correctness may break the verification of run-time errors. As any errors in the frontend, either misguiding the place of run-time checks or missing some checks, will make a sound verification into an unsound one. So a sound method is to verify the correctness of the frontend as well.

### 1.2.4 Declassification Policy

In contrast to the noninterference mechanism, which is too restrictive to describe the information security requirements of real applications, declassification mechanism provides a more flexible way for specifying, reasoning about, and enforcing information security for practically useful information flow control systems. It allows secret information to be declassified to some extent by weakening noninterference mechanism, but, at the same time, it also ensures that the system cannot leak more secret information than intended. There has been a lot of work on language-based security policies for enforcing declassification mechanism, which are usually classified into four different categories by the following four dimensions: what can be declassified, where declassification can occur, who is able to declassify information and when its allowed to be declassified. In this section, we don’t plan to discuss them in four different dimensions as the declassification policies usually touch multiple dimensions and it’s hard to categorize them into exact one dimension.

*Flow locks*\textsuperscript{29} introduce a very simple mechanism for specifying conditional declassifica-
tion policies, which can specify conditions under which data may be read by certain system principals (or security levels). In this approach, each variable is associated with a set of system principals that can read it, and each principal can be guarded with conditions under which the principal has the right to access to the variable’s value. The conditions for declassification are represented as locks, which can be manipulated by means of special program instructions to either open or close locks according to the changes of program state.

Nondisclosure policy\(^{30}\) is a block-structured approach to dynamically manipulate flow policies. It is a generalisation of non-interference that supports locally induced flow policies through the use of the special construct “flow \(F\) in \(M\)”, where \(F\) is a flow policy and \(M\) is an expression or subprogram, meaning that \(M\) is executed in the context of current flow policy extended with \(F\), and the current policy is restored when the execution of \(M\) terminates. The flow policy \(F\) is in the form of \(\ell_1 \prec \ell_2\), meaning declassification from security level \(\ell_1\) to \(\ell_2\), for example, for the security levels \(H\) and \(L\), with \(H > L\), “flow \((H \prec L)\) in \((y_l = x_h + 2)\)” is legal because the flow policy \(H \prec L\) allows the declassification from \(H\) to \(L\) in this assignment.

Delimited nondisclosure policy (DND)\(^{31}\) uses commands of the form “\(\text{declassify}(\text{exp})\) in \{c\}” to specify what (through the value of expression \(\text{exp}\)) may be declassified in the program \(c\). Security domains are not explicitly mentioned in declassification commands because implicitly a flow policy with only two domains, public and secret, is assumed.

Delimited release\(^{32}\) uses so called escape hatches to indicate what may be declassified by a program. An escape hatch has the syntax \(\text{declassify}(\text{exp}, \ d)\), where \(\text{exp}\) is an expression and \(d\) is a security domain in the given flow policy. Semantically, the escape hatch specifies that the value of \(\text{exp}\) in the initial state (i.e. before program execution begins) may be revealed to the security domain \(d\).

Localized delimited release\(^{33}\) strengthens the demands of the delimited release by location sensitivity. It defines the security based on low-bisimulation relations for each pair of states in two execution traces. It collects the declassified expressions along the trace, and at
each declassification point, check whether current two states are low equivalent when they agree on the values of collected declassified expression up to this point.

*Conditioned gradual release*\(^ {34} \) use revealed knowledge to bound the observed knowledge by the attacker. It defines security as what a low observer knows about the initial state by observing the visible part of the trace should be bounded by the declassification policy. A declassification policy \((P\&\psi \ mod \ x)\) is checked at each declassification point to see whether condition \(P\) is always true in current state, and what the attacker observes from current declassification operation is bound by what are allowed to be leaked by observing the current value of \(\psi\).

*Declassification with explicit reference points*\(^ {35} \) uses declassification guard \(dguard(r, \exp, d)\) to specify the values that may be declassified by an expression \(\exp\) and by a reference point \(r\). The reference point determines a set of states with the intention that the value of \(\exp\) in any of these states may be declassified to domain \(d\). Their framework allows one to make explicit in which states \(\exp\) is evaluated to be declassified.

More others’ work on declassification policy are still going on, but few of them have ever tried to integrate their proposed policy framework into some real programming languages and be used for the checking of the information security for the real programs.

### 1.3 Contributions

This dissertation presents and explores formal methods to ensure the software correctness properties from different aspects, and makes the following contributions towards the development of highly reliable and robust softwares used in critical domains to increase our confidence in their correctness.

**Formalization of the Language Semantics for SPARK** SPARK is a subset of Ada programming language designed for development of safety- and security-critical applications. Our first contribution is the formalization work towards a formal semantics for SPARK 2014,
the latest version of SPARK with new executable specification features. In this work, we have formalized the dynamic semantics for a core subset of the SPARK 2014 language in the Coq proof assistant (Section 3.3). The core language subset includes scalar subtypes and derived types, array types, record types, procedure calls, and locally defined subprograms; a large class of programs can be desugared to this core subset, thus, enabling evaluations on realistic SPARK systems to some extent. The formal semantics specification represents our trust-base (along with Coq, which itself has been highly-regarded as a proof system that has a smaller trust-base compared to others); the specification is trustable because, for example, it has been manually inspected by leading experts in SPARK/Ada both in industry and academia. Hence, it can be considered as the reference SPARK 2014 formal semantics. The main novelty in our semantics formalization is the modeling of on-the-fly run-time checks within the language semantics, which lays the foundation for our (future) work on mechanical reasoning about the correctness of SPARK translation and analysis.

**Formalization, Proof and Certification for Run-time Check Generation and Optimization** In SPARK, the compiler (called GNAT compiler) and verification tool set (GNATProve) are integrated seamlessly and GNAT compiler plays an important role in the GNATProve verification architecture. For example, to prove the absence of run-time errors, GNATProve relies on the correctness of run-time checks placement by the GNAT compiler frontend. Our second contribution is to build a mechanized proof infrastructure to formally certify the run-time checks within program AST produced by GNAT compiler frontend, which includes:

- An implementation of a certified run-time check generator for the core language (Section 4.2); that is, the implementation is proved to be consistent with the reference semantics with respect to the class of errors that can arise in the language subset (such as overflow checks, range checks, array index checks and division by zero checks). The consistency guarantees that if language-defined run-time checks generated by the
certified implementation do not fail, a program cannot “go wrong” according to the SPARK formal semantics. The generated checks by the implementation represents the baseline as the most conservative run-time check set (i.e., a larger set is unnecessary and could even be problematic).

- An implementation of a certified run-time check optimizer (Section 4.3). The optimization is needed because the GNAT frontend employs various optimizations to reduce the set of run-time checks that it generates for run-time efficiency sake. The certified optimizer uses an abstract interpretation-based interval analysis; it generates a smaller set of run-time checks compared to the ones produced by the GNAT frontend, while still being consistent.

- An implementation of a conformance checker as a back-end of the GNAT front-end (including, e.g., a SPARK program translator to fully resolved SPARK ASTs in Coq) that automates evaluations of the GNAT frontend against the certified run-time check generators (Section 4.4). This essentially turns the industrial GNAT frontend into a certifying\footnote{Certifying here means that the tool generates evidence testifying that it is in fact consistent with its specification for a particular use of the tool.} tool with respect to introduction of run-time error check decorations. This increases the confidence in the GNAT compiler back-end that embeds run-time assertion checking when it emits machine code for testing, as well as in the GNAT-prove verifier that uses the run-time check decorations to determine what verification conditions to generate.

A Sound Information Security Policy Framework The third contribution of this dissertation is to propose a sound information security policy framework to enforce the safety of information flow within a system that manipulates data of different security levels, including:

- A design of a language-based information security policy specification framework (Sec-
tion 5.1) to capture the desired information flow from one domain to another on the event of certain actions, and show how the framework can be specified and integrated into SPARK using Ada 2012 aspects decorations. It enhances SPARK current information flow analysis with ability to declare domains and specify declassification policy between those domains.

- A design of a static enforcement algorithm (Section 5.2) that can automatically check whether the program conforms to the specified information security policy, it’s a type checking system that based on type constraint generation followed by constraint solving procedure. The checking algorithm is compositional, and the verification of information security can be achieved modularly, at the level of individual subprograms, such that a called procedure can be checked with its inferred type constraint specification.

- Formalization of the operational semantics for programs with declassification procedures and proof of the correctness of the policy enforcement algorithm with respect to the formalized program semantics (Section 5.4).

**Toolset and Evaluation** We have provided a software certification and verification infrastructure for static analysis of SPARK programs. This infrastructure includes a toolset for program translation and analysis, such as translator (Jago) of SPARK 2014 ASTs into Coq ASTs, conformance checker for run-time check decorations of ASTs and the automatic checker for information security policies. Various evaluation of our proposed formal methods have been carried out on our toolset to demonstrate their effectiveness to assure the software certification process to build highly reliable and robust softwares.

**1.4 Outline**

The rest of this dissertation is organized as follows.
Chapter 2 is a background introduction, which contains a brief tutorial on SPARK 2014, a simple introduction about program run-time errors, a simple introduction to the information flow control techniques to ensure secure information flow of a system, and a summary about the formalization and proof techniques with Coq.

Chapter 3 presents the formal semantics of a core subset of SPARK 2014 programming language, including the run-time checks that are enforced by the SPARK language.

Chapter 4 provides the correctness proof for both the run-time check generation and run-time check optimizations, and their application to certify the correctness of run-time check decorations produced by GNAT compiler frontend.

Chapter 5 presents the proposed information security policy to be integrated into SPARK 2014 and an efficient and sound static analysis algorithm for automatic and modular enforcement of the proposed security policy.

Chapter 6 summarizes and discusses about future work.
Chapter 2

Background

2.1 SPARK 2014 Language

SPARK 2014 is a programming language designed for development of high integrity softwares with a set of associated verification tool set. It is a subset of Ada 2012 by removing some Ada features that defy analysis and proof and at the same time adding some new Ada aspects to support modular, constructive and formal verification. SPARK advocates the “correctness by construction” approach, which encourages the development of programs that are guaranteed to be correct by virtue of the techniques used in its construction. By now, SPARK has been used by various organizations, including Rockwell Collins and the US National Security Agency (NSA), to develop safety critical and security critical systems used in the area of finance, military and aviation, such as Lockheed C130J and EuroFighter projects.

Some promising features of the SPARK 2014 language:

- It’s a language with unambiguous semantics, which is required to achieve sound analysis. This also makes it possible for us to develop formal semantics for SPARK and do formal proof and verification for SPARK techniques.
• SPARK 2014 introduced executable contract based on Ada 2012, which enables SPARK 2014 to perform both contract-based static deductive verification and contract-based, dynamic verification of interesting properties of a program. In SPARK 2014, contract specification language is an essential part of SPARK programming language rather than a different annotation language separate from SPARK programming language. This makes it natural to have program development and contract specification in the same programming language, reflecting the philosophy of “correctness by construction”.

• SPARK 2014 supplies various aspects (e.g., Global, Depends, Pre and Post), attributes (e.g., 'Old, 'Update) and pragmas (e.g., Assert, Assume) to specify contracts of programs.

• The associated verification tool chain for SPARK support for a mixture of proof and other verification methods such as testing. This enables a range of static analysis, including information-flow analysis (e.g., dependence between input and output variables), formal verification of robustness properties (e.g., overflow check), formal verification of functional properties (e.g., Pre and Post conditions) and others. That’s one of major reasons why SPARK is suitable for development of high integrity (either safety critical or security critical) softwares.

Syntax of Aspects  (introduced in SPARK 2014)

```
aspect_specification ::= with aspect_mark [⇒ aspect_definition ]   
                           {, aspect_mark [⇒ aspect_definition ] } 
```

The following are some simple SPARK examples to demonstrate how the SPARK 2014 language looks like:

Example 2.1.1  (Global Aspect)
```
procedure Increase_By_One
    with Global ⇒ (InOut ⇒ X);
```
is
begin
X := X + 1;
end Increase_By_One;

The *Global* aspect specifies all the global variables that are accessible by the specified procedures to read and/or write.

**Example 2.1.2** (Depends Aspect)

```plaintext
procedure Swap (X, Y : in out Integer)
  with Depends ⇒ (X ⇒ Y,
                  Y ⇒ X);
  is
    T : Integer;
  begin
    T := X;
    X := Y;
    Y := T;
  end Swap;
```

The *Depends* aspect specifies a dependency relation between inputs and outputs of the program, and the SPARK associated verification tool sets will automatically check whether all information flow within the program implementation complies with the user specified dependency relations.

**Example 2.1.3** (Pre and Post Conditions Aspects)

```plaintext
function Divide (X, Y : Integer) return Integer
  with Pre ⇒ Y /= 0 and X > Integer ’First,
       Post ⇒ Divide ’Result = X / Y;
  is
begin
  return (X / Y);
end Divide;
```

The *Pre* aspect specifies the preconditions that should be always satisfied when the subprogram (procedure or function) is called, and the *Post* aspect specifies the postconditions
that are promised to hold by the subprogram on its call return. In the above example, the preconditions ensure that the subprogram will never get into division error guarded by condition \( Y \neq 0 \), divisor \( y \) being not equal to zero, and overflow error guarded by condition \( X > \text{Integer}'\text{First} \) as \((\text{Integer}'\text{First} / (-1))\) will cause overflow. The postcondition specifies that the subprogram \textit{Divide} will always return result \( X / Y \) if \( X \) and \( Y \) satisfies its preconditions.

Example 2.1.4 (Contract Cases Aspect)

```haskell
function Max (X, Y : Integer) return Integer
    with Contract_Cases ⇒ ( X >= Y ⇒ Max' Result = X,
                            X < Y ⇒ Max' Result = Y);

is
    Z : Integer := X;
begin
    if X < Y then
        Z := Y;
    end if;
    return Z;
end Max;
```

The \textit{Contract_Cases} aspect provides a structured way of defining a subprogram contract using mutually exclusive subcontract cases. In other words, the subprogram satisfies different contract case under different conditions and each time only one of the conditions of the contract case list is satisfied. In the above example, the \textit{Contract_Cases} specifies that if \( X \) is greater than or equal to \( Y \) in the initial state of the program, then subprogram \textit{Max} should return the value of \( X \), otherwise, it returns the value of \( Y \).

A \textit{Contract_Cases} aspect can be used together with the \textit{Pre} and \textit{Post} aspects for the same subprogram. When the subprogram is called, it must be checked that the precondition specified by the \textit{Pre} aspect is satisfied and only one of the conditions of the contract case list holds. When the subprogram returns, the postcondition specified by the \textit{Post} aspect and the contract case under true condition should be satisfied. \textit{Contract_Cases} is a little like the refinement of the \textit{Post} aspect but not exactly.
In the following example, we will show an example specified by various aspects.

**Example 2.1.5** (Combination of Various Aspect)

```plasmalike
procedure Abs (X : in Integer; Y : out Integer)
  with Global ⇒ null,
    Depends ⇒ (Y ⇒ X),
    Pre ⇒ (X > Integer 'First),
    Post ⇒ (Y >= 0),
    Contract_Cases ⇒ (X >= 0 ⇒ Y = X,
                      X < 0 ⇒ Y = -X);
begin
  if X >= 0 then
    Y := X;
  else
    Y := -X;
  end if;
end Abs;
```

In the above example, subprogram Abs computes the absolute value of X and stores the result in the output variable Y. The precondition \((X > \text{Integer’First})\) specified in Pre aspect is to guard against the possible overflow.

### 2.2 Run-Time Errors

Run-time error is an error occurs during the execution of a program that prevents a program from working correctly. Unlike stop errors, which typically cause program to stop working, a program can continue to work after a run-time error, but in an unspecified way and the execution result is unexpected. Therefore, many programming languages provide some run-time check mechanism to raise run-time error exception as the executing program gets into a run-time error state. This is sufficient in some applications, but in the domain of safety- and security- critical applications, any of such run-time exception is undesirable and may cause disaster, such as applications in medical device systems. It can be achieved by testing of all
different usage scenarios before the deployment of the applications. The method of testing can detect some run-time errors but it cannot guarantee the absence of all run-time errors, while formal verification methods can formally verify a program to be free of run-time errors through static analysis of all possible program states.

In Ada 2012, there are four predefined exceptions in the language:

- `CONSTRAINT_ERROR`: raised whenever a subtype’s constraint is not satisfied or whenever a numeric operation cannot deliver a correct result (e.g. for arithmetic overflow, division by zero).

- `PROGRAM_ERROR`, raised upon an attempt to call a subprogram or activate a task when the body of the corresponding unit has not yet been elaborated.

- `STORAGE_ERROR`: raised whenever space is exhausted.

- `TASKING_ERROR`: raised when exceptions arise during intertask communication.

SPARK is a subset of Ada and many of run-time errors in Ada are excluded in SPARK because of the constraints of language features in SPARK, e.g. no dynamic allocation in SPARK, therefore no `STORAGE_ERROR` exception. Thus the remaining main run-time errors in SPARK are: incorrect indexing of array, writing to a variable with a value out of the range of the variable’s type through either assignment or pass-by-value in procedure call, overflow of an arithmetic operation with regard to the range of the target base type, and division by zero. All of these interesting errors can be divided into three major categories: range error, overflow error and division error. The following are some classical examples to illustrate these kinds of run-time errors in SPARK.

**Range Error**

**Example 2.2.1** (Incorrect Indexing of Array)

```ada
subtype IndexT is Integer range 0 .. 10;
type ArrayT is array (IndexT) of Integer;
procedure Update(A: in out ArrayT; I: in IndexT; V: in Integer) is
```
The `ArrayT` is defined as an array of integer type with `IndexT` as its indexing bound type. It will raise range error in `A(I+1)` when `I` is 10, as the result of `I+1` falls out of the range of `IndexT` and will cause illegal array access.

**Example 2.2.2 (Assignment Range Error)**

```pascal
subtype T is Integer range 0 .. 10;
procedure Assign(X: in T; Y: in Integer; Z: out T) is
begin
  ...  
  Z := X + 1;  -- possible range error;
  Z := Y;  -- possible range error;
  ...
end Assign;
```

Type `T` is defined as a subtype of integer with range 0 to 10. For a variable `Z` of type `T`, the assignment with `X+1` will cause range error when the value of `X` is 10. Similarly, the range error is raised for assignment from `Y` to `Z` whenever the value of `Y` goes beyond the range between 0 and 10.

**Example 2.2.3 (Procedure Call Range Error)**

```pascal
subtype T is Integer range 0 .. 10;
procedure f(X: in T; Y: out Integer) is
begin
  Y := X + 1;
end f;
procedure g(U: in Integer; V: out T) is
begin
  ...  
  f(U, V);  -- possible range error;
  ...
end g;
```
Procedure \( f \) increases the value of \( x \) by one and assigns the result to \( y \), with \( x \) being a variable of subtype \( T \) and \( y \) being an integer, and there is no run-time error in this procedure. But, when it’s called in procedure \( g \) with arguments \( u \) and \( v \), it may causes range errors in two possible places: the range error raised when passing in the value from \( u \) to \( x \) and the range error when passing out the value from \( y \) to \( v \) when it returns. In both cases, they try to write to a variable with a value that maybe out of its range.

**Overflow Error**

**Example 2.2.4 (Arithmetic Operation Overflow)**

```plaintext
procedure f(X: in Integer; Y: in out Integer) is
begin
  . . .
  Y := X + Y;  -- possible overflow error;
  . . .
  Y := X / Y;  -- possible overflow error;
  . . .
end f;
```

Overflow error happens when the resulting value of an arithmetic operation, including \(+\), \(-\), \(*\) and \(/\), gets out of the range of the base type integer. For example, for \( X/Y \), the overflow happens when \( X \) is the minimum value of integer and \( Y \) is -1.

**Division Error**

**Example 2.2.5 (Division by Zero)**

```plaintext
procedure f(X: in Integer; Y: in out Integer) is
begin
  . . .
  Y := X / Y;  -- possible division error;
  . . .
  Y := X mod Y;  -- possible division error;
  . . .
end f;
```

The division by zero error happens whenever the divisor is zero for division operators, such as \(/\) and \( \text{mod} \).
2.3 Information Security

There exist a wide range of safety- and security- critical information systems being used in the area of finance, healthcare, military and aviation to process multi-level security data, including password checking system, online shopping and banking system. In general, there are three ways to protect information confidentiality and integrity. Cryptography provides a way to hide information in data and protect data from tampering, but it’s costly and impractical to decrypt and encrypt the data each time we need to process the data. Access controls can protect data from being read or modified by unauthorized users. However, it cannot prevent the propagation of information after it has been released for processing by a program. Information flow is the transfer of information from an input to an output in a given process. It reflects an end-to-end behavior of a system and Information flow control provides a complementary approach to track and regulate the information flow of a system to prevent secret data from leaking into public. One promising way to ensure the secure information flow of a system is to use noninterference, which requires that secret data may not interfere with (or affect) public data, as an end-to-end semantic security condition to reason about information flow security.

However, the security requirements enforced by noninterference are too restrictive. In fact, computing systems often need to deliberately declassify (or release) parts of its confidential information, for example, in password checking system, it’s necessary to reveal some information about the stored password, telling the user whether his input password is correct or not. Declassification of information occurs when the confidentiality of information is decreased or become less restrictive. It’s an exception to the normal secure information flow. The major challenge is to design an elegant declassification mechanism to precisely capture the intentional information release and ensure that the release is safe: the attacker could neither get around the declassification mechanism nor exploit the declassification mechanism to reveal more secret information than intended, e.g. secret laundering. A lot of research has been carried out in the area of declassification, and Sabelfeld and Sands provides a road
map of the main declassification methods in current language-based security research and classify them according to four dimensions: what can be declassified, where declassification can occur, who is able to declassify information and when it's allowed to be declassified.

2.4 Formalization and Proof in Coq

Coq is a proof assistant, which is based on the Curry-Howard correspondence (propositions-as-types, formulas-as-types or proofs-as-programs), for interactive theorem proving. It provides a specification language called Gallina, which integrates programming and proving within a single formal language. That’s, Gallina can be used as either a higher-ordering functional programming language to develop normal functional programs or a logic system to build and prove mathematical theories.

The logical foundation of Coq is the Calculus of Inductive Constructions (CIC), which has been proved to have property of strong normalization. This is a crucial property to ensure termination of all programs (or proofs) written in Gallina, as non-terminating programs introduce logical inconsistency, where any theorem can be proved with an infinite loop. This is also one way to avoid the halting problem, which is common in other programming languages.

Coq comes with a set of built-in automation tactics, such as intros, inversion, simpl, auto and so on, to address a goal directly or be applied repeatedly to reduce a goal to its subgoals until they are completely proved. While it is possible to conduct proofs using only those tactics, you can significantly increase your productivity by working with a set of more powerful user-defined tactics. To do it, Coq supplies a tactic language called Ltac, which enables users to develop powerful problem-specific tactics for domain-specific applications.

In contrast to the classical program verification method, where programs and specifications for program correctness properties are developed separately and finally a proof is built for the programs with respect to their specifications, dependent types in Coq make it
possible to integrate programming, specification, and proof into a single phase. It supports
to encode correctness properties and proof within types, for example, the array type with
specified array size can guarantee the absence of out-of-bounds errors. Adam Chlipala from
MIT wrote a book about how to develop certified programs with dependent types in Coq.
Both the classical program verification method and dependent types mechanism have their
own advantages, and it’s hard to say which one is the best choice. It totally depends on
the favor of the user which verification style he prefers, as both can achieve the same goal.
In general, dependent types work great and save effort for verification of programs manip-
ulating data structures with some invariant properties. It can automatically propagate the
invariant properties attached to data structures along the verification procedure without
assuming and proving these invariants all the way down. And in most other cases, the
classical program verification method is more convenient.

Certified Program vs Certifying Program: A certified program is the one coming with
formal mathematical artifacts (such as machine-checked proofs) that serve as evidence that
its implementation is consistent with its specification, while a certifying program is the
one to generate evidence for another program to testify that it’s in fact consistent with its
specification for a particular use of the program.

Some of the most famous certified programs include CompCert and VST as mentioned
before. CompCert is a certified compiler that the compiler itself is proved to be correct in
Coq. VST is a certified software verification toolchain built based on top of CompCert that
targets to prove the correctness of the whole software toolchain in Coq from source code
analyzer to program translator and the program supporting context. Furthermore, as part
of the VST project, researchers even developed a certified theorem prover, called VeriStar,
for a decidable subset of separation logic. All these certified programs can guarantee the
correctness of the program verified at the source code level can still hold at the machine
code level.

Usually, it’s much easier to check the output of the program than verifying the correct-
ness of the program itself. Certifying program is built based on this idea such that the
program itself is not necessarily to be correct but it can generates a certified output, such
as certifying compiler\textsuperscript{13}. It constructs proof for the compiled program, with the proof to be
easily checked by the program consumer, thus the output of the certifying compiler is also
called proof-carrying code\textsuperscript{14}. Proof-carrying code establishes the trust between the program
consumer and its producer, which is important to convince the consumer that the foreign
program is safe to execute, particularly in distributed and web computing where mobile
code is allowed. A lot of work on certifying compiler focus on type-safety properties that
it constructs proofs of type safety for programs with respect to some typing rules. How-
ever, it’s difficult to automatically build proof for more advanced correctness and security
properties for a program with certifying compiler.
Chapter 3

Formal Semantics of SPARK 2014

One main contribution of this thesis is a formal language reference semantics of core SPARK 2014 defined using the Coq proof assistant\(^4\); Coq allows for specifying, implementing, and proving programming language related properties. We chose Coq due to the fact that it has a relatively small core which has been vetted by many experts in the programming language community (small trust-base).

The core language includes features typically found in imperative languages such as arrays, records, and procedure calls, as well as SPARK-specific structures, such as nested procedures and subtypes. In Ada programming language, it has introduced four categories of predefined exceptions that may be raised during the program execution. As a subset of Ada, the exceptions that may occur in SPARK are referred as run-time errors. One major difference between SPARK and other programming languages (e.g., C) is that verification for absence of run-time errors is required by the semantics of the language itself. Thus, our reference semantics perform the required run-time checks; that is, run-time checks are enforced at appropriate points during the program execution, and the program will terminate with a run-time error message as soon as one of its run-time checks fails.

The formalization includes: (a) a SPARK AST representation (symbols and types are fully resolved), and (b) SPARK evaluation semantics (including, e.g., state/value represent-
3.1 Syntax of Core SPARK 2014

Figures 3.1, 3.2 and 3.3 show the syntax of core SPARK that we have formalized in Coq. The syntax is given according to both SPARK 2014 and Ada 2012 reference manuals, and we have ignored some trivial details, e.g., numeric_literals denoting the integer literals.

In Figure 3.1, a SPARK program compilation is constructed from a set of compilation units compilation_unit that can be separated compiled. A library_item is a compilation unit that’s the body of a library unit library_unit_body. A subprogram_body specifies the execution of a subprogram and it’s an implementation of the library_unit_body.

Subprogram is a procedure that can have side effects to externally visible variables through either output parameters or global variables. SPARK supports three access modes: in, out and in out. A parameter with in mode is readable only, out means writable only, and in out means both readable and writable. By default, the mode of parameter is in. Within the declarative part of the procedure body, the user can declare new objects (e.g., integer variable) through object_declaration, new types (e.g., subtype, array and record type) through subtype_declaration and type_declaration, or even nested procedures. A derived type defined with derived_type_definition is a new type created from an existing one (e.g., type T is new Integer range 1 .. 5). A procedure implementation body contains a sequence of statements handled_sequence_of_statements, which can be either simple statement simple_statement (e.g., assignment assignment_statement) or compound statement compound_statement (e.g., if statement if_statement and loop statement loop_statement). An expression expression can be integer/boolean constant (numeric_literal), variable, or nested array or record (name).

For example, the following procedure Factorial defines a subprogram_body, which can work as a compilation unit. The procedure name Factorial corresponds to the defining_program_unit_name in the syntax, and the parameters M and N are its parameter_profile. The local declarations
compilation ::= \{compilation_unit\}

compilation_unit ::= library_item

library_item ::= library_unit_body

library_unit_body ::= subprogram_body

subprogram_body ::= subprogram_specification is
  declarative_part
  begin
    handled_sequence_of_statements
  end;

subprogram_specification ::= procedure_specification

procedure_specification ::= procedure defining_program_unit_name parameter_profile

defining_program_unit_name ::= defining_identifier

parameter_profile ::= [formal_part]

formal_part ::= (parameter_specification {; parameter_specification})

parameter_specification ::= defining_identifier_list : mode subtype_mark

mode ::= [in] | in out | out

defining_identifier_list ::= defining_identifier {, defining_identifier}

defining_identifier ::= identifier

subtype_mark ::= subtype_name

Figure 3.1: Core SPARK Syntax (Part 1)
declarative_part ::= \{declarative_item\}
declarative_item ::= basic_declarative_item | subprogram_body
basic_declarative_item ::= basic_declaration

basic_declaration ::= type_declaration | subtype_declaration | object_declaration
type_declaration ::= full_type_declaration

subtype_declaration ::= subtype defining_identifier is subtype_indication

full_type_declaration ::= type defining_identifier is type_definition
type_definition ::= integer_type_definition | array_type_definition | record_type_definition | derived_type_definition

handled_sequence_of_statements ::= sequence_of_statements
sequence_of_statements ::= statement \{statement\}
statement ::= simple_statement | compound_statement

simple_statement ::= null_statement | assignment_statement | procedure_call_statement

compound_statement ::= if_statement | loop_statement

Figure 3.2: Core SPARK Syntax (Part 2)
assignment_statement ::= name := expression;

procedure_call_statement ::= procedure_name actual_parameter_part;

if_statement ::= if condition then sequence_of_statements
[else sequence_of_statements] end if;

loop_statement ::= while condition loop sequence_of_statements end loop;

name ::= identifier
| indexed_component
| selected_component

equation ::= relation {and relation} | relation {or relation}

indexed_component ::= prefix(expression {, expression})

selected_component ::= prefix . identifier

prefix ::= name

relation ::= simple_expression [relational_operator simple_expression]

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary | not primary

primary ::= numeric_literal | name | (expression)

Figure 3.3: Core SPARK Syntax (Part 3)
of Result and \( T \) are the declarative part of the procedure and the implementation body of the procedure specified in between begin and end.

```java
procedure Factorial (N : in Integer; M : out Integer)
is
    Result : Integer := 1;
    T: Integer;
begin
    T := N;
    while T > 0 loop
        Result := Result * T;
        T := T - 1;
    end loop;
    M := Result;
end Factorial;
```

This is a core subset of SPARK 2014 that the other SPARK language features can also be translated and represented with this core subset. Formalization of the language semantics is discussed in detail in the following sections.

### 3.2 Run-Time Checks

SPARK defines a set of run-time checks to be inserted and verified during the program certification process and guarantee that for all possible execution of SPARK program, it’s impossible for the checks to be violated. For the core SPARK 2014, we mainly focus on the following widely used and highly important run-time checks as required to be enforced by SPARK reference manual.

- Overflow Check: this is a check for an operator where its operation may cause overflow or underflow, such as binary operators (+, -, *, /) and unary operator (-).

- Division Check: this is check for division operators, such as (/, mod), to indicate a zero divide check.
• Range Check: this is a check for an expression whose computed value must fall within the target type range of the context, such as the right hand side expression of an assignment with its left hand side being some variable of integer subtype, subscript expression in an indexed component, argument expressions for a procedure call and initialization value expression for an object declaration where the types of their target variables are some range constrained ones.

3.3 Formal Language Semantics

3.3.1 SPARK AST Representation

SPARK ASTs are represented using inductive type definitions in Coq, as shown in Figures 3.4 and 3.5, where each constructor of the type definitions is annotated with a unique AST number. The AST numbers are useful as keys for source location, symbol, and type tables. For example, the following inductively defines an expression:

```coq
Inductive expr : Type := BinOp : astnum → binop → expr → expr → expr
```

where BinOp is the constructor for binary expression from two sub-expressions and one binary operator, labeled with a unique AST number. As an example, the following shows the Coq AST for the binary expression Result * T, where 25 is its astnum and 26 and 28 are the astnum for both sub-expressions Result and T and Multiply is a binop to denote the operator *. The BinOp is a Coq AST constructor to bind all these elements together to specify the multiplication expression and comments in Coq are enclosed between (* and *).

```coq
(BinOp 25 Multiply
    (Name 26 (Identifier 27 (Result Multiplication 5) ))
    (Name 28 (Identifier 29 (T Multiplication 6) ))
)
```

For any SPARK program, all of its used names can be represented as natural numbers in Coq, including variable name, type name, procedure name and others, as long as the number is a unique representation of that name, e.g., 5 representing Result in the above example.
Definition astnum := nat.
Definition idnum := nat.
Definition procnum := nat.
Definition typenum := nat.

Inductive literal: Type :=
| Integer_Literal: Z -> literal
| Boolean_Literal: bool -> literal.

Inductive exp: Type :=
| Literal: astnum -> literal -> exp
| Name: astnum -> name -> exp
| BinOp: astnum -> binop -> exp -> exp -> exp
| UnOp: astnum -> unop -> exp -> exp

with name: Type :=
| Identifier: astnum -> idnum -> name
| IndexedComponent: astnum -> name -> exp -> name
| SelectedComponent: astnum -> name -> idnum -> name.

Inductive stmt: Type :=
| Null: stmt
| Assign: astnum -> name -> exp -> stmt
| If: astnum -> exp -> stmt -> stmt
| While: astnum -> exp -> stmt -> stmt
| Call: astnum -> astnum -> procnum -> list exp -> stmt
| Seq: astnum -> stmt -> stmt -> stmt.

Inductive type: Type :=
| Boolean
| Integer
| Subtype (t: typenum)
| Derived_Type (t: typenum)
| Integer_Type (t: typenum)
| Array_Type (t: typenum)
| Record_Type (t: typenum).

Figure 3.4: SPARK AST in Coq (Part 1)
**Inductive** range: Type := Range (l: Z) (u: Z).

**Inductive** typeDecl: Type :=
  | SubtypeDecl:
    astnum -> typenum -> type -> range -> typeDecl
  | DerivedTypeDecl:
    astnum -> typenum -> type -> range -> typeDecl
  | IntegerTypeDecl:
    astnum -> typenum -> range -> typeDecl
  | ArrayTypeDecl:
    astnum -> typenum -> type -> type -> typeDecl
  | RecordTypeDecl:
    astnum -> typenum -> list (idnum * type) -> typeDecl.

**Record** objDecl: Type := mkobjDecl{
  declaration_astnum: astnum;
  object_name: idnum;
  object_nominal_subtype: type;
  initialization_expression: option (exp) }.

**Record** paramSpec: Type := mkparamSpec{
  parameter_astnum: astnum;
  parameter_name: idnum;
  parameter_subtype_mark: type;
  parameter_mode: mode }.

**Inductive** decl: Type :=
  | NullDecl: decl
  | TypeDecl: astnum -> typeDecl -> decl
  | ObjDecl: astnum -> objDecl -> decl
  | ProcBodyDecl: astnum -> procBodyDecl -> decl
  | SeqDecl: astnum -> decl -> decl -> decl

with procBodyDecl: Type :=
  mkprocBodyDecl
  (procedure_astnum: astnum)
  (procedure_name: procnum)
  (procedure_parameter_profile: list paramSpec)
  (procedure_declarative_part: decl)
  (procedure_statements: stmt).

**Figure 3.5:** SPARK AST in Coq (Part 2)
To be more precise, instead of sharing the same natural number domain for all different names, we use \( \text{idnum} \) as natural number domain for variable names, \( \text{procnum} \) for procedure names, and \( \text{typenum} \) for type names. In other words, different types of names can share the same number but they are differentiated with respect to different semantic domains. For example, given the following SPARK program:

```spake
type T is range 0 .. 10;
...
procedure Increase (X : in out T) is
begin
    X := X + 1;
end Increase;
```

its Coq AST is represented as:

```
(TypeDecl 4 (IntegerTypeDecl 5 ((T) 1) (Range 0 10)))
...
(ProcBodyDecl 1
  (mkprocBodyDecl 2
    (* Procedure Name = = =
    (Increase) 1)
    (* Formal Parameters = = =
    (X) 1 Integer In_Out : : nil)
    (* Object Declarations = = =
    ...
    (* Procedure Body = = =
    ...
  )
)
```

where the type name \( T \), procedure name \( \text{Increase} \) and parameter name \( X \) are all represented as number 1, which is allowed as they appear in different semantics domains. The 1 in constructor \( \text{IntegerTypeDecl} \) distinguishes with the 1 in constructors \( \text{mkprocBodyDecl} \) and \( \text{mkparamSpec} \).

Our formalized SPARK subset supports two kinds of literals, as defined as literal type, where \( \text{Integer_Literal} \) is the constructor for integer literal and \( \text{Boolean_Literal} \) for boolean literal.
Expressions includes literal, binary expression, unary expression, nested array and record access and so on, which is defined as exp and name types. Statement statement includes assignment, conditional statement, while loop and procedure call.

SPARK supports range constrained scalar (integer) types that are useful as array index types; that is, the range constraints are used to determine in-bounds/out-of-bounds array operations (instead of using a special .length field such as in Java). Range constrained types can be declared using either a subtype declaration (e.g., subtype T10 is Integer range 1 .. 10), a derived type definition (e.g., type U10 is new Integer range 1 .. 10), or an integer type definition (e.g., type W10 is range 1 .. 10); they semantically differ in that the last two introduce a new type, while the first one does not; the differences have to be taken into account in the formalization. This illustrates the non-trivial number of language features that one has to cover when formalizing a real programming language that can be directly leveraged for developing high-integrity industrial tools. In Coq, these range constrained types can be defined with type declaration, for example, Subtype_Declaration is the constructor for subtype declaration, where typenum is the declared subtype name, type is its parent type name, range specifies the range of the declared subtype. Array_Type_Declaration is the constructor for declaring new array type, where typenum is the declared array type name, the first type denotes array index subtype mark and the second type denotes array component type; Record_Type_Declaration is the constructor for declaring new record type, where typenum is the declared record type name and list (idnum * type) defines a list of record fields.

Besides the declaration of new types, user can define new objects, such as declaration of variables, and it even supports the declaration of nested procedures to enable new procedure to be defined and used within the scope of the enclosing procedure. Type declaration is defined for such purpose with support for declaration of various types, objects and procedures.
3.3.2 State/Value

Due to the run-time checks, evaluating either an expression or a statement may produce an error state when the run-time check fails (otherwise a value or a state is produced, respectively), which is captured using the following generic inductive type:

\[
\text{Inductive } \text{Ret} \ (A: \text{Type}): \text{Type} := \text{OK: A} \rightarrow \text{Ret} A \\
| \text{RTE: errorType } \rightarrow \text{Ret} A.
\]

Type parameter \(A\) is either the value/state type, and errorType is the run-time error state type (e.g., division by zero, overflow, out of range), which are defined as following:

\[
\text{Inductive } \text{value: Type :=} \\
| \text{Undefined} \\
| \text{Int} (n : \mathbb{Z}) \\
| \text{Bool} (n : \text{bool}) \\
| \text{ArrayV} (a : \text{list} (\text{arrindex } \times \text{value})) \\
| \text{RecordV} (r : \text{list} (\text{idnum } \times \text{value})).
\]

The value of an uninitialized variable is labeled as Undefined. ArrayV represents the array object and RecordV represents record object.

\[
\text{Definition store : Type := list (idnum } \times \text{value}).
\]

\[
\text{Definition state := list store}.
\]

The state is a stack of frames, which is represented as a list of store to capture the mapping from variable id to value.

\[
\text{Inductive } \text{errorType: Type :=} \\
| \text{Division} \_ \text{By} \_ \text{Zero} \\
| \text{Overflow} \\
| \text{Out} \_ \text{Of} \_ \text{Range}.
\]

The errorType represents the three categories of run-time errors to be modeled in the SPARK language semantics.

3.3.3 Expression Semantics

The expression operational semantics are defined by the inductive types given below. They are complicated by the fact that sub-expression evaluation can produce an error state. In
Coq, inductive types are used for mathematical constructions of a collection of objects. Here, we apply the inductive types for constructing evaluation rules for expressions (or programs), which inductively defines the computation relationships between the resulting values (or states) and the expression (or program) to be evaluated under certain states.

\textbf{Inductive} \textit{evalExp}: symTable \rightarrow state \rightarrow expr \rightarrow Ret \textit{value} \rightarrow Prop :=

\textit{EvalBinOpE1.RTE}: for all st s e1 msg n op e2,
\textit{evalExp} st s e1 (RTE msg) \rightarrow
\textit{evalExp} st s (BinOp n op e1 e2) (RTE msg)

\textit{EvalBinOpE2.RTE}: for all st s e1 v1 e2 msg n op,
\textit{evalExp} st s e1 (OK v1) \rightarrow \textit{evalExp} st s e2 (RTE msg) \rightarrow
\textit{evalExp} st s (BinOp n op e1 e2) (RTE msg)

\textit{EvalBinOp}: for all st s e1 v1 e2 v2 op n,
\textit{evalExp} st s e1 (OK v1) \rightarrow \textit{evalExp} st s e2 (OK v2) \rightarrow \textit{evalBinOp} op v1 v2 v \rightarrow
\textit{evalExp} st s (BinOp n op e1 e2) v

\ldots

\textbf{Inductive} \textit{evalBinOp}: BinOp \rightarrow value \rightarrow value \rightarrow Ret \textit{value} \rightarrow Prop :=

\textit{CheckBinops}: for all op v1 v2 v v',
\textit{op} = \text{Add} /\op = \text{Sub} /\op = \text{Mul} \rightarrow
\textit{Denotational.binOp} op v1 v2 = \text{Some} (\text{Int} v) \rightarrow \textit{overflowCheck} v v' \rightarrow
\textit{evalBinOp} op v1 v2 v'

\textit{CheckDivRTE}: for all v1 v2,
\textit{divCheck} v1 v2 (RTE \text{Division\_By\_Zero}) \rightarrow
\textit{evalBinOp} Div (\text{Int} v1) (\text{Int} v2) (RTE \text{Division\_By\_Zero})

\textit{CheckDiv}: for all v1 v2 v v',
\textit{divCheck} v1 v2 (OK (\text{Int} v)) \rightarrow \textit{overflowCheck} v v' \rightarrow
\textit{evalBinOp} Div (\text{Int} v1) (\text{Int} v2) v'

\ldots

\textit{EvalBinOpE1.RTE} specifies the evaluation of a binary expression \((e1 \text{ op } e2)\) where the evaluation of \(e1\) produces an error state (similarly, \textit{EvalBinOpE2.RTE} for when \(e2\) fails). \textit{EvalBinOp} specifies the situation where evaluations of both \(e1\) and \(e2\) produce operand values, which are then evaluated using \textit{evalBinOp}; \textit{evalBinOp} incorporates various run-time checks such as division by zero and overflow checks by using \textit{divCheck} and \textit{overflowCheck}; \textit{divCheck} produces a value if the second operand is non-zero (otherwise, it produces the error state \text{RTE Division\_By\_Zero}), and \textit{overflowCheck} produces a value if the given value fits within the (platform) integer type value
range (otherwise, it produces RTE Overflow). Run-time checks for other language features such as array indexing are specified in the same spirit as the above, but in some cases, it requires to enforce more language specific run-time checks, such as range check for array indexing. The following is the formalization of name evaluation for both accessing array component and record field.

**Inductive** evalName : symTable -> state -> name -> Ret value -> Prop :=

| EvalIndexedComponentX_RTE: forall s t s x msg ast_num e,
  evalName s t s x (RTE msg) ->
  evalName s t s (Indexed_Component ast_num x e) (RTE msg)
| EvalIndexedComponentE_RTE: forall s t s x a e msg ast_num,
  evalName s t s x (Ok (ArrayV a)) ->
  evalExp s t s e (RTE msg) ->
  evalName s t s (Indexed_Component ast_num x e) (RTE msg)
| EvalIndexedComponent_RTE: forall s t s x a e i t l u ast_num,
  evalName s t s x (Ok (ArrayV a)) ->
  evalExp s t s e (Ok (Int i)) ->
  fetch_exp_type (name_astnum x) st = Some (Array_Type t) ->
  extract_array_index_range st t (Range l u) ->
  rangeCheck i l u (RTE Out_Of_Range) ->
  evalName s t s (Indexed_Component ast_num x e) (RTE Out_Of_Range)
| EvalIndexedComponent: forall s t s x a e i t l u v ast_num,
  evalName s t s x (Ok (ArrayV a)) ->
  evalExp s t s e (Ok (Int i)) ->
  fetch_exp_type (name_astnum x) st = Some (Array_Type t) ->
  extract_array_index_range st t (Range l u) ->
  rangeCheck i l u (Ok (Int i)) ->
  array_select a i = Some v ->
  evalName s t s (Indexed_Component ast_num x e) (Ok v)
| EvalSelectedComponentX_RTE: forall s t s x msg ast_num f,
  evalName s t s x (RTE msg) ->
  evalName s t s (Selected_Component ast_num x f) (RTE msg)
| EvalSelectedComponent: forall s t s x r f v ast_num,
  evalName s t s x (Ok (RecordV r)) ->
  record_select r f = Some v ->
  evalName s t s (Selected_Component ast_num x f) (Ok v)

EvalIndexedComponentX_RTE specifies the evaluation of an array component accessing \((x(e))\) where the evaluation of the array \(x\) produces an error state, and EvalIndexedComponentE_RTE for the
case when the evaluation for indexing expression $e$ fails. In the evaluation for array indexed component, an additional range check is required to be performed according to the index type of the array, which is fetched from a preconstructed symbol table. \texttt{EvalIndexedComponent\_RTE} is the evaluation rule for the case when the value of $e$ falls out of the range of array indexing type $t$ and throws out an \texttt{Out\_Of\_Range} error, while \texttt{EvalIndexedComponent} specifies the run-time error free evaluation of array indexing. The evaluation of record field accessing ($r.f$) is similar, it returns with a run-time error whenever the evaluation of record $r$ fails, as specified with \texttt{EvalSelectedComponent\_X\_RTE}, otherwise, returns the value of the specified record field with \texttt{EvalSelectedComponent}.

### 3.3.4 Statement Semantics

For the semantics of statement, range checks are enforced during statement executions of, for example, assignments and procedure calls. We describe the intuition behind statement semantic rules using examples before giving their formal semantics in Coq.

For an assignment, range check is enforced for its right hand side expression if the left hand side expression’s type is a range constrained type. For example,

```plaintext
subtype MyInt is Integer range 1 .. 10;
X: MyInt;
...;
X := X + 1;
```

That is, $x$ is a variable of type $MyInt$, which is defined as a subtype of $Integer$ ranging from 1 to 10. The assignment increments $x$ by 1, as follows. First, $x + 1$ is evaluated; if it returns a value (instead of an error state), the value is checked against the range of $MyInt$ before updating $x$.

For a procedure call, range checks are required for both input arguments and output parameters if the types of input parameters and output arguments are range constrained types because input arguments are assigned to the procedure input parameters, and output arguments are assigned to the output parameters. For example,
procedure foo(U: In MyInt, V: Out Integer) is
begin
  V := U + 1;
end foo;
X: Integer;
Y: MyInt;
...
foo(X, Y);

The procedure foo has one input parameter U and one output parameter V. When foo(X, Y) is executed, the value of x (whose type is the general Integer type) is checked against MyInt's range first. Similarly, a range check is enforced when assigning the value of v to y once foo finishes executing.

The formal semantics for statement is defined as following:

\[
\text{Inductive evalStmt : symTable} \to \text{state} \to \text{statement} \to \text{Ret state} \to \text{Prop :=}
\]
\[
\text{| EvalAssignE\_RTE: for all st s e msg ast_num x,}
\quad \text{evalExp st s e (RTE msg)} \to
\quad \text{evalStmt st s (Assign ast_num x e) (RTE msg)}
\]
\[
\text{| EvalAssign: for all st s e v x t s1 ast_num,}
\quad \text{evalExp st s e (Ok v)} \to
\quad \text{fetch_exp_type (name_astnum x) st = Some t} \to
\quad \text{is_range_constrained_type t = false} \to
\quad \text{storeUpdate st s x v s1} \to
\quad \text{evalStmt st s (Assign ast_num x e) s1}
\]
\[
\text{| EvalAssignRange\_RTE: for all st s e v x t l u ast_num,}
\quad \text{evalExp st s e (Ok (Int v))} \to
\quad \text{fetch_exp_type (name_astnum x) st = Some t} \to
\quad \text{extract_subtype_range st t (Range l u)} \to
\quad \text{rangeCheck v l u (RTE Out_Of_Range)} \to
\quad \text{evalStmt st s (Assign ast_num x e) (RTE Out_Of_Range)}
\]
\[
\text{| EvalAssignmentRange: for all st s e v x t l u ast_num,}
\quad \text{evalExp st s e (Ok (Int v))} \to
\quad \text{fetch_exp_type (name_astnum x) st = Some t} \to
\quad \text{extract_subtype_range st t (Range l u)} \to
\quad \text{rangeCheck v l u (Ok (Int v))} \to
\quad \text{storeUpdate st s x (Int v) s1} \to
\quad \text{evalStmt st s (Assign ast_num x e) s1}
\]
...
The inductive type evalStmt specifies the evaluation semantics for statement that translates an initial state from one normal state to another normal or run-time error state. EvalAssignE_RTE is the evaluation rule for assignment (Assign ast_num x e) when the evaluation of the right hand side expression e gets into run-time error state, which is propagated as the final state of assignment evaluation. EvalAssign gives the evaluation semantics for a normal assignment to a non-range-constrained variable by first evaluating the value of e and then using it to update the state of x with storeUpdate operation, where fetch_exp_type returns the type of x and is_range_constrained_type is a function to check whether it’s a range constrained type. EvalAssignRange_RTE defines the evaluation semantics for assignment to range-constrained variable and it returns an Out_Of_Range error when the value of the right hand side expression e goes beyond the allowed value range of the left hand side name x, while EvalAssignmentRange is for the case when both the evaluation of e and the range check of its value against type of x are all Ok. storeUpdate is used to update the value of a name, which can be either a variable, array indexed component or record selected component, with its formal semantics defined as the following:

\[
\text{Inductive storeUpdate: symTable \rightarrow state \rightarrow name \rightarrow value \rightarrow Ret state \rightarrow Prop}
\]

\[
:= \mid \text{SU_Identifier: for all } s x v s1 st ast_num, \\
\quad \text{update } s x v = \text{Some } s1 \rightarrow \\
\quad \text{storeUpdate st s (Identifier ast_num x) v (Ok s1)} \\
\mid \text{SU_IndexedComponentX_RTE: for all } st s x msg ast_num e v, \\
\quad \text{evalName st s x (RTE msg)} \rightarrow \\
\quad \text{storeUpdate st s (Indexed_Component ast_num x e) v (RTE msg)} \\
\mid \text{SU_IndexedComponentE_RTE: for all } st s x a e msg ast_num v, \\
\quad \text{evalName st s x (Ok (ArrayV a))} \lor \text{evalName st s x (Ok Undefined)} \rightarrow \\
\quad \text{eval_expr st s e (RTE msg)} \rightarrow \\
\quad \text{storeUpdate st s (Indexed_Component ast_num x e) v (RTE msg)} \\
\mid \text{SU_IndexedComponent_RTE: for all } st s x a e i t l u ast_num v, \\
\quad \text{evalName st s x (Ok (ArrayV a))} \lor \text{evalName st s x (Ok Undefined)} \rightarrow \\
\quad \text{eval_expr st s e (Ok (Int i))} \rightarrow \\
\quad \text{fetch_exp_type (name astnum x) st = Some (Array_Type t)} \rightarrow \\
\quad \text{extract_array_index_range st t (Range l u)} \rightarrow \\
\quad \text{do_range_check i l u (RTE Out_Of_Range)} \rightarrow \\
\quad \text{storeUpdate st s (Indexed_Component ast_num x e) v (RTE Out_Of_Range)} \\
\]\n
43
SU_IndexedComponent: for all st s x arrObj a e i t l u v a1 s1 ast_num,
  evalName st s x (Ok arrObj) ->
  arrObj = (ArrayV a) \(/\) arrObj = Undefined ->
  eval_expr st s e (Ok (Int i)) ->
  fetch_exp_type (name astnum x) st = Some (Array_Type t) ->
  extract_array_index_range st t (Range l u) ->
  do_range_check i l u (Ok (Int i)) ->
  arrayUpdate arrObj i v = (Some (ArrayV a1)) ->
  storeUpdate st s x (ArrayV a1) s1 ->
  storeUpdate st s (Indexed_Component ast_num x e) v s1

...
3.3.6 Evaluation

The main evaluation mechanism that we used to assess the integrity of the mechanized semantics was through manual inspection by various experts including SPARK/Ada designers/developers. Moreover, we have proved that the SPARK formal semantics enjoys a form of type safety (Section 4.3), which guarantees, to some extent, its internal consistency. Furthermore, we double checked the semantics against the SPARK\textsuperscript{46} and Ada\textsuperscript{38} reference manuals, as well as against the GNAT compiler by probing it on some test programs and experiments (Section 4.4).
Chapter 4

Run-Time Check Certification

4.1 Architecture

Figure 4.1 gives an architectural overview of our approach; the subsequent sections will describe each of the components.

4.2 Certified Run-Time Check Generator

Given a SPARK program, the GNAT compiler front-end builds the program fully-resolved (symbol/type) AST decorated with flags that indicate the position and nature of the run-time checks to be performed. When down-stream tools process the ASTs, they interpret or transform the decorations. For example, a later phase of the GNAT compiler replaces each decoration with an assertion AST representing code that implements the corresponding run-time check. In contrast, the Why3\textsuperscript{47}-based GNATprove verification tool uses the decorations to generate verification conditions. Both tools assume that the run-time check decorations inserted by the GNAT compiler front-end are correct.

To formally capture the notion of decorating ASTs with run-time check information, we implemented in Coq a run-time check decoration generator (RT-GEN in Figure 4.1) whose
consistency with the mechanized SPARK reference semantics was established via a Coq proof. Hence, the correctness of RT-GEN is certified. To achieve this, a different set of operational semantic rules is needed (called EVAL-RT) – one that “evaluates” an AST with run-time check decorations and only enforces the checking semantics where a decoration occurs. Then, one can prove that, for any program and for any program initial state, EVAL-RT supplied with run-time check decorations generated by RT-GEN produces exactly the same state as EVAL (i.e., the SPARK reference semantics).

There are many ways one can engineer EVAL-RT and RT-GEN. One way is to create EVAL-RT by modifying EVAL to accept a table that maps astnum to a collection of run-time checks and then have RT-GEN generate such a table. Another way is to define a new AST type (AST-RT) that explicitly holds run-time decorations as attributes, and RT-GEN transforms EVAL’s AST to AST-RT. We developed both; perhaps surprisingly, the consistency proof is
shorter in the latter case (the former is more general/extensible, hence, more complex). Thus, for simplicity, we present the latter here.

**EVAL-RT**: is a modified EVAL that accepts AST-RT where run-time check decorations are represented as tree attributes. For example, AST-RT expression is defined as follows.

**Inductive** expRT: Type :=

| BinOpRT: astnum → binOp → expRT → expRT →
  interiorChecks → exteriorChecks → expRT |

| ... |

The difference from AST is that two additional fields interiorChecks and exteriorChecks are introduced; interiorChecks are intended for run-time checks associated with the binary operator (e.g., addition requires overflowCheck), while exteriorChecks are checks associated with expression’s context (e.g., if the expression is used for array indexing, then it should be range-checked against the array size). Once AST-RT is defined, one can then define EVAL-RT that accepts AST-RT and enforces the explicitly listed run-time checks (e.g., in interiorChecks and exteriorChecks), as illustrated below.

**Inductive** evalExpRT:

symTabRT → state → expRT → Ret value → Prop :=

(* rule: no error from e1 & e2 evaluations *)

| EvalBinOpRT: ∀ st s e1 v1 e2 v2 ins op v n exs ,
  (* evaluate args *)
  evalExpRT st s e1 (OK v1) →
  evalExpRT st s e2 (OK v2) →
  (* process RT checks *)
  evalBinOpRTS ins op v1 v2 v →
  evalExpRT st s (BinOpRT n op e1 e2 ins exs) v |

...
Inductive evalBinOpRTS : checkFlags → binOp →
    value → value → Ret value → Prop :=
    (∗ rule: done with checks, perform op ∗)
| CheckBinOpNil: ∀ op v1 v2 v,
    Denotational.binOp op v1 v2 = Some v →
    evalBinOpRTS nil op v1 v2 (OK v)

    (∗ rule: performing next check returns error ∗)
| ChecksBinOpRTE: ∀ ck op v1 v2 msg cks,
    evalBinOpRT ck op v1 v2 (RTE msg) →
    evalBinOpRTS (ck :: cks) op v1 v2 (RTE msg)

    (∗ rule: performing next check returns OK,
        continue iteration over checks ∗)
| ChecksBinOp: ∀ ck op v1 v2 v cks v’,
    evalBinOpRT ck op v1 v2 (OK v) →
    evalBinOpRTS cks op v1 v2 v’ →
    evalBinOpRTS (ck :: cks) op v1 v2 v’.

EvalBinOpRT specifies how a binary expression’s interiorChecks are enforced through evalBinOpRTS, that iterates over the run-time check decorations (CheckBinop). If none of the run-time checks produces an error state, the binary operation is then performed (CheckBinOpNil) using evalBinOpRT (which is defined similarly to evalBinOp in Section 3.3); otherwise, it returns the error state (CheckBinOpRTE). (Note that enforcement of exteriorChecks is not presented above as it involves arrays, which is not presented due to space constraint.)

RT-GEN: translates AST to AST-RT. In developing RT-GEN, we first specified its behavior declaratively as a Coq inductively defined relation (e.g., toExpRT below) between AST to AST-RT (with the symbol table as an auxiliary component). Then, we implemented the
translation as a Coq function (e.g., toExpRTImpl).

**Inductive** toExpRT : symTab → exp → expRT → Prop :=

(* insert checks for overflow on op result *)

| ToBinOpO: ∀ st op e1 e1RT e2 e2RT n,
  op = Add ∨ op = Sub ∨ op = Mul →
  toExpRT st e1 e1RT → toExpRT st e2 e2RT →
  toExpRT st (BinOp n op e1 e2)
  (BinOpRT n op e1RT e2RT [OverflowCheck] nil)

(* for Div, insert checks div by 0 and overflow on op result *)

| ToBinOpDO: ∀ st op e1 e1RT e2 e2RT n,
  op = Div → toExpRT st e1 e1RT →
  toExpRT st e2 e2RT →
  toExpRT
  st
  (BinOp n op e1 e2)
  (BinOpRT n op e1RT e2RT
   [DivCheck, OverflowCheck] nil)

...
(complete) with respect to the SPARK mechanized semantics. For example, for expressions, we proved the following consistency lemma:

**Lemma** toExpRTConsistent : \( \forall \ e \ eRT \ st \ stRT \ s \ v \),

\[ \text{toExpRT} \ st \ e \ eRT \rightarrow \text{toSymTabRT} \ st \ stRT \rightarrow \]

\( (\text{evalExpRT} \ stRT \ s \ eRT \ v \leftrightarrow \text{evalExp} \ s \ s e \ v) \).

where toSymTabRT transforms symTab to symTabRT, which, among other things, maps procedure names to their AST-RT. We then proved that the RT-GEN implementation is consistent with respect to its specification, for example:

**Lemma** toExpRTImplConsistent : \( \forall \ e \ eRT \ stRT \),

\[ \text{toExpRTImpl} \ stRT \ e = eRT \leftrightarrow \text{toExpRT} \ stRT \ e \ eRT \].

Therefore, the implementation is transitively consistent with respect to the SPARK semantics (by transitivity of implication \( \rightarrow /\leftrightarrow \)).

**Lemma** RTGenConsistent : \( \forall \ p \ pRT \ st \ stRT \ s \ s' \),

\[ \text{RTGen} \ st \ p \ pRT \rightarrow \text{toSymTabRT} \ st \ stRT \rightarrow \]

\( (\text{EvalRT} \ stRT \ s \ pRT \ s' \leftrightarrow \text{Eval} \ s \ s p s') \).

**Lemma** RTOptConsistent : \( \forall \ pRT \ pRTOpt \ stRT \ s \ s' \),

\[ \text{RTOpt} \ stRT \ pRT \ pRTOpt \rightarrow \]

\( (\text{EvalRT} \ stRT \ s \ pRT \ s' \leftrightarrow \text{EvalRT} \ stRT \ s \ pRTOpt \ s') \).

\{\( \tau \leq \tau' \)}

\{\( \text{SECRET} \leq \text{SECRET}, \ \text{PUBLIC} \leq \text{PUBLIC} \}\}

\{\( \text{SECRET} \leq \tau, \ \tau' \leq \text{PUBLIC}, \ \tau \leq \tau' \)}

\{\( \text{SECRET} \leq \text{SECRET}, \ \text{PUBLIC} \leq \text{PUBLIC}, \ \text{SECRET} \leq \tau, \ \tau' \leq \text{PUBLIC}, \ \tau \leq \tau' \)}

\{\( \text{SECRET} \leq \text{SECRET}, \ \text{PUBLIC} \leq \text{PUBLIC}, \ \text{SECRET} \leq \tau, \ \tau' \leq \text{PUBLIC}, \ \text{SECRET} \leq \text{PUBLIC} \)\}
4.3 Certified Run-Time Check Optimizer

While RT-GEN generates a sufficient set of run-time checks, some of them may not be necessary. In fact, the GNAT front-end uses optimization techniques to reduce the set of run-time checks that it generates; in practice, we expect the set generated by GNAT to be a subset of the RT-GEN generated set (we confirmed through experiments that this is indeed the case in Section 4.4). The question is then whether GNAT’s optimizations are (certifiably) sound. Our approach to answer this is to have a certified optimizer – RT-OPT, that reduces the run-time checks generated by RT-GEN. Now, it is widely known that, in general, an optimizer cannot actually ever be optimal (due to the halting problem). Thus, the best we can hope for is to have RT-OPT reduce to the same (or even better, i.e., smaller) set as GNAT’s (a smaller set implies that GNAT can be improved further).

RT-OPT: transforms AST-RT to another AST-RT by removing some run-time checks whose corresponding verification conditions (VCs) can be discharged; it discharges the VCs by employing abstract interpretation\textsuperscript{48} with interval numeric domain. Similar to RT-GEN, we first specified RT-OPT as Coq inductively defined relation and then implemented it as a Coq function. For expressions, RT-OPT produces AST-RT along with the expression’s interval domain (if any) as follows:

Inductive optExp : symTabRT \rightarrow 
expRT \rightarrow (expRT \ast \text{interval}) \rightarrow \text{Prop} := . . .

Function optExpImpl (st : symTabRT) 
(e : expRT) : option (expRT \ast \text{interval}) := . . .

where optExp is typeset in Figure 4.3 for readability. One invariant of RT-OPT is that integer expression optimization should produce an interval that fits within the compilation target platform-specific two’s complement integer range, which makes up the default interval \([\text{INT}_\text{MIN}, \text{INT}_\text{MAX}]\). \Gamma holds the abstract interpretation context such as symbol table, etc. For notational convenience, interiorChecks and exteriorChecks are not explicitly shown;
Similarly, for subtract and multiply operations, runtime checks are preserved, and the resulting interval is the platform’s integer range. Sim-
Add2
check associated the binary operation can be safely removed; otherwise,
the case where the bounds are within the platform’s integer range, hence, the overflow
the two operands and compute the expression interval bounds (i.e., \([u,v]\)). On the other hand,
concrete interpretation). On the other hand, int2 specifies the case where the overflow check
is kept whenever \(n\) is outside the range, thus, the default interval is returned (in this case, an
error message can be generated to notify the developer). Add1 and Add2 first try to optimize
the two operands and compute the expression interval bounds (i.e., \([u,v]\)). Add1 specifies the
case where the bounds are within the platform’s integer range, hence, the overflow
check associated the binary operation can be safely removed; otherwise, Add2 specifies that
run-time checks are preserved, and the resulting interval is the platform’s integer range. Sim-
ilarly, for subtract and multiply operations, Sub1 and Sub2 specify the two cases for run-time
checks optimization for subtract expressions, and Mul1 and Mul2 for optimizations of mulplication expressions. In Mul1 and Mul2, \(\min(\min(v1 \cdot v2, v1 \cdot w2), \min(w1 \cdot v2, w1 \cdot w2))\)
and \(\max(\max(v1 \cdot v2, v1 \cdot w2), \max(w1 \cdot v2, w1 \cdot w2))\) compute the lower and upper bound
for multiplication of two values within intervals \([v1, w1]\) and \([v2, w2]\) respectively, which has
been proved correct with Coq.

\[
\begin{align*}
\text{divInterval} & \quad (v1, w1, v2, w2) = \\
& \quad \begin{cases} 
[w1/v2, \min(v1/w2, \text{INT_MAX})], & \text{if } w2 < 0 \land w1 < 0 \\
[w1/w2, v1/v2], & \text{if } w2 < 0 \land v1 > 0 \\
[w1/w2, \min(v1/w2, \text{INT_MAX})], & \text{if } w2 < 0 \land -(w1 < 0 \lor v1 > 0) \\
[v1/v2, w1/w2], & \text{if } v2 > 0 \land w1 < 0 \\
[v1/w2, w1/v2], & \text{if } v2 > 0 \land v1 > 0 \\
[v1/v2, w1/v2], & \text{if } v2 > 0 \land -(w1 < 0 \lor v1 > 0) \\
[v1, \min(|v1|, \text{INT_MAX})], & \text{if } \neg(w2 < 0 \lor v2 > 0) \land w1 < 0 \\
[-w1, w1], & \text{if } \neg(w2 < 0 \lor v2 > 0) \land v1 > 0 \\
[-\max(|v1|, |w1|), \min(\max(|v1|, |w1|), \text{INT_MAX})], & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{modInterval} & \quad (v1, w1, v2, w2) = \\
& \quad \begin{cases} 
[v2 + 1, 0], & \text{if } w2 < 0 \\
[0, w2 - 1], & \text{if } w2 > 0 \\
[\text{if } v2 = 0 \text{ then } 0 \text{ else } v2 + 1, \text{if } w2 = 0 \text{ then } 0 \text{ else } w2 - 1], & \text{otherwise}
\end{cases}
\end{align*}
\]

**Figure 4.2:** Interval Bounds For Divide and Modulus

\(\text{EraseOverflowCheck}\) and \(\text{EraseDivCheck}\) remove overflow and division interiorChecks, respectively, while \(\text{EraseRangeCheck}\) removes range exteriorChecks.

The int1 rule in Figure 4.3 optimizes away the overflow check in the case of an integer literal AST-RT \(n\) where \(n\) is within the the platform’s integer range; a single-value interval \([n, n]\)
is returned along with the optimized AST-RT (i.e., the tight single-value interval allows for
concrete interpretation). On the other hand, int2 specifies the case where the overflow check
is kept whenever \(n\) is outside the range, thus, the default interval is returned (in this case, an
error message can be generated to notify the developer). Add1 and Add2 first try to optimize
the two operands and compute the expression interval bounds (i.e., \([u,v]\)). Add1 specifies the
case where the bounds are within the platform’s integer range, hence, the overflow
check associated the binary operation can be safely removed; otherwise, Add2 specifies that
run-time checks are preserved, and the resulting interval is the platform’s integer range. Sim-
ilarly, for subtract and multiply operations, Sub1 and Sub2 specify the two cases for run-time
checks optimization for subtract expressions, and Mul1 and Mul2 for optimizations of mulplication expressions. In Mul1 and Mul2, \(\min(\min(v1 \cdot v2, v1 \cdot w2), \min(w1 \cdot v2, w1 \cdot w2))\)
and \(\max(\max(v1 \cdot v2, v1 \cdot w2), \max(w1 \cdot v2, w1 \cdot w2))\) compute the lower and upper bound
for multiplication of two values within intervals \([v1, w1]\) and \([v2, w2]\) respectively, which has
been proved correct with Coq.
\[ n \in [\text{INTMIN}, \text{INTMAX}] \]
\[ \Gamma \vdash \text{optExp}(n) = (\text{EraseOverflowCheck}(n), [n, n]) \]
\[ \text{INT1} \]
\[ n \notin [\text{INTMIN}, \text{INTMAX}] \]
\[ \Gamma \vdash \text{optExp}(n) = (n, [\text{INTMIN}, \text{INTMAX}]) \]
\[ \text{INT2} \]
\[ \Gamma \vdash \text{optExp}(e_1) = (e_1', [v_1, w_1]) \]
\[ \Gamma \vdash \text{optExp}(e_2) = (e_2', [v_2, w_2]) \]
\[ v = v_1 + v_2 \]
\[ w = w_1 + w_2 \]
\[ \{v, w\} \subseteq [\text{INTMIN}, \text{INTMAX}] \]
\[ \text{ADD1} \]
\[ \Gamma \vdash \text{optExp}(e_1 + e_2) = (e_1' + e_2', [v, w]) \]
\[ \text{ADD2} \]
\[ \Gamma \vdash \text{optExp}(e_1) = (e_1', [v_1, w_1]) \]
\[ \Gamma \vdash \text{optExp}(e_2) = (e_2', [v_2, w_2]) \]
\[ v = v_1 - v_2 \]
\[ w = w_1 - w_2 \]
\[ \{v, w\} \subseteq [\text{INTMIN}, \text{INTMAX}] \]
\[ \text{SUB1} \]
\[ \Gamma \vdash \text{optExp}(e_1 - e_2) = (\text{EraseOverflowCheck}(e_1' - e_2'), [v, w]) \]
\[ \text{SUB2} \]
\[ \Gamma \vdash \text{optExp}(e_1) = (e_1', [v_1, w_1]) \]
\[ \Gamma \vdash \text{optExp}(e_2) = (e_2', [v_2, w_2]) \]
\[ v = \min(\min(v_1 * v_2, e_1 * w_2), \min(w_1 * v_2, w_1 * w_2)) \]
\[ w = \max(\max(v_1 * v_2, v_1 * w_2), \max(w_1 * v_2, w_1 * w_2)) \]
\[ \text{MUL1} \]
\[ \Gamma \vdash \text{optExp}(e_1 * e_2) = (\text{EraseOverflowCheck}(e_1' * e_2'), [v, w]) \]
\[ \text{MUL2} \]
\[ \Gamma \vdash \text{optExp}(e_1) = (e_1', [v_1, w_1]) \]
\[ \Gamma \vdash \text{optExp}(e_2) = (e_2', [v_2, w_2]) \]
\[ v = \min(\min(v_1 * v_2, e_1 * w_2), \min(w_1 * v_2, w_1 * w_2)) \]
\[ w = \max(\max(v_1 * v_2, v_1 * w_2), \max(w_1 * v_2, w_1 * w_2)) \]
\[ \text{DIV1} \]
\[ \Gamma \vdash \text{optExp}(e_1 / e_2) = (\text{EraseOverflowCheck}(\text{EraseDivCheck}(e_1' / e_2')), \text{divInterval}(v_1, v_2, w_2)) \]
\[ \text{DIV2} \]
\[ \Gamma \vdash \text{optExp}(e_1) = (e_1', [v_1, w_1]) \]
\[ \Gamma \vdash \text{optExp}(e_2) = (e_2', [v_2, w_2]) \]
\[ v = \min(\min(v_1 * v_2, e_1 * w_2), \min(w_1 * v_2, w_1 * w_2)) \]
\[ w = \max(\max(v_1 * v_2, v_1 * w_2), \max(w_1 * v_2, w_1 * w_2)) \]
\[ \text{DIV3} \]
\[ \Gamma \vdash \text{optExp}(e_1 / e_2) = (\text{EraseOverflowCheck}(e_1' / e_2'), \text{divInterval}(v_1, v_2, w_2)) \]
\[ \text{DIV4} \]
\[ \Gamma \vdash \text{optExp}(e_1) = (e_1', [v_1, w_1]) \]
\[ \Gamma \vdash \text{optExp}(e_2) = (e_2', [v_2, w_2]) \]
\[ v = \min(\min(v_1 * v_2, e_1 * w_2), \min(w_1 * v_2, w_1 * w_2)) \]
\[ w = \max(\max(v_1 * v_2, v_1 * w_2), \max(w_1 * v_2, w_1 * w_2)) \]
\[ \text{MOD1} \]
\[ \Gamma \vdash \text{optExp}(e_1) = (e_1', [v_1, w_1]) \]
\[ \Gamma \vdash \text{optExp}(e_2) = (e_2', [v_2, w_2]) \]
\[ v = \min(\min(v_1 * v_2, e_1 * w_2), \min(w_1 * v_2, w_1 * w_2)) \]
\[ w = \max(\max(v_1 * v_2, v_1 * w_2), \max(w_1 * v_2, w_1 * w_2)) \]
\[ \text{MOD2} \]
\[ \Gamma \vdash \text{optExp}(e_1 / e_2) = (\text{EraseOverflowCheck}(e_1' / e_2'), \text{modInterval}(v_1, v_2, w_2)) \]
\[ \text{NEG1} \]
\[ \Gamma \vdash \text{optExp}(e) = (e', [v, w]) \]
\[ \{ -w, -v \} \subseteq [\text{INTMIN}, \text{INTMAX}] \]
\[ \Gamma \vdash \text{optExp}(-e) = (\text{EraseOverflowCheck}(-e'), [-w, -v]) \]
\[ \text{NEG2} \]
\[ \Gamma \vdash \text{optExp}(-e) = (e', [v, w]) \]
\[ \{ -w, -v \} \subseteq [\text{INTMIN}, \text{INTMAX}] \]
\[ \Gamma \vdash \text{optExp}(-e) = (\text{EraseOverflowCheck}(-e'), [-w, -v]) \]
\[ \text{VARINT} \]
\[ \Gamma \vdash \text{optExp}(x) = (x, \text{max}(\text{tauMIN}, \text{INTMIN}), \text{min}(\text{tauMAX}, \text{INTMAX})) \]

Figure 4.3: RT-OPT Specification for Expression (excerpts)
Γ ⊢ optName(a) = (a’, Aggregate)  Γ ⊢ optExp(e) = (e’, [v₁, w₁])
Γ ⊢ arrayIndexTypeRange(a) = [l, u]  Γ ⊢ arrayComponentTypeBound(a) = componentBound  {v₁, w₁} ⊆ [l, u]
Γ ⊢ optName(a[e]) = (EraseRangeCheck(a'[e']), componentBound)

Figure 4.4: RT-OPT Specification for Array and Record Access

For division, four cases (Div1-4) specify the different situations where division by zero and/or operation overflow (i.e., when dividing INT_MIN by -1) could occur; in all the cases, the resulting interval is specified by divInterval (see Figure 4.2) that does case analysis on the positivity/negativity of the interval operands. For example, (5) specifies the case where both of the operand intervals [v₁, w₁] and [v₂, w₂] are positive (i.e., the low bounds v₁ and v₂ are positive); in this case, the resulting interval is [v₁/w₂, w₁/v₂] where its low bound v₁/w₂ is computed by dividing the smallest value of the first operand’s interval with the largest value of the second operand’s interval, and its high bound w₁/v₂ is computed by dividing the largest value of the first operand’s interval with the smallest value of the second operand’s interval. The divInterval specification illustrates a slice of the RT-OPT’s complexity for computing tight intervals in order to optimize away many run-time checks; rest assured however that they are proven to be correct (and tested to boot!). It’s similar for modulus operator that is specified with Mod1 and Mod2, whose resulting interval is specified by modInterval in Figure 4.2.

For unary negative operator, the rule Neg1 safely removes the overflow check for the unary expression when all possible values of the negative operation on its operand are within the integer range; and rule Neg2 preserves the check as the result of negation may cause overflow (i.e. negation of INT_MIN).

Lastly, the VarInt rule specifies that an integer variable reference’s interval is its integer type range intersected by the platform’s integer range (i.e., leveraging the RT-OPT invariant
that all computed integer values are always checked for overflows).

Figure 4.4 shows the check optimization rules for array and record access. As array and record are defined as name in SPARK AST syntax in Coq (see Figure 3.4), optName is used for specifying the optimization of name expressions to distinguish with optExp for normal expressions. Array1 and Array2 define the optimization of both interiorChecks and exteriorChecks for array expressions. As array can be nested, optName(a[e]) is defined recursively by calling optName(a), and then do the interiorChecks optimization for indexing expression e by calling optExp(e); if the value interval [v₁, w₁] of optExp(e) falls within the array index type range [l, u], remove the range exteriorChecks from the array expression as specified in Array1, otherwise, keep the range check in Array2. The resulting array component bound can be either (a) an interval [v, w], bounded by machine integer range [INT_MIN, INT_MAX], if it’s an array of integer; or (b) Boolval if it’s an array of boolean values; or (c) Aggregate if it’s a nested array (e.g. array of array, array of record). The rule Record shows the similar optimization for record expressions, and its resulting bound value fieldBound can also be an integer interval [v, w], Boolval or Aggregate, depending on the type of the visited record field.

Figure 4.5: RT-OPT Specification for Statement
Γ ⊢ optArgs(nil, nil) = (nil) \hspace{1cm} \text{ARGS1}

Γ ⊢ mode(p) = In \hspace{.1cm} Γ ⊢ rangeConstrained(p) = false
Γ ⊢ optExp(a) = (a', aBound) \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ optArgs(a :: args, p :: params) = (a' :: args') \hspace{1cm} \text{ARGS2}

Γ ⊢ mode(p) = In \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} \{v_1, w_1\} ⊆ [v_2, w_2]
Γ ⊢ optArgs(a :: args, p :: params) = (EraseRangeCheck(a') :: args') \hspace{1cm} \text{ARGS3}

Γ ⊢ mode(p) = In \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} \{v_1, w_1\} ⊆ [v_2, w_2]
Γ ⊢ optArgs(a :: args, p :: params) = (a' :: args') \hspace{1cm} \text{ARGS4}

Γ ⊢ mode(p) = Out \hspace{.1cm} Γ ⊢ rangeConstrained(a) = false
Γ ⊢ optExp(a) = (a', aBound) \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ optArgs(a :: args, p :: params) = (false) \hspace{1cm} \text{ARGS5}

Γ ⊢ mode(p) = Out \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} \{v_1, w_1\} ⊆ [v_2, w_2]
Γ ⊢ optArgs(a :: args, p :: params) = (EraseRangeCheckOnReturn(a') :: args') \hspace{1cm} \text{ARGS6}

Γ ⊢ mode(p) = Out \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} \{v_1, w_1\} ⊆ [v_2, w_2]
Γ ⊢ optArgs(a :: args, p :: params) = (a' :: args') \hspace{1cm} \text{ARGS7}

Γ ⊢ mode(p) = InOut \hspace{.1cm} Γ ⊢ rangeConstrained(p) = false \hspace{.1cm} Γ ⊢ rangeConstrained(a) = false
Γ ⊢ optExp(a) = (a', aBound) \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ optArgs(a :: args, p :: params) = (a' :: args') \hspace{1cm} \text{ARGS8}

Γ ⊢ mode(p) = InOut \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} v_1 = v_2 \hspace{.1cm} w_1 = w_2
Γ ⊢ optArgs(a :: args, p :: params) = (EraseRangeCheckOnReturn(EraseRangeCheck(a')) :: args') \hspace{1cm} \text{ARGS9}

Γ ⊢ mode(p) = InOut \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} \{v_1, w_1\} ⊆ [v_2, w_2] \hspace{.1cm} \{v_2, w_2\} ⊆ [v_1, w_1]
Γ ⊢ optArgs(a :: args, p :: params) = (EraseRangeCheck(a') :: args') \hspace{1cm} \text{ARGS10}

Γ ⊢ mode(p) = InOut \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} \{v_1, w_1\} ⊆ [v_2, w_2] \hspace{.1cm} \{v_2, w_2\} ⊆ [v_1, w_1]
Γ ⊢ optArgs(a :: args, p :: params) = (EraseRangeCheckOnReturn(a') :: args') \hspace{1cm} \text{ARGS11}

Γ ⊢ mode(p) = InOut \hspace{.1cm} Γ ⊢ optArgs(args, params) = (args')
Γ ⊢ paramTypeBound(p) = [v_1, w_1] \hspace{.1cm} Γ ⊢ optExp(a) = (a', [v_2, w_2]) \hspace{.1cm} \{v_1, w_1\} ⊆ [v_2, w_2] \hspace{.1cm} \{v_2, w_2\} ⊆ [v_1, w_1]
Γ ⊢ optArgs(a :: args, p :: params) = (a' :: args') \hspace{1cm} \text{ARGS11}

\textbf{Figure 4.6: RT-OPT Specification for Arguments}
Figure 4.5 lists the optimization rules for statements. For assignments, as illustrated in Assign1, Assign2, and Assign3, perform optimizations on both left and right side expressions; the optimization for range exteriorChecks is carried out in Assign2 only when $x$ is a range constrained variable and the value bound of $e$ cannot escape the expected value range of $x$, where the predicate $rangeConstrained(x)$ checks whether $x$ is a variable of range constrained type. Rules if, while and seq define run-time check optimizations recursively on their subcomponents. For procedure call, run-time checks are required when evaluating and passing the values of arguments to the callee procedure and when returning the result to the calling procedure. The procedure call rule call specifies the optimization for its arguments against to its formal parameters.

Figure 4.6 gives all possible cases for check optimization of arguments with respect to different in/out modes of the corresponding parameters. Args1 is a special case for procedure call without parameters; Args2-4 for arguments that are only readable by the called procedure, and Args5-7 for arguments that are only writable by the called procedure, and Args8-11 for both readable and writable arguments. In general, there are two kinds of exteriorChecks optimizations for arguments when calling and returning from the called procedure: EraseRangeCheck is to remove the range check RangeCheck and EraseRangeCheckOnReturn is to remove the range check RangeCheckOnReturn. Both RangeCheck and RangeCheckOnReturn enforce the same kind of range check, but at different situations and in different directions. RangeCheck enforces the range check on the values of In/InOut arguments before they are passed into the parameters when the procedure is called, and RangeCheckOnReturn enforces the range check on the values of parameters before they are written to the Out/InOut arguments when the procedure returns. Both range checks reside in arguments AST.

Well-Typed State: VariInt assumes that it can use the variable’s integer type range for the variable reference’s interval. This holds if all values in the state are well-typed. To discharge this assumption, we first specified the meaning for a state to be well-typed:

**Inductive** wellTypedState : symTabRT → state → Prop:=

58
WellTypedState: ∀ stRT s, 
(∀ x v, fetch x s = Some v → 
∃ t, lookup stRT x = Some t ∧ 
wellTypedValue t v) → 
wellTypedState stRT s.

then proved that EVAL-RT specification (hence, by virtue of consistency transitivity, EVAL specification) preserve state well-typed-ness, for example, for EVAL-RT statement semantics that may incur state changes, we proved the following preservation lemma:

Lemma wellTypedStatePreservation: ∀ s s’ stmt stmtRT st stRT, 
toSymTabRT st stRT → 
toStmtRT st stmt stmtRT → 
wellTypedStmt stRT stmtRT → 
wellTypedState stRT s → 
evalStmtRT stRT s stmtRT s’ → 
wellTypedState stRT s’.

Evaluation: To certify RT-OPT, we proceeded similarly to RT-GEN certification (albeit much more complex to prove); that is, we proved that RT-OPT specification is consistent (sound and complete) with the respect to the RT-GEN specification described in Section 4.2, and that RT-OPT implementation is consistent with respect to its specification. Therefore, the implementation is transitively consistent with respect to the SPARK mechanized semantics. For example, for statements, we proved the following consistency lemma:

Lemma optStmtConsistent: ∀ stmt stmtRT stmtRTOpt st stRT s s’, 
toStmtRT st stmt stmtRT → 
toSymTabRT st stRT → 
wellTypedStmt stRT stmtRT → 
wellTypedState stRT s →
We then proved that the RT-OPT implementation is consistent with respect to its specification, for example:

**Lemma** optStmtImplConsistent : \( \forall \) stRT stmt stmtRT,

\[
\text{optStmtImpl stRT stmt = Some stmtRT } \leftrightarrow \text{optStmt stRT stmt stmtRT}.
\]

Therefore, the implementation is transitively consistent with respect to the SPARK semantics (by transitivity of implication \( \rightarrow /\leftrightarrow \)).

### 4.4 Certifying GNAT RT Generator

Now that we have RT-GEN and RT-OPT, we can implement a conformance checker that can establish that, for a SPARK 2014 program \( p \), the run-time check decoration insertion of the GNAT front-end for \( p \) conforms to the mechanized SPARK 2014 reference semantics. Specifically, for program \( p \), the GNAT front-end generates a fully resolved AST with run-time check decorations, and we developed a tool called Jago that takes the GNAT AST and produces: (1) a Coq object of type \( \text{AST} (\text{ast}) \), where the GNAT run-time decorations are erased, and (2) a Coq object of type \( \text{AST-RT} (\text{ast-rt-gnat}) \), where the GNAT run-time decorations are preserved (Jago also applies some program transformations to desugar language constructs that lie outside of the language subset to fall within the language subset). Then, applying RT-GEN on \( \text{ast} \) produces \( \text{ast-rt-gen} \), and applying RT-OPT on \( \text{ast-rt-gen} \) produces \( \text{ast-rt-opt} \), both of which are of type \( \text{AST-RT} \).

To automate the actual AST conformance check, we implemented a tool in Coq \( \subseteq \), that given two objects of type \( \text{AST-RT} \), it determines whether the set of run-time checks in the first object is a subset of the second’s. Thus, GNAT run-time decoration insertion on program \( p \) is conformant to the SPARK 2014 reference semantics if \( \text{ast-rt-opt} \subseteq \text{ast-rt-gnat} \subseteq \text{ast-rt-gen} \). This toolchain essentially turns the GNAT front-end into a certifying run-time check dec-
oration generator; that is, for a given program \( p \), it generates evidence of “conformity to SPARK 2014 reference semantics for \( p \)’s run-time check decorations” that is automatically machine-checked by certified tools.

Note that this does not guarantee that the actual binary run-time check assertion code for \( p \) subsequently generated by the GNAT compiler back-end is correct; it simply means that decorations indicating what assertions should be produced is correct. This, alone has significant value because, for example, it goes a long way toward establishing the correspondence between GNAT and GNATprove’s (as well as any other SPARK backend tools’) treatment of run-time checks. Moreover, since there are only three categories of run-time checks relevant for this language subset, since each of these categories can be represented by a simple code pattern involving a few numerical comparisons, since the pattern itself can be easily inspected and tested, and since the generation of binary code for the pattern is reasonable straightforward and can also be easily tested, one might argue that establishing the correctness of the decorations is one of the more important steps in establishing trust in the overall end-to-end production of the executable run-time checks.

4.5 Evaluation: Certifying GNAT

We evaluated GNAT according to the methodology described in Section 4.4 on a collection of programs. Table 4.1 presents the experiment data for various program units (packages/procedures) from the test programs. The first two SPARK programs come from the Ada Conformity Assessment Test Suite (ACATS)\(^{49} \) that all Ada compilers must pass. SPARKSkein is an implementation of the Skein hash algorithm in SPARK, which was proved free of runtime errors\(^ {50} \). Tetris is the motivating example for run-time check certification, which is implemented partly in SPARK and partly in Ada (we only checked the SPARK part). All other examples are representative code from AdaCore, Altran and our own designed benchmark covering the core language subset. For each unit, \textbf{LoC} in Table 5.1 gives the unit’s
Table 4.1: Experiment Data

number of lines of code. **Base** gives the number of run-time checks in `ast-rt-gen`, **GNAT** gives the number of run-time checks in `ast-rt-gnat`, **Opt** gives the number of run-time checks in `ast-rt-opt`, and **Diff** represents the number of run-time checks in **GNAT** that differs than the ones in **Opt**. Dash (“–”) means “none”; a negative number `-n` in **Diff** means that **Opt** removes `n` more run-time checks than **GNAT**; and, a positive number `+m` means **Opt** has `m` more number of run-time checks than **GNAT** (“somehow”). Sub-column **D** gives the number of division by zero run-time checks; **O** gives the number of overflow run-time checks; **R** gives the number of range run-time checks; and, **T** is the total number of run-time checks (i.e., `D+O+R`).

<table>
<thead>
<tr>
<th>Unit</th>
<th>LoC</th>
<th>Base</th>
<th>GNAT</th>
<th>Opt</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACATS_c53007a</td>
<td>143</td>
<td>-16</td>
<td>-16</td>
<td>-14</td>
<td>-14</td>
</tr>
<tr>
<td>ACATS_c55002b</td>
<td>74</td>
<td>-2</td>
<td>7</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>array_record_package</td>
<td>54</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>array_subtype_index</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>arrayrecord</td>
<td>43</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>assign_subtype_var</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>binary_search</td>
<td>40</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bounded_in_out</td>
<td>17</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>-3</td>
</tr>
<tr>
<td>dependence_test_suite_01</td>
<td>164</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>division_by_non_zero</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>faultintegrator</td>
<td>25</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>gcd</td>
<td>18</td>
<td>1</td>
<td>3</td>
<td>-4</td>
<td>1</td>
</tr>
<tr>
<td>gnatprove_test_bool</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>linear_div</td>
<td>21</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>modulus</td>
<td>24</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>odd</td>
<td>14</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>p_simple_call</td>
<td>36</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>prime</td>
<td>21</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>procedure_test01</td>
<td>25</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>quantified_test</td>
<td>14</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>recordtest01</td>
<td>23</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>recursive_proc_pkg</td>
<td>18</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>SPARKSkein</td>
<td>646</td>
<td>7</td>
<td>94</td>
<td>246</td>
<td>347</td>
</tr>
<tr>
<td>sort</td>
<td>43</td>
<td>-5</td>
<td>6</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>Tetris</td>
<td>373</td>
<td>-29</td>
<td>58</td>
<td>87</td>
<td>-25</td>
</tr>
<tr>
<td>the_stack</td>
<td>42</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>the_stack_praxis</td>
<td>35</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>two_way_sort</td>
<td>49</td>
<td>-4</td>
<td>17</td>
<td>21</td>
<td>-4</td>
</tr>
</tbody>
</table>

(i.e., `D+O+R`).
As can be observed from Table 4.1, the GNAT frontend is a solid tool for run-time check generation/verification as most of its generated run-time checks match the certified RT-OPT. This is reasonable because our formalization captures the most commonly used run-time checks in SPARK and GNAT is quite mature after many years of effort to improve it, as well as the effort to improve the GNATprove toolchain by AdaCore and Altran (which drives some of the improvements in GNAT). However, RT-OPT edges out GNAT in some cases, especially for SPARKSkein. One reason is that GNAT does not take any advanced optimizations, for the division/modulus binary operator, it does not optimize even with constant; for example, GNAT generates division by zero check for the expression \((R + 1) \mod 3\) while RT-OPT optimized it away.

For SPARKSkein, consider a procedure call \(\text{Inject\_Key}(R \times 2)\), (for procedure declaration \(\text{Inject\_Key}(X: \text{in U32})\)), \(R\) is a variable of type U32, and U32 is a subtype of Integer with range \(0..\text{INT\_MAX}\); an overflow check for \(R \times 2\) is enough to guarantee the absence of both overflow and range error, while GNAT keeps both overflow check and range check for such cases. For expression \((\text{Byte\_Count} - (\text{Blocks\_Done} \times 64))\), where \(\text{Byte\_Count}\) is a variable of U32 and \(\text{Block\_Done}\) is a variable of U32’s subtype with range \(1..2\), the overflow check for both subtract and multiply operators, as requested by GNAT, can be optimized by RT-OPT. Similarly, for assignment \(\text{Bytes\_Hashed} := \text{Block\_Count} \times 64\), the overflow check for multiply operator and the range check against the type of \(\text{Bytes\_Hashed}\) can all be safely removed by RT-OPT.

There are other cases showing that RT-OPT is better than GNAT’s optimizations: for binary_search example, the division check on expression \((\text{Right} - \text{Left}) / 2\) can be safely removed as specified in RT-OPT, while GNAT makes no optimization on division expressions with even constant divisor 2; this also applies to the modular operator as shown in modulus example, where the division check for \(\mod\), as required by GNAT, within the array expression \(\text{KS}((R + I) \mod (8 + 1))\) will be optimized by RT-OPT. The similar optimizations by RT-OPT vs GNAT can be found in division_by_non_zero and odd examples.
In some cases, GNAT produces fewer run-time checks than RT-OPT; these inconsistencies are benign because they are due to differences in how GNAT (vs RT-OPT) reports the need for checks and in how it assumes down-stream translation will interpret decorations for run-time checks.

Regarding the reporting issue, in procedure array_subtype_index, there is an assignment A(0) := 0, where the index type of A is a subtype of integer with range 1 .. 10; thus, accessing A with the index 0 is out of its required range, so it will cause range error. GNAT gives a warning at compile time in such case, and it does not generate a range check; on the other hand, RT-OPT keeps this check. (A similar issue exists for assign_subtype_var.)

Another inconsistency is due to the fact that a single run-time check decoration in the GNAT AST can lead downstream translation steps to introduce run-time checking code that implements multiple checks. Consider bounded_in_out:

```plaintext
subtype T1 is Integer range 0 .. 10;
subtype T2 is Integer range 5 .. 15;
procedure Decrease (X: in out T1) is
begin  X := X − 1;  end Decrease;
V: T2; . . . ; Decrease (V);
```

when Decrease(V) is called, according to the SPARK semantics, there should be two different range checks for argument V on both passing in argument and passing out return value because X is declared using both in and out. However, GNAT only generates one range check decoration; this is because it is assumed that the check decoration should be interpreted according to the variable in/out modes. That is, if the variable is (strictly) either in or out (and not both), then the run-time check decoration should be interpreted giving rise to code for one run-time check. However, if the variable is both in and out, then the GNAT run-time check AST decoration should be interpreted as giving rise to code for two checks, one for each direction. On the other hand, RT-GEN explicitly generates the two check decorations (which are kept by RT-OPT).
The inconsistencies above can be easily rectified by either modifying GNAT or RT-GEN/RT-OPT/$\subseteq$; they are kept here to document our findings.

**Further Assessment:** The fact that RT-OPT is better in some cases illustrates that, despite its maturity, GNAT can still be improved further, e.g., by adopting the optimization specified and implemented in RT-OPT; that is, the RT-OPT specification can be used as a reference for implementing further optimizations in GNAT, and once implemented, they can then be checked for conformance against the RT-OPT implementation. Furthermore, in the case where new optimizations are added to GNAT that goes beyond RT-OPT as presented here, those new optimizations can be added to RT-OPT in order to: (a) mechanically verify that they are correct, and (b) further keep GNAT as a certifying run-time check generator.

Our research work demonstrates the feasibility of engineering an approach and corresponding tools with mechanized correctness proofs that leverage recent advancements and maturity of various formal method techniques and tools to make a direct impact in significantly increasing confidence in industrial tools; in our case, the industrial tools are used to develop critical systems that require the utmost level of integrity, thus, warranting such effort.

From a business perspective, we believe it is desirable as it adds to the value proposition – the trustworthiness of GNAT compiler and associated SPARK 2014 is increased. Furthermore, we believe that the approach can eventually help in tool qualification processes typically done in software certifications and regulatory reviews associated with standards (e.g., DO-178C in avionics) that increasingly recognize the value of formal methods and an official tool qualification process.

**Threats To Internal Validity:** Our approach is predicated on the assumption that practitioners are willing to trust the approach’s trust-base, which includes Coq and the SPARK formal language semantics presented in Section 3.3. That is, problems in Coq or in the formal semantics threaten the internal validity of our approach. As to why this might be a concern, recent stable releases of Coq (e.g., v8.4pl5) have an unguarded optimization bug
that cause them to accept a proof of false ($\perp$) in rare cases where an inductive type is defined
with more than 255 type constructors; once $\perp$ is accepted, any proof can be accepted,
which essentially renders any proof system untrustable. Understandably, the bug has since
been fixed (starting in Coq v8.4pl6). In addition, our current implementation uses: (a) the
parser, symbol resolver, and type checker of GNAT itself, and (b) Jago to build program
representations in Coq; both are not certified tools. Ideally, a certified frontend can be
developed to address this issue; this certified frontend is orthogonal and out of the scope
of the work presented here, and they can be addressed in the future. Moreover, the $\subseteq$ tool
that compares AST-RT objects is manually inspected instead of certified (it is small – 172
LoC, and its functionality is very simple); regardless, it should be considered as part of the
trust-base at this point of time.

**Threats To External Validity:** One must also consider the extent to which the results
presented for the given test suite would extend to SPARK 2014 programs in general. For
this objective, program size and execution time are not really issues – the cost of insertion
of run-time checks is in general linear in the number of AST nodes. The interval analysis
needed for optimization does add some additional complexity, but not enough to significantly
impact performance. On the other hand, a principle concern is that our test suite provides
appropriate coverage of all the different types of run-time checks specified in the SPARK
2014 language reference manual. In addition, our language subset needs to be expanded to
eventually cover the full programming language (in fact, this work represents our second
iteration; we presented our initial work with a small language subset corresponding to a
simple imperative language with while-loop as a 2-page abstract paper for an industrial
position presentation).

**Overall Perspective:** In the end, if GNAT/GNATProve is a run-time checker, our ap-
proach is a checker of checker, and Coq is the checker of a checker of a checker. Both
Coq and the SPARK formal semantics are ultimately checked by manual inspections (which
would improve over time). So, who checks the checkers? It seems the answer lies in a
well-engineered towering house of checkers with decreasingly unsafe spaces\textsuperscript{1} as one ascends the tower and eventually comes to the realization of having to trust (hopefully minimal) creations of intelligent beings.

\textsuperscript{1}Analogous to “reducing attack surfaces” commonly touted in cybersecurity to manage risks.
Chapter 5

Declassification Policy for SPARK

There exist a wide range of safety/security critical information systems being used in the area of finance, healthcare, military and aviation to process multi-level security data, including password checking system, online shopping and banking system. In general, there are three ways to protect information confidentiality and integrity. Cryptography provides a way to hide information in data and protect data from tampering, but it is costly and impractical to decrypt and encrypt the data each time we need to process the data. Access controls can protect data from being read or modified by unauthorized users. However, it cannot prevent the propagation of information after it has been released for processing by a program. Information flow is the transfer of information from an input to an output in a given process. It reflects an end-to-end behavior of a system and information flow control provides a complementary approach to track and regulate the information flow of a system to prevent secret data from leaking into public. One promising way to ensure the secure information flow of a system is to use noninterference, which requires that secret data may not interfere with (or affect) public data, as an end-to-end semantic security condition to reason about information flow security.

However, the security requirements enforced by noninterference are too restrictive. In fact, computing systems often need to deliberately declassify (or release) parts of its confi-
dential information, for example, in password checking system, it’s necessary to reveal some
information about the stored password, telling the user whether his input password is correct
or not. Declassification of information occurs when the confidentiality of information is de-
creased or becomes less restrictive. It’s an exception to the normal secure information flow.
The major challenge is to design an elegant declassification mechanism to precisely capture
the intentional information release and ensure that the release is safe: the attacker could
neither get around the declassification mechanism nor exploit the declassification mechanism
to reveal more secret information than intended, e.g. secret laundering. A lot of research
has been carried out in the area of declassification according to four dimensions\(^53\): \textit{what} can
be declassified, \textit{where} declassification can occur, \textit{who} is able to declassify information and
\textit{when} it’s allowed to be declassified.

The main contributions of this chapter are:

- a design of a language-based information security policy specification framework to
capture the desired information flow from one domain to another on the event of
certain actions, and show how the framework can be specified and integrated into
SPARK using Ada 2012 aspects decorations. It enhances SPARK current information
flow analysis with ability to declare domains and specify declassification policy between
those domains.

- a design of a static enforcement algorithm that can automatically check whether the
program conforms to the specified information security policy. It’s a type checking
system that based on type constraint generation followed by constraint solving pro-
cedure. The checking algorithm is compositional, and the verification of information
security can be achieved modularly, at the level of individual subprograms, such that
a called procedure can be checked with its inferred type constraint specification.

- formalization of the operational semantics for programs with declassification proce-
dures and proof of the correctness of the policy enforcement algorithm with respect
to the formalized program semantics.
• evaluation of our designed security policy framework on some examples to assess the expressiveness of the policy language and the degree of automation provided by the algorithm to verify that the program behaves as expected.

5.1 Syntax of Security Policy

5.1.1 Policy Design Principles

Our goal is to propose a security policy framework for SPARK that can provide the ability to specify security domains and control information flow of different domains. Here are some design principles that we have adopted:

• Simple to use, which means little burden for users to specify policy.

• Easy to integrate into SPARK 2014.

• Easy to check with static analysis.

• Expressive enough to specify and check security policy for real SPARK programs.

5.1.2 Policy Aspects

As SPARK’s contract language is aspect-oriented, to be consistent with this language feature, we also propose some new aspects to support the security policy, which has ability to specify:

• the security domains as enumeration types, with or without ordering.

• the association between data and security domains.

• the allowed declassification operations.
**Ordered Aspect**  We introduce the security type as a natural part of the SPARK language by adding a new aspect *Ordered* to the normal enumeration type. With this new aspect, a type can be annotated to indicate different types of security domains. The symbol $\leq$ is used to denote the ordering between two security domains and its antisymmetry. For any two security domains $\ell_1$ and $\ell_2$, $\ell_1 \leq \ell_2$ means the security level of $\ell_2$ is at least as high as the security level of $\ell_1$, so any information from the security domain $\ell_1$ (or annotated with $\ell_1$) is allowed to flow into the destination of security domain $\ell_2$.

Totally ordered security domains can be defined as:

```
type T is (\ell_1, \ell_2, \ell_3) with
  Ordered;
```

where the defined security domain values are ordered according to their positions in the declared enumeration type $T$. That’s $\ell_1 \leq \ell_2, \ell_2 \leq \ell_3$ and $\ell_1 \leq \ell_3$.

Partially ordered security domains can be defined as:

```
type T is (\ell_1, \ell_2, \ell_3) with
  Ordered $\Rightarrow \{\ell_1 \leq \ell_2\}$;
```

which means $T$ is a security type and only $\ell_1$ and $\ell_2$ are ordered with $\leq$. The security domain $\ell_3$ is isolated from the other two security domains.

Totally separated security domains can be defined as:

```
type T is (\ell_1, \ell_2, \ell_3) with
  Ordered $\Rightarrow$ null;
```

which means that all security domains defined in security type $T$ should be isolated from each other.

The same idea could also be used to define integrity domains, where the normal flows go from critical data to non-critical data. In this work, we mainly focus on the confidentiality (also referred as security) domains. Specification of multiple dimensional domains (both confidentiality and integrity) will be included in the future work.
Domain Aspect  With the definition of security types, data can be annotated with the new aspect *Domain* to indicate that it belongs to a security domain, for example:

\[ X : \text{Integer with Domain } \Rightarrow \ell \]

where \( X \) is defined as an integer variable within security domain \( \ell \).

The security domains in these annotations are all static, it means that any data should belong to exactly the listed domain, and it cannot belong to two different incompatible domains at the same time. The associated domain can be regarded as a invariant property for the data, whatever information flow involving the domain-associated data should follow the security policy enforced on its domain.

In this proposal, we are limiting this feature to global variables, as the extension to library-level variables, subprogram parameters/results, and local variables is likely increasing the work in the tools for no clear benefit. Furthermore, to make it simpler and more precise, we can only associate domains for those security-critical global variables that may either hold sensitive data that should be protected or separated from other domains or work as source input ports that provide sensitive input information or work as output ports that deliver information to the public or other sessions.

Declassifier Aspect  Verifying that data flows respect the allowed flows between domains can be achieved modularly, at the level of individual subprograms, based on *Depends* contracts. Indeed, all dependencies listed in *Depends* contracts (either manually written or generated by the tool) should respect the allowed directions of flow. For example, the following would be allowed if the security domain of \( Y \) is at least as high as \( X \):

```
procedure Secure_Copy (X: in T; Y: out T) with
Depends \Rightarrow (Y \Rightarrow X); \text{ --- read as: } Y \text{ depends on } X
```

while the following would be detected as a violation of the security policy if security domains of \( X \) and \( Y \) have no \( \leq \) ordering:

```
procedure Insecure_Copy (X: in T; Y: out T) with
```
Depends \Rightarrow (Y \Rightarrow X); \quad \text{read as: } Y \text{ depends on } X

It is almost never the case that all flows respect the simple security policy stated above. In general, a few operations known as declassifications are allowed to violate the default security policy, for example the operation that encrypts secret data before sending it on a public network.

We propose to allow specifying which operations are declassifiers with a new aspect \textit{Declassifier}. Such operations should specify fully the possible source and target domains for the declassification(s). For example:

\begin{itemize}
  \item \textit{with domains } \ell_1 \text{ and } \ell_2 \text{ and } \ell_2 \leq \ell_1
  \item \textbf{procedure} Encrypt (X: \textbf{in} T; Y: \textbf{out} T) \textbf{with}
    \begin{itemize}
      \item \textbf{Declassifier},
      \item Domain \Rightarrow (X \Rightarrow \ell_1, \ Y \Rightarrow \ell_2),
      \item Depends \Rightarrow (Y \Rightarrow X); \quad \text{read as: } Y \text{ depends on } X
    \end{itemize}
\end{itemize}

it means that \textit{Encrypt} is a declassification procedure that’s allowed to declassify information from domain \ell_1 to \ell_2. The domain annotations for parameters X and Y also enforce that whenever calling \textit{Encrypt}(u, v), the security domain of passed in argument u should be less than and equal to \ell_1, and when the call returns, \ell_2 should be less than and equal to the security domain of the passed out argument v.

For those procedures denoted as declassifiers, they are assumed to be trustful and being implemented correctly. So unlike the non-declassifier procedures, we don’t need to get into its implementation body during static analysis.

In summary, the security policy language is quite simple and easy to use, what users need to do is just to define security types of their interest, and then annotate those critical source/target data with the defined security domains, and specify the trusted declassification procedures. The enforcement for checking security policy will be performed automatically, which will be discussed in the next section.
5.2 Enforcement of Security Policy

The introduction of declassification means that not all subprograms can be analyzed simply by checking that the final variable dependencies respect the security policy, as violations may occur which are allowed by some declassification step. Thus, flow analysis should be updated to analyze each assignment for its respect of security policies, whether this is a direct assignment (assignment operation) or an indirect one (call to a procedure that has outputs). In both cases, there are restrictions on the flows from input variables (either from the expression assigned, or the inputs of the call on which this output depends, or the control under which this assignment takes place) to the assigned variable. In this chapter, we propose to effectively enforce these restrictions by a constraint-based typing system, which is proved to be sound with respect to the operational semantics of program with declassification procedures. Any program satisfying the type constraint system is called an information secure program.

Here are some symbols that will be used in the following sections:

- $\ell$: concrete security domain (or security level)
- $\alpha$: security type variable
- $C$: set of subtyping constraints, which are in the form of $T_1 \leq T_2$
- $p$: context security type served to eliminate indirect information flows
- $\Gamma_g$: a mapping from global variables to security types
- $\Gamma_l$: a mapping from local variables to security types
- $\text{freshType}()$: generate fresh security type variable
- $\text{freshVar}()$: generate fresh variable name
- $\text{ModVar}_{\text{local}}(c)$: set of local variables maybe modified within statement $c$
A security type constraint (also referred as type constraint) is an ordering (or subtyping) relationship between two type expressions (see Figure 5.2): $T_1 \leq T_2$, and it’s called an atomic type constraint if both $T_1$ and $T_2$ are $\tau$, such as $\alpha \leq \ell$, $\ell_1 \leq \ell_2$.

### 5.2.1 Type Constraint Generation

As mentioned before, user only needs to annotate the security types for global variables of their interests, and the enforcement can infer security types for the rest, including locally defined variables and parameters of procedures. The security policy is enforced by generating type constraints at each execution step. The satisfiability of the type constraint system determines the information security of the program, and any solution for the type constraint system is also a type inference for the untyped variables.

Figure 5.1 and 5.2 show the type inference rules for expression and type constraint generation rules for statement respectively. In the type inference rules for expression, a constant can flow to any domain as denoted by type $\perp$ in the rule LITERAL. We treat differently for the security types of global and local variables. The security type for a global variable is fixed, while it’s varied for the local variable depending on its current content. This is because in the information security, we concern about the security of information flow between global variables, which maybe accessible to the outside, and local variables are only working as
Γ;Γ1 ⊢ e : T,
Γ(x) ≠ null,
α = freshType(),
C = {T ≤ α, p ≤ α},
Γ′ = [x → α]

Γ;Γ1 ⊢ e ⇒ Γ′;C

ASSIGN-LOCAL-UPDATE

Γ;Γ1 ⊢ p ⊢ x := e ⇒ Γ′;C

ASSIGN-GLOBAL-UPDATE

Γ;Γ1 ⊢ x := e ⇒ Γ′;C

π = ModVarLocal(c) = {x1, . . . , xk},
Γ′ = Γ1[x1 → α1,...,xk → αk], where αi = freshType(), i ∈ [1, k],
Γ;Γ1 ⊢ e : T,
Γ;Γ1 ⊢ p ⊢ T ⊢ c ⇒ Γ′;C,
C0 = {Γ1(x) ⊆ Γ1′(x) | x ∈ Ψ},
C1 = {Γ1′(x) ≤ Γ1′(x) | x ∈ Ψ},
C′ = C0 ∪ C1 ∪ C′

Γ;Γ1 ⊢ while e do c ⇒ Γ′;C

ψ = ModVarLocal(c) = {x1, . . . , xk},
Γ′ = Γ1[x1 → α1,...,xk → αk], where αi = freshType(), i ∈ [1, k],
Γ;Γ1 ⊢ e : T,
Γ;Γ1 ⊢ p ⊢ T ⊢ c ⇒ Γ′;C,
C0 = {Γ1(x) ⊆ Γ1′(x) | x ∈ Ψ},
C1 = {Γ1′(x) ≤ Γ1′(x) | x ∈ Ψ},
C′ = C0 ∪ C1 ∪ C′

Γ;Γ1 ⊢ while e do c ⇒ Γ′;C

PROCEDURE-DECL

Γ;Γ1 ⊢ call f(a1, a2, . . . , ak) ⇒ Γ′;C

PROCEDURE-DECL except that ττ1 = ττ2 and C′ = ∅

Γ;Γ1 ⊢ call fDecl2(a1, a2, . . . , ak) ⇒ Γ′;C

PROCEDURE-DECL

Γ;Γ1 ⊢ procedure f(x1 : τ1, . . . , xk : τk) C′ ⇒ Γ′

Figure 5.2: Type Constraints Generation Rules
a mediator for transferring different information. $\Gamma_g$ and $\Gamma_l$ are used to record types for global and local variable respectively as specified in the rules $\text{Global-Var}$ and $\text{Local-Var}$. Type inference for both unary and binary expression are defined in rules $\text{Unary-Exp}$ and $\text{Binary-Exp}$.

In type constraint generation rules for statement $(\Gamma_g; \Gamma_l; p \vdash c \Rightarrow \Gamma'_g; \Gamma'_l; C)$, we define two different rules, $\text{Assign-Global-Update}$ and $\text{Assign-Local-Update}$, for assignments to global and local variables. The rule $\text{Assign-Global-Update}$ defines the type constraint generation for assignment to global variable: first, we fetch the type $\tau'$ of the target variable $x$ from $\Gamma_g$ and infer the type $T$ of the right hand side expression $e$, then, according to information security constraints, two types of constraints are generated: $T \leq \tau'$ and $p \leq \tau'$. It means that the security types of source information (either explicit or implicit) should be lower (or equal) than the security types of target information. Variable $p$ denotes the security type of the context where the assignment happens. For the local variable, its security type changes with each assignment of new values. Thus, in the rule $\text{Assign-Local-Update}$ for local assignment, a fresh type variable $\alpha$ for $x$ is generated with fresh function $\text{freshType}()$, followed by the generation of two type constraints for $\alpha$ and the update of $\Gamma_l$ with new type variable $\alpha$ for $x$. For example:

\begin{itemize}
  \item assignment to a global variable $g_0$, \\
  \item where $\Gamma_g(g_0) = \tau$, $\Gamma_g; \Gamma_l \vdash x : \tau_1$, $\Gamma_g; \Gamma_l \vdash y : \tau_2$
\end{itemize}

$$g_0 = x + y;$$

The produced type constraints for the above statement is $\{\tau_1 \sqcup \tau_2 \leq \tau, \; p \leq \tau\}$ with $p$ being the type of path condition (or context) leading to this statement execution. While for assignment to local variables, as shown in the following,

\begin{itemize}
  \item assignment to a local variable $l_0$, \\
  \item where $\Gamma_g; \Gamma_l \vdash x : \tau_1$, $\Gamma_g; \Gamma_l \vdash y : \tau_2$
\end{itemize}

$$l_0 = x + y;$$

first, we have to call $\text{freshType}()$ to generate a fresh type variable $\tau_{new}$, used for constraints generation of the above statement $\{\tau_1 \sqcup \tau_2 \leq \tau_{new}, \; p \leq \tau_{new}\}$, and then update $\Gamma_l$ by
associating \( x \) with \( \tau_{new} \), which reflects the new type of the content held in \( x \) after the assignment.

In the rule \( \mathit{iv} \) for conditional statement, executed in the context \( p \), if the type of conditional expression \( e \) is \( T \), then generate type constraints separately for \( c_1 \) and \( c_2 \) under context \( p \sqcup T \), finally merge the resulting constraints \( C' \) and \( C'' \) from both branches to get \( C''' \), for example:

\[
\text{if } x > 0 \text{ then}
\]
\[
y := m;
\]
\[
\text{else}
\]
\[
y := n;
\]
\[
\text{end if;}
\]

the type of conditional expression \( x > 0 \) is \( \tau_1 \sqcup \bot \). The type constraints generated along the true branch are \( C' = \{ \tau_2 \leq \tau_{new1}, p \sqcup (\tau_1 \sqcup \bot) \leq \tau_{new1} \} \), and the type constraints generated in the false branch are \( C'' = \{ \tau_3 \leq \tau_{new2}, p \sqcup (\tau_1 \sqcup \bot) \leq \tau_{new2} \} \), where \( \tau_{new1} \) and \( \tau_{new2} \) are two fresh type variables generated by function \( \mathit{freshType()} \). The final produced type constraints for conditional statement are \( C''' = C' \cup C'' \). The type mapping of local variables at the end of each branch are: \( \Gamma'_l = \Gamma_l[y \rightarrow \tau_{new1}] \) and \( \Gamma''_l = \Gamma_l[y \rightarrow \tau_{new2}] \). Thus, the final type mapping for local variables become \( \Gamma'''_l = \Gamma_l[y \rightarrow \tau_{new1} \sqcup \tau_{new2}] \), that’s \( \Gamma'''_l = \Gamma'_l \sqcup \Gamma''_l \).

For sequent statement, generate type constraints sequentially for \( c_1 \) and \( c_2 \), and use the union of their type constraints as its final constraint set, as specified in rule \( \mathit{SEQUENCE} \).

Rule \( \mathit{WHILE} \) defines the type constraint generation procedure for the while loop statement, where \( \mathit{ModVar}_{local}(c) \) calculates the set of local variables that maybe modified within the loop.
body $c$, $\Gamma_l$ and $\Gamma'_l$ are the type mapping for local variables before and after executing the while statement, and $\alpha_i$ is final fixed point type for the local variable $x_i$, $i \in [1, k]$. As illustrated in Figure 5.3 (a), the while loop is treated like a procedure call, all used variables, such as \{x_1, \ldots, x_k, y_1, \ldots, y_n\}, in while loop are serving as its interfaces/parameters: including both local variables and global variables. The types of those modified local variables, such as \{x_1, \ldots, x_k\}, will reach a fixed point \{\alpha_1, \ldots, \alpha_k\} after several loop iterations and the types of other variables, both global variables and those used but unmodified local variables, such as \{y_1, \ldots, y_n\}, will keep the same. Box 1 shows the types of used variables before entering while loop, where $\tau'_i$ is type for local variable $x_i$, $i \in [1, k]$, corresponding to $\Gamma_l(x_i)$ in the rule `While`. As the type of $y_i$ keeps the same during the while loop iteration, they are ignored to be tracked. Box 2 is fixed point type mapping for variables and it works as the interface of the while loop, with fresh type variable $\alpha_i$ being the fixed point type for $x_i$. Box 3 is type mapping for variables at the end of each loop iteration. In type constraint generation rule `While`, type constraint set $C_0$ corresponds to $\{\tau'_i \leq \alpha_i \mid 1 \leq i \leq k\}$ between type mapping of Box 1 and Box 2, similar to type constraints for passing in value from
arguments to parameters in a procedure call; $C'$ is type constraint set generated for while loop body $c$; and $C_1$ corresponds to type constraint set $\{\tau_i'' \leq \alpha_i \mid 1 \leq i \leq k\}$ between Box 3 and Box 2, similar to type constraints for a recursive call by passing the values at the end of each iteration to the interface of the loop for next round of iteration. Box 4 is fixed type mapping after the while loop, which is the same as Box 2. For example:

```plaintext
-- while loop,
-- where x, y, z are all local variables

while x > 0 loop
  z := x;
  x := y + z;
end loop;
```

assume that the types of $x$, $y$ and $z$ are $\tau_{x_0}$, $\tau_{y_0}$ and $\tau_{z_0}$ when reaching the while loop and $\alpha_x$, $\alpha_y$ and $\alpha_z$, which are final fixed point types, when exiting the while loop. Fixed point type is the least upper bound type for all possible values that can be held in a local variable modified within the while loop body. The type of conditional expression $x > 0$ based on fixed type of $x$ is $\alpha_x \sqcup \bot$ and it guarantees that the generated type constraints for the while loop body can capture its all possible information flows. The produced type constraints for loop body are $C' = \{\alpha_x \leq \alpha_{z_1}, \alpha_x \sqcup \bot \leq \alpha_{z_1}, \alpha_y \sqcup \alpha_{z_1} \leq \alpha_{x_1}, \alpha_x \sqcup \bot \leq \alpha_{x_1}\}$, where $\alpha_{z_1}$ is the fresh type of $z$ after the first assignment and $\alpha_{x_1}$ is the fresh type of $x$ after the second assignment. The type constraints $C_0 = \{\tau_{x_0} \leq \alpha_x, \tau_{y_0} \leq \alpha_y, \tau_{z_0} \leq \alpha_z\}$ and $C_1 = \{\alpha_{x_1} \leq \alpha_x, \alpha_y \leq \alpha_y, \alpha_{z_1} \leq \alpha_z\}$ can ensure that the fixed types of $x$, $y$ and $z$ are always the upper bound types for whatever number of loop iterations. The complete type constraints for the while statement are $C'' = C_0 \cup C_1 \cup C'$, and the final local type mapping $\Gamma'_l = \Gamma_l[x \rightarrow \alpha_x][y \rightarrow \alpha_y][z \rightarrow \alpha_z]$.

In the rule Procedure-Call for procedure call $f(a_1, a_2, \ldots, a_k)$, two kinds of constraints are generated: constraints between types of arguments and parameters $C_i$ with $i \in [1, k]$, and constraints between types of parameters enforced by the procedure itself $C'_f$, as illustrated
in Figure 5.3 (b) (c). Type constraint generation for procedure call is similar to the one for assignments between arguments and parameters depending on the in/out mode of the parameters. To avoid the mess up for the case of calling the same procedure in multiple places, fresh type variables are introduced for types of parameters and their type constraints are updated accordingly with the fresh type variables. $\Gamma_g(f)$ extracts the type constraints for the parameters of the called procedure $f$, where $\eta_i$ is the security type of the parameter $x_i$ and $C_f$ is the constraints between $\eta_i$, $i \in [1, k]$. If $x_i$ is an input (or output) variable, in In (or Out) mode, $\eta_i = \tau_i$, meaning the type of any input arguments (or output parameters) should be less or equal to (or greater or equal than) $\tau_i$; if $x_i$ is both an input and output variable, then $\eta_i$ is a pair $(\tau_{i1}, \tau_{i2})$, with $\tau_{i1}$ enforcing the type constraint for the input arguments passed into $x_i$ and $\tau_{i2}$ for the output arguments returned from $x_i$. The function $\lambda_1(\tau)$ produces a fresh type variable to replace $\tau$ if $\tau$ is a type variable, or returns $\tau$ if it’s a concrete type value, e.g. $\lambda_1(High)$ returns $High$, but $\lambda_1(\alpha)$ produces a fresh type variable $\alpha'$. In function $\lambda_2(x, \tau_x)$, $\tau_x$ is the type of $x$, after the assignment to $x$, it produces a fresh type variable if $x$ is a local variable, or returns $\tau_x$ if it’s a global variable.

In order to do modular analysis for procedure, the types of its parameters and the associated constraints enforced by the information flow within the procedure body are generated once and used as a summary for type constraints for arguments whenever the procedure is called. The rule $\text{Procedure-Call}$ specifies type constraints generation for a declared procedure $f(x_1, .., x_k)\{c\}$: first assign fresh type variable $\alpha_i$ for each parameter $x_i$, and then generate their type constraints $C$ according to the procedure body. The resulting constraints $C_f$ between types of parameters and global variables are stored as a summary for procedure $f$ in $\Gamma_g'$. The constraint operators $\text{Normalize}$ and $\text{Simplify}$ are used for generating atomic constraints, computing constraint closure and simplifying constraints, which will be discussed in subsection 5.2.2. The operator $\text{Reduce}$ removes all the type constraints except those about parameters and visible global variables. The parameters work as interface for receiving/returning the information from/to the external context, and their types and con-
straints are fixed when the procedure is called. Thus, the types of arguments should satisfy these constraints enforced on the parameters to guarantee the security of information flow within the procedure. For parameter \( x_i \), if it’s an input variable, it’s security type will always be \( \tau_i \) as it’s only readable; if it’s an output variable, it’s final security type is \( \Gamma'_l(x_i) \); and if it’s both an input and output variable, it’s type \( \eta_i \) is a pair \((\tau_i, \Gamma'_l(x_i))\) with \( \tau_i \) being the input type of \( x_i \) and \( \Gamma'_l(x_i) \) being the output type of \( x_i \). For example:

```
procedure double (x: in Integer; y: out Integer) is
begin
  y := 2 * x;
end double;
```

to generate type constraint summary for procedure `double`, first we assign two fresh type variables \( \alpha_1 \) and \( \alpha_2 \) for parameters \( x \) and \( y \), then generate their type constraints according to the information flow within the procedure body. In this example procedure, there is only one assignment that requires to enforce the constraints \( \{ \bot \sqcup \alpha_1 \leq \alpha_3 \} \), where \( \alpha_3 \) is a fresh type for \( y \). After performing the normalization and simplification procedure (to be discussed later), we get the following type constraint summary: `procedure double(x: \alpha_1, y: \alpha_3)` associated with constraints \( \alpha_1 \leq \alpha_3 \). It means that the parameter \( x \) accepts a value of type \( \alpha_1 \) and the parameter \( y \) outputs a value of type \( \alpha_3 \), and it holds that \( \alpha_1 \leq \alpha_3 \). The usage of the type constraint summary can be illustrated in the following example:

```
procedure calls ,
where \( \Gamma_g; \Gamma_l \vdash m : \tau_1, \Gamma_l \vdash u : \tau_2 \)
...
double (m, g_1);
double (u, g_2);
```

if \( g_1 \) and \( g_2 \) are global variables with types \( \tau_3 \) and \( \tau_4 \), then the type constraints generated for the first procedure call are \( \{ \tau_1 \leq \alpha'_1, \alpha'_3 \leq \tau_3, \alpha'_1 \leq \alpha'_3 \} \) and the type constraints for the
second procedure call are \( \{ \tau_2 \leq \alpha''_1, \alpha''_3 \leq \tau_4, \alpha''_1 \leq \alpha''_3 \} \). Each time when the procedure is called, all the type variables appearing in the procedure’s constraint summary are replaced with some fresh type variables, e.g., in the first procedure call for double, \( \alpha_1 \) and \( \alpha_3 \) are replaced with fresh type variables \( \alpha'_1 \) and \( \alpha'_3 \) (\( \alpha''_1 \) and \( \alpha''_3 \) in the second procedure call). It can distinguish the type constraints generated for different procedure calls and reflect the polymorphism of type constraints for the called procedure.

### 5.2.2 Type Constraint Simplification

The type constraints produced by constraint generator can be huge and most of them can be either simplified or optimized away. To achieve this, we have designed a set of simplification rules for constraints after the normalization operation.

**Normalization:** It’s a sequence of constraint transformations to finally get a set of atomic constraints. There are two types of constraints produced by the constraint generation rules, constraint for direct data flow from source information to target variable and the constraint for indirect control flow from the execution context to the enclosed execution statements. As we have discussed before, the target variable can be either global or local variables, and the type of global variable is fixed and the assignment to a local variable will always assign a fresh type variable to it, so the type of target variable is always \( \tau \), which is either some type variable or some concrete type value. Source information is expressed as an expression, whose type is T, having at most type operator \( \sqcup \), and the type of context is sequence of conditional expressions that are joined with \( \sqcup \) operator, so the final produced constraints can only be in the form of \( T \leq \tau \) that \( \sqcup \) is the only possible operation within \( T \). The following is the normalization procedure for produced constraints:

\[
C \cup \{ T_1 \sqcup T_2 \leq \tau \} \Rightarrow_{\text{norm}} C \cup \{ T_1 \leq \tau, T_2 \leq \tau \}
\]

It means that for any constraint in the form of \( T_1 \sqcup T_2 \leq \tau \), split it into two sub-constraints \( T_1 \leq \tau \) and \( T_2 \leq \tau \), and this procedure continues until all constraints are atomic.

In the previous section, we have given an example showing how to generate constraints for
a while loop, and here we will continue to show how to normalize the resulting constraints $C''$, which is equal to $C_0 \cup C_1 \cup C'$ and $C_0 = \{ \tau_{x_0} \leq \alpha_x, \tau_{y_0} \leq \alpha_y, \tau_{z_0} \leq \alpha_z \}$, $C_1 = \{ \alpha_{x_1} \leq \alpha_x, \alpha_y \leq \alpha_y, \alpha_{z_1} \leq \alpha_z \}$, and $C' = \{ \alpha_x \leq \alpha_{z_1}, \alpha_x \sqcup \bot \leq \alpha_{z_1}, \alpha_y \sqcup \alpha_{z_1} \leq \alpha_{x_1}, \alpha_x \sqcup \bot \leq \alpha_{x_1} \}$. Each constraint in $C_0$ and $C_1$ are already atomic, and only $C'$ needs to be normalized. According to the normalization rule, $\alpha_x \sqcup \bot \leq \alpha_{x_1}$ is normalized to $\{ \alpha_x \leq \alpha_{z_1}, \bot \leq \alpha_{x_1} \}$, and $\alpha_y \sqcup \alpha_{z_1} \leq \alpha_{x_1}$ is normalized to $\{ \alpha_y \leq \alpha_{x_1}, \alpha_{z_1} \leq \alpha_{x_1} \}$, and $\alpha_x \sqcup \bot \leq \alpha_{x_1}$ is normalized to $\{ \alpha_x \leq \alpha_{x_1}, \bot \leq \alpha_{x_1} \}$.

The normalization results of $C''$ is $C_\pi'' = \{ \tau_{x_0} \leq \alpha_x, \tau_{y_0} \leq \alpha_y, \tau_{z_0} \leq \alpha_z, \alpha_{x_1} \leq \alpha_x, \alpha_y \leq \alpha_y, \alpha_{z_1} \leq \alpha_z, \alpha_x \leq \alpha_{z_1}, \alpha_x \leq \alpha_{z_1}, \bot \leq \alpha_{z_1}, \alpha_y \leq \alpha_{x_1}, \alpha_{z_1} \leq \alpha_{x_1}, \alpha_x \leq \alpha_{x_1}, \bot \leq \alpha_{x_1} \}$.

**Simplification:** It’s a procedure to remove redundant constraints, optimize away the constraints that are always true, type inference and others, following the above constraint normalization, which is performed in two steps:

First, compute the closure for the constraint set according to the transitivity rule of ordering. In other words, if both $\tau_1 \leq \alpha$ and $\alpha \leq \tau_2$ are in constraint set, then add $\tau_1 \leq \tau_2$ into the constraint set too. This procedure is repeated until no new constraints are added, as shown in the following rule:

$C \cup \{ \tau_1 \leq \alpha, \alpha \leq \tau_2 \} \Rightarrow_{trans} C \cup \{ \tau_1 \leq \alpha, \alpha \leq \tau_2 \}$

Then, do simplification with the following rules.

1. $C \cup \{ \tau_1 \leq \alpha, \alpha \leq \tau_2 \} \Rightarrow_{simp} C \cup \{ \tau_1 \leq \tau_2 \}$
2. $C \cup \{ \bot \leq \tau \} \Rightarrow_{simp} C$  
3. $C \cup \{ \tau \leq \bot \} \Rightarrow_{simp} C$
4. $C \cup \{ \tau \leq \tau \} \Rightarrow_{simp} C$
5. $C \cup \{ \ell_1 \leq \ell_2 \} \Rightarrow_{simp} C$, if $\ell_1 \leq \ell_2$, otherwise report error
6. $C \cup \{ \alpha \leq \tau, \tau \leq \alpha \} \Rightarrow_{antisym} (C[\tau/\alpha], \Gamma_g[\tau/\alpha])$
7. $C \cup \{ \ell_1 \leq \alpha \} \cup \{ \ell_2 \leq \alpha \} \Rightarrow_{simp} (C[\tau/\alpha], \Gamma_g[\tau/\alpha]),$ if $\ell_1 \cup \ell_2 = \bot$
8. $C \cup \{ \alpha \leq \ell_1 \} \cup \{ \alpha \leq \ell_2 \} \Rightarrow_{simp} (C[\tau/\alpha], \Gamma_g[\tau/\alpha]),$ if $\ell_1 \cup \ell_2 = \bot$
9. $C \cup \{ \ell \leq \alpha \} \Rightarrow_{simp} (C[\tau/\alpha], \Gamma_g[\tau/\alpha]),$ if there is no $\ell'$ that $\ell \leq \ell'$
10. $C \cup \{ \alpha \leq \ell \} \Rightarrow_{simp} (C[\tau/\alpha], \Gamma_g[\tau/\alpha]),$ if there is no $\ell'$ that $\ell \leq \ell'$

Rule (1) removes the redundant constraint, and rules (2)-(4) optimize away the constraints that will always hold, and rule (5) reports error if the constraints enforced by the
security policy are violated. Rules (6)-(10) infer types for type variable in different cases, for example, in rule (7), if there are both $\ell_1 \leq \alpha$ and $\ell_2 \leq \alpha$ with condition $\ell_1 \sqcup \ell_2 = \top$ holds, then we can infer that $\top = \ell_1 \sqcup \ell_2 \leq \alpha$, thus $\alpha$ can only be $\top$. For the rule (9), if there exists constraint $\ell \leq \alpha$ and there is no $\ell'$ other than $\ell$ making $\ell \leq \ell'$ true, then $\alpha$ is $\ell$, for example, if $\ell$ is $\top$, then no type satisfying $\top \leq \alpha$ except $\top$ itself.

The final constraint set $C$ is a set of atomic constraints in the form of: $\alpha \leq \ell$, $\ell \leq \alpha$ or $\alpha_1 \leq \alpha_2$.

To continue on the while loop example, we will show how to apply the simplification rules to $C'''_n = \{\tau_{x_0} \leq \alpha_x, \tau_{y_0} \leq \alpha_y, \tau_{z_0} \leq \alpha_z, \alpha_{x_1} \leq \alpha_x, \alpha_y \leq \alpha_y, \alpha_{z_1} \leq \alpha_z, \alpha_x \leq \alpha_{z_1}, \alpha_x \leq \alpha_{z_1}, \bot \leq \alpha_{z_1}, \alpha_y \leq \alpha_{x_1}, \alpha_{z_1} \leq \alpha_{x_1}, \alpha_x \leq \alpha_{x_1}, \bot \leq \alpha_{x_1}\}$. To be simple, we skip the computation of the closure set for $C'''_n$. The resulting simplified constraint set $C'''_s = \{\tau_{x_0} \leq \alpha_x, \tau_{y_0} \leq \alpha_y, \tau_{z_0} \leq \alpha_z, \alpha_{x_1} \leq \alpha_x, \alpha_{z_1} \leq \alpha_z, \alpha_x \leq \alpha_{z_1}, \alpha_x \leq \alpha_{z_1}\}$ by removing $\bot \leq \alpha_{z_1}$ and $\bot \leq \alpha_{x_1}$ with rule (2), removing $\alpha_y \leq \alpha_y$ with rule (4), and $\alpha_x \leq \alpha_{z_1}$ with rule (1).

### 5.2.3 Type Constraint Satisfiability

There have been a lot of work done on the type interference with subtyping constraints. Some focus on the general algorithms for type inference, such as\textsuperscript{54,55,56}, and some study the inherent difficulty of the problem such as\textsuperscript{57,58}, which showed that the general constraint satisfaction problem is PSPACE-complete. Paper\textsuperscript{58} gives a sound decision procedure for checking satisfiability of the subtyping constraints, and proves that the satisfiability problem for subtyping constraints is solvable if the poset of atomic subtypings is a disjoint union of lattices. Since our goal is to check the security of the program with respect to the security policy, we only need to check the satisfiability of its type constraint set. In other words, the program is secure if its type constraints are solvable, otherwise, it’s insecure. In this work, we will use the checking algorithm proposed in\textsuperscript{58}. 
Theorem 1 (proved in\textsuperscript{58}). Subtyping constraint set $\mathcal{C}$ is satisfiable in $\mathcal{L}$ iff (1) it’s weakly satisfiable and (2) ground consistent, where $\mathcal{L}$ is a poset which is a disjoint union of lattices and $\mathcal{C}$ is a flat system of constraints.

$\mathcal{C}$ is said to be flat if each of its term is atomic constraint, which is satisfied for our generated type constraints after simplification. User defined partial order set of security types with $Ordered$ aspect are corresponding to $\mathcal{L}$, which is a disjoint union of lattices $\mathcal{L} = \mathcal{L}_1 \cup \ldots \cup \mathcal{L}_m$. Then, the satisfiability problem can be achieved by checking whether the following two conditions hold:

1. $\mathcal{C}$ is weakly satisfiable with respect to $\mathcal{L}$. $\mathcal{C} = \{\tau_1 \leq \tau'_1, \ldots, \tau_n \leq \tau'_n\}$ is weakly satisfiable if $\mathcal{C}_s = \{(\tau_1)_s = (\tau'_1)_s, \ldots, (\tau_n)_s = (\tau'_n)_s\}$ is satisfiable, which is a unification problem and thus can be decidable in polynomial time. $(\tau)_s$ is called a shape of $\tau$ and it’s defined as: (a) $(\ell)_s = \ast_i$ if $\ell \in \mathcal{L}_i$, (b) $(x)_s = x$, where $\ell$ denotes a concrete security type and $x$ is a type variable.

2. $\mathcal{C}$ is ground consistent with respect to $\mathcal{L}$. $\mathcal{C}$ is ground consistent if for all $\ell, \ell' \in \mathcal{L}$, $\ell \leq \ell'$ holds in $\mathcal{L}$ whenever $\mathcal{C} \vdash \ell \leq \ell'$, where $\ell \leq \ell'$ is either a term of $\mathcal{C}$ or can be computed from its transitive closure.

For example, if there are two ordered domains \(LOW\) and \(HIGH\) in the lattice $\mathcal{L}_1$, with $LOW \leq HIGH$, and there are two global variables $g_1$ and $g_2$ belonging to domains $LOW$ and $HIGH$ separately, as shown in the following:

\begin{verbatim}
type Domains is (LOW, HIGH) with Ordered;  --- LOW \leq HIGH
  g1: Integer with Domain ⇒ LOW;
  g2: Integer with Domain ⇒ HIGH;
procedure f1 (X: in Integer, Y: out Integer) is
begin
  Y := g1 + g2;
end
\end{verbatim}
\begin{verbatim}
g2 := X;
end f1;

procedure f2 (X: in Integer) is
begin
  X := g2;
  g1 := X;
end f2;
\end{verbatim}

For the procedure \textit{f1}, given the type variable \(\tau_1\) for \(X\) and \(\tau_2\) for \(Y\), we will have the constraint set \(\mathcal{C} = \{LOW \leq \tau_2, \text{HIGH} \leq \tau_2, \tau_1 \leq \text{HIGH}\}\) for the procedure by the type constraint generation rules. According to the above satisfiability conditions, (1) \(\mathcal{C}\) is weakly satisfiable as \(\mathcal{C}_* = \{(\text{Low})_* = (\tau_2)_*, (\text{HIGH})_* = (\tau_2)_*, (\tau_1)_* = (\text{HIGH})_*\}\), which can be simplified to \(\{*_1 = \tau_2, *_1 = \tau_2, \tau_1 = *_1\}\), and it’s satisfiable when both \(\tau_1\) and \(\tau_2\) are \(*_1\), which can be either \(LOW\) or \(HIGH\); and (2) \(\mathcal{C}\) is ground consistent with respect to \(\mathcal{L}_1\) as there is no \(\text{HIGH} \leq \text{LOW}\) in \(\mathcal{C}\) or its transitive closure, thus \(\mathcal{C}\) is satisfiable.

For the procedure \textit{f2}, given the type variable \(\tau\) for \(X\), we will have the constraint set \(\mathcal{C} = \{\text{HIGH} \leq \tau, \tau \leq \text{LOW}\}\) for the procedure. According to the above satisfiability conditions, (1) \(\mathcal{C}\) is weakly satisfiable as \(\mathcal{C}_* = \{(\text{HIGH})_* = (\tau)_*, (\tau)_* = (\text{LOW})_*\}\), which can be simplified to \(\{*_1 = \tau, \tau = *_1\}\), and it’s satisfiable when \(\tau\) is \(*_1\), which can be either \(LOW\) or \(HIGH\); but (2) \(\mathcal{C}\) is not ground consistent as there is \(\text{HIGH} \leq \text{LOW}\) in the transitive closure of \(\mathcal{C}\), thus \(\mathcal{C}\) is not satisfiable.

5.3 Example

In SPARK, a method can be defined with either \textit{procedure} or \textit{function}, where \textit{procedure} allows side-effects to externally visible variables while \textit{function} cannot.

Figure 5.4 (a) shows excerpts of a SPARK program with four global integer variables (\textit{Key}, \textit{SSN}, \textit{BankAccount}, \textit{Disk}) and three procedures (\textit{Encrypt}, \textit{Write}, \textit{Write_E}). Proce-
procedure Encrypt(K: in Integer; V: in Integer; R: out Integer) is

begin
    R := K * V; -- encryption simulation;
end Encrypt;

procedure Write() is

begin
    Encrypt(Key, SSN, Result);
    Disk := Result;
end Write;

procedure Write_E() is

begin
    Encrypt(Key, BankAccount, Result);
    Disk := Result;
end Write_E;

procedure Encrypt(K: in Integer; V: in Integer; R: out Integer) with

Declassifier,
    Domain -> (K => Secret, V => Secret, R => Public)

begin
    R := K * V; -- encryption simulation;
end Encrypt;

procedure Write() is

begin
    Encrypt(Key, SSN, Result);
    Disk := Result;
end Write;

procedure Write_E() is

begin
    Encrypt(Key, BankAccount, Result);
    Disk := Result;
end Write_E;

Figure 5.4: Illustration of Declassification Policy Framework

dure Encrypt simulates a simple encryption operation by encrypting value V with key K and putting the encrypted result in R. Procedure Write encrypts SSN and writes the result on Disk, and procedure Write_E puts the encrypted BankAccount on Disk.

The security requirements for the information of the system can be described as the following: all secret information should be encrypted before they are stored on disk, while top secrets should not allow to be stored on disk even after encryption. We assume that Key and SSN are classified as secret information, BankAccount as top secret, and Disk is accessible to the public. Furthermore, procedure Encrypt is assumed to perform our desired encryption function. Now the question is how can we verify that the program is information secure with respect to the above specified security requirements.

Figure 5.4 (b) shows the program specified with our proposed security policy framework to automatically enforce the above security requirements, where the code highlighted with green bars is the specification of the required information security contracts for the
program to be analyzed. In this paper, we propose to define security domains as an enumeration type annotated with a special aspect called *Ordered*. For example, the enumeration type *Domains* is defined as a type of security domains and the aspect *Ordered* by default means totally ordering between domain values: $\{\text{Public} \leq \text{Secret}, \text{Secret} \leq \text{TopSecret}, \text{Public} \leq \text{TopSecret}\}$, which means information of lower security domain is allowed to flow into destination of higher domains. Following the definition of the type of security domain are the associations of security domains with variables holding information of our interests.

The aspect *Domain* binds the security domain for the specified variable, for example, *Key* is associated with domain *Secret*. To specify the declassification policy, we introduce another new aspect called *Declassifier*, which means the declassification of information from one domain to another. *Encrypt* is specified with *Declassifier* aspect, with security domain association for each of its parameters. It means that *Encrypt* is a trusted declassifier function to perform declassification of information from *Secret* domain $(K, V)$ to *Public* domain $(R)$.

With user specified information contract, our last step is to check whether the program satisfies the declassification policy or not. First, we need to build the inter-procedure call graph, and then do static analysis for each procedure in a bottom-up way. In this example, *Encrypt* is called by both *Write* and *Write_E*, so we start with *Encrypt*. When we analyze *Encrypt*, we find that it’s a *Declassifier*, so we just skip it as it’s specified by the user as a trusted declassification function. Then we reach *Write* and get into its implementation body. On procedure call *Encrypt*, it will pass in the value of global variables *Key* and *SSN* to parameters $K$ and $V$, which would generate type constraints $\{\text{Secret} \leq \text{Secret}, \text{Secret} \leq \text{Secret}\}$ as required by the secure information flow rules that the security domain of source information should be less restrictive than the security domain of target information.

On the return of procedure call *Encrypt*, it will pass out the return value from parameter $R$ to local variable *Result*, which will produce type constraints $\{\text{Public} \leq \alpha\}$ as the local variable *Result* is defined without any security domain association so we assign it a fresh type variable $\alpha$. For assignment following the *Encrypt* call, the type constraint $\{\alpha \leq \text{Public}\}$
is generated. Finally, we get the constraints \(\{\text{Secret} \leq \text{Secret}, \text{Secret} \leq \text{Secret}, \text{Public} \leq \alpha, \alpha \leq \text{Public}\}\). Obviously, these constraints are satisfiable when \(\alpha\) takes the security domain \(\text{Public}\).

For procedure \(\text{Write}_E\), we can analyze it in the same way, and finally we will get the following constraints \(\{\text{Secret} \leq \text{Secret}, \text{TopSecret} \leq \text{Secret}, \text{Public} \leq \alpha, \alpha \leq \text{Public}\}\). As we can see, \(\text{TopSecret} \leq \text{Secret}\) violates the security requirement. So, from the above analysis, we can conclude that \(\text{Write}\) is secure while \(\text{Write}_E\) is not.

This section gives an example to show how we can use the security policy framework to automatically enforce the information security requirements. In the following sections, we will discuss in detail about our proposed security policy framework.

## 5.4 Soundness of Policy Enforcement

### 5.4.1 Operational Semantics

Usually, an information flow from \(u\) to \(v\) is said to be secure whenever \(\Gamma(u) \leq \Gamma(v)\) holds. But, the introduction of declassifier procedures makes it possible to allow information flow that doesn’t need strictly follow the condition \(\Gamma(u) \leq \Gamma(v)\) as long as the information flow goes through some trust classifiers and satisfies their constraints. A program is secure with respect to a declassification policy iff (1) either the program is a declassifier to be trusted, or (2) all information flow incurred in each transition of the program is secure.

To formally define the security of information flow and prove the soundness of the policy enforcement algorithm, we have given the formal semantics of the language with declassifier procedures, as shown in Figure 5.5.

The major difference of this semantics is the one for procedure call of declassifier, as shown in the rule \(\text{Declassify}_s\). For a declassifier procedure as specified by the user, as long as the security type of the arguments is compatible with the type of parameters, the information flow within the procedure is assumed to be secure. For example, for a declassifier procedure
Figure 5.5: Operational Semantics

to declassify information from High to Low, as long as the type of the information to be declassified is less than and equal to High, and the type of the resulting argument is greater than and equal to Low, then it’s a secure procedure call. So declassifier procedure is specified to be a trust procedure and work like a black box, we don’t need to track the information flow within it. Based on this assumption, we define the semantics for calling declassifier procedure as a sequence of assignments between arguments and some security domain source and destination. For any security domain $\tau$, we introduce two global variables $S_{\tau}$ (representing information source from domain $\tau$) and $D_{\tau}$ (representing information destination to domain $\tau$) such that $D_{\tau}$ is used to receive data from domain $\tau$ and $S_{\tau}$ is used to produces data of domain $\tau$. So, for any input argument $a_i$, if the type of its corresponding parameter is $\tau_i$, then we translate it into the assignment $D_{\tau_i} := a_i$; and for any output argument $a_j$, if the type of its corresponding parameter is $\tau_j$, then we translate it into the assignment $a_j := S_{\tau_j}$. In the rule $\text{Call}_{\text{as}}$, $\text{PassIn}(f(a_1, a_2, \ldots, a_k))$ passes in the arguments to the procedure $f$ by assigning $a_i$ to $x_i$ when $x_i$ is a parameter with in or in out mode, and $\text{PassOut}(f(a_1, a_2, \ldots, a_k))$ passes out the resulting values when the procedure call returns by assigning $x_j$ to $a_j$ when $x_j$ is a parameter with out or in out mode, where $i, j \in [1..k]$. 

\[
\begin{align*}
[n]_s & = n & \text{Lit}_s & \\
[x]_s & = s(x) & \text{Var}_s & \\
[e]_s & = v & \text{UNEXP}_s & \\
[e]_s & = \square v & \text{UNEXP}_s & \\
[e]_s & = {e_1, e_2} & \text{UNEXP}_s & \\
\end{align*}
\]

\[
\begin{align*}
[e]_s & = v & s' & = s[x \mapsto v] & \text{ASSIGN}_s & \\
[e]_s & = true & s' & = c_1 & \text{IFT}_s & \\
[e]_s & = false & s' & = c_2 & \text{IFF}_s & \\
[e]_s & = false & s' & = \text{WhileF}_s & \\
[e]_s & = true & s' & = \text{WhileT}_s & \\
\end{align*}
\]

\[
\begin{align*}
\text{procedure } f^{\text{Declassify}}(x_1 : \tau_1, \ldots, x_k : \tau_k) & \text{ with } \tau_i \text{ being specified security type of } x_i, i \in [1, k] \\
s[p(x_1, a_1); \ldots; p(x_k, a_k)] s' & = s \text{ Call}_s f(a_1, a_2, \ldots, a_k) s' \\
\end{align*}
\]

\[
\begin{align*}
\text{if } e \text{ then } s_1 \text{ else } s_2 & \text{ If}_s \\
\text{while } e \text{ do } c & \text{ WhileF}_s \\
\end{align*}
\]
### 5.4.2 Security of Information Flow

**τ-Equal (≡).** ∀ s t, s ≡^Γ τ t iff \([x]_s = [x]_t\) for all x that \(Γ(x) ≤ τ\).

For example, for two states s and t, with \(s = \{l : 1, h : 3\}\) and \(t = \{l : 1, h : 5\}\), if \(Γ(l) = LOW\) and \(Γ(h) = HIGH\) and \(LOW ≤ HIGH\), then we would say that \(s ≡^Γ LOW t\), but \(s ≠ Γ HIGH t\) as they don’t agree on the value of \(HIGH\) variable \(h\).

**Security (Γ ⊢ Secure(c)).** ∀ s s′ t t′, s [c] s′ and t [c] t′, ∀ τ ∈ security lattice, s′ ≡^Γ t′ whenever s ≡^Γ t.

The definition of the program security looks similar to the standard end-to-end noninterference property, but its based program semantics as shown in Figure 5.5 is different with standard semantics because of the introduction of declassification. For example,

- Assume two domains LOW and HIGH, with LOW ≤ HIGH, and
- \(Γ(l) = LOW\)
- \(Γ(time) = LOW\)
- \(Γ(h) = HIGH\)

```plaintext
v : Integer := h + time;
Encrypt(v, l);
```

the above program is intended to attach a \(time\) property to a \(HIGH\) information \(h\) before it’s encrypted and stored on a \(LOW\) variable \(l\). The \(Encrypt(x, y)\) is a declassifier function that makes it legal to transfer \(HIGH\) information (through parameter \(x\)) to \(LOW\) destination (through parameter \(y\)) after the encryption. Based on standard end-to-end noninterference property, the information flow within the program is insecure, which deviates from our expectation, as for any two initial states \(s\) and \(t\), whenever \(s ≡^Γ LOW t\), the conclusion \(s′ ≡^Γ LOW t′\) will not always hold for the final states \(s′\) and \(t′\) because the \(LOW\) variable \(l\) depends on the initial values of both \(LOW\) variable \(time\) and \(HIGH\) variable \(h\). While, in our new semantics for program with declassification, two global dummy variables are introduced for each domain, that’s \(S\) and \(D\) for \(LOW\) domain, and \(SH\) and \(DH\) for \(HIGH\) domain, and the program is transformed into the
following form:
\[ v : \text{Integer} := h + \text{time} ; \]
\[ \mathcal{D}_{\text{HIGH}} := v ; \]
\[ l := \mathcal{S}_{\text{LOW}} ; \]

The security requirement for assignment \( \mathcal{D}_{\text{HIGH}} := v \) guarantees the compatibility of the domain of \( v \) with the domain \( \text{HIGH} \) and enforces that the declassifier function \( \text{Encrypt} \) can only declassify information from \( \text{HIGH} \) compatible domains. Similarly, the assignment \( l := \mathcal{D}_{\text{LOW}} \) reflects the user’s view of information returned by \( \text{Encrypt} \), that’s, variable \( l \) receives some information from \( \text{LOW} \) domain that has been preprocessed (or declassified) from other domain. It does not matter how the declassification is processed within the declassifier function \( \text{Encrypt} \) as long as it’s trusted, and for the user of the declassifier function, he will be guaranteed to get a value that can be visible in \( \text{LOW} \) domain once he gives a value compatible with \( \text{HIGH} \) domain and there will be no information leak. So it make sense to do the program transformation for the procedure call of declassifier functions. Based on the semantics of the transformed program, we can prove the information flow security for the program: for any two initial states \( s \) and \( t \), whenever \( s \equiv_{\text{LOW}}^\Gamma t \), that’s \( [l]_s = [l]_t \land \text{time}_s = \text{time}_t \land [\mathcal{S}_{\text{LOW}}]_s = [\mathcal{S}_{\text{LOW}}]_t \), the conclusion \( s' \equiv_{\text{LOW}}^\Gamma t' \) will always hold for the final states \( s' \) and \( t' \), as \( l \) only depends on \( \mathcal{S}_{\text{LOW}} \) and \( \text{time} \) depends on itself.

5.4.3 Soundness of Policy Enforcement

Well-Typedness \((\Gamma \vdash \text{WellTyped}(s))\): \( \forall \Gamma s, \)

\[
(\forall x y_1 \ldots y_k, \\
\text{x in state } s \land x \text{ derives from } \{y_1, \ldots, y_k\} \rightarrow \\
\Gamma(y_1) \leq \Gamma(x) \land \ldots \land \Gamma(y_k) \leq \Gamma(x) \\
) \rightarrow \Gamma \vdash \text{WellTyped}(s).
\]

Well-typedness means that for any state variable \( x \) of a well typed state \( s \), the types of its derived values should be less than and equal to its type. In other words, for any state variable \( x \), there exists a function \( f_x \) such that the final value of \( x \) can be found as: \( f_x(y_1, \ldots, y_k) \), with the
condition \( \Gamma(y_i) \leq \Gamma(x) \), \( i \in [1..k] \), being satisfied.

**Lemma Well-Typedness-Preserve:** \( \forall \Gamma_g \Gamma_i \Gamma_i' C p c s s' \),

\[
\Gamma_g; \Gamma_i; p \vdash c \Rightarrow \Gamma_i' C \rightarrow C \text{ is satisfiable} \rightarrow
\]
\[
s \left[ c \right] s' \rightarrow \Gamma_g; \Gamma_i \vdash \text{WellTyped}(s) \rightarrow
\]
\[
\Gamma_g; \Gamma_i' \vdash \text{WellTyped}(s').
\]

For any well-formed program \( c \), with \( \Gamma_g; \Gamma_i; p \vdash c \Rightarrow \Gamma_i' C \), the program is secure with respect to the declassification policy if the constraints \( C \) is satisfiable.

**Theorem Soundness:** \( \forall \Gamma_g \Gamma_i \Gamma_i' C p c \),

\[
\Gamma_g; \Gamma_i; p \vdash c \Rightarrow \Gamma_i' C \rightarrow
\]
\[
C \text{ is satisfiable} \rightarrow
\]
\[
\Gamma_g \vdash \text{Secure}(c).
\]

With the lemma Well-Typedness-Preserve (to be proved later), the proof for the soundness theorem is straightforward: given assumptions \((A_1) \ \Gamma_g; \Gamma_i; p \vdash \Gamma_i' C \), with \((A_2) \ C \text{ being satisfiable, and (A3) } s \left[ c \right] s' \text{ and (A4) } t \left[ c \right] t', \text{ our goal is to prove that (Goal) for any security domain } \tau, \text{ if } s \equiv^\tau s \text{ then } s' \equiv^\tau s'.\) In our security policy enforcement framework, user has to specify the security domains for the global variables that are security critical, and make sure that information of different domains cannot interfere with each other unless through the user specified declassification procedures. At any initial state of the program \( c \), the value of any state variable \( x \) depends on itself, that’s \( f_x(x) \) and it holds that \( \Gamma(x) \leq \Gamma(x) \). Thus, we get \((A_5) \ \Gamma_g; \Gamma_i \vdash \text{WellTyped}(s) \) and \((A_6) \ \Gamma_g; \Gamma_i \vdash \text{WellTyped}(t) \). By applying the lemma Well-Typedness-Preserve on \((A_1) \ (A_2) \ (A_3)\) and \((A_5)\), we get \( \Gamma_g; \Gamma_i' \vdash \text{WellTyped}(s') \); similar application can lead to \( \Gamma_g; \Gamma_i' \vdash \text{WellTyped}(t') \).

In other words, for any global variable \( x \) of domain \( \tau \), where \( \tau = \Gamma_g(x) \), there exists a function \( f_x \) such that the final value of \( x \) (in both states \( s' \) and \( t' \)) at the end of the program execution can be represented as \( f_x(y_1, \ldots, y_k) \) and \( \Gamma_g(y_i) \leq \tau \) for \( i \in [1..k] \), thus \( \left[ x \right] s' = \left[ x \right] t' \) when \( s \equiv^\tau t \), and the conclusion for (Goal) is done.
Proof for Lemma Well-Typedness-Preserve. The lemma can be proved by a structural induction on $\Gamma_g; \Gamma_i; p \vdash c \Rightarrow \Gamma'_g; \Gamma'_i; C$ as shown in the below:

**case 1:** $c$ is $(x := e)$, and our goal is to prove that $\Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s')$. First, assume that $\Gamma_g; \Gamma_i; \vdash e : T$. If $x$ is a local variable, we will get $C = \{ T \leq \alpha, p \leq \alpha \}$ with $\alpha = \text{freshType()}$ and $\Gamma'_l = \Gamma_l[x \rightarrow \alpha]$. As $\Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s)$, it can be inferred that, for any used variable $x_e$ in $e$ with type $\tau_{x_e}$, it holds that $\tau_{x_e} \leq T$, and there exists a function $f_{x_e}$ such that the value of $x_e$ can be found as $f_{x_e}(y_1, \ldots, y_m)$ with $\tau_{y_i} \leq \tau_{x_e}$ and $\tau_{y_i}$ being the type of $y_i$, thus $\tau_{y_i} \leq T$ for $i \in [1..m]$. Similarly, for any variable $x_e$ in conditions leading to the assignment, it holds that $\tau_{x_e} \leq p$, with $\tau_{x_e}$ being the type of $x_e$ and $p$ being the type of conditions, and there exists a function $f_{x_e}$ such that the value of $x_e$ can be found as $f_{x_e}(z_1, \ldots, z_n)$ with $\tau_{z_i} \leq \tau_{x_e}$ and $\tau_{z_i}$ being the type of $z_i$, thus $\tau_{z_i} \leq p$ for $i \in [1..n]$. With constraint $C$ being true, we can infer that $\tau_{y_i} \leq \alpha$ from $\tau_{y_i} \leq T$ and $T \leq \alpha$ for $i \in [1..m]$, and $\tau_{z_i} \leq \alpha$ from $\tau_{z_i} \leq p$ and $p \leq \alpha$ for $i \in [1..n]$. As $x$ is data-depending on $y_i$, with $\tau_{y_i} \leq \alpha$ for $i \in [1..m]$ and control-depending on $z_i$, with $\tau_{z_i} \leq \alpha$ for $i \in [1..n]$, we get that $x$ is well-typed in state $s'$ after the assignment and the well-typedness for other variables will carry over from $s$ to $s'$, thus we prove that $s'$ is well-typed.

If $x$ is a global variable and $\Gamma_g(x) = \tau$, we will get $C = \{ T \leq \tau, p \leq \tau \}$ and $\Gamma'_l = \Gamma_l$. Similar to the proof of the assignment to local variables, as $\Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s)$ and in the assignment, $x$ is updated with the value $e$ under some condition of type $p$, the truth of the constraints $C$ can guarantee that the type of the source value domain for the updated $x$ in $s'$ is $\leq \tau$, thus $\Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s')$ holds.

**case 2:** $c$ is $(\text{if } e \text{ then } c_1 \text{ else } c_2)$. As an inductive case, it introduces two induction hypotheses: $IH1 : \forall s s' , C_1 \text{ is satisfiable } \Rightarrow s \llbracket c_1 \rrbracket s' \Rightarrow \Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s) \Rightarrow \Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s')$, and $IH2 : \forall s s' , C_2 \text{ is satisfiable } \Rightarrow s \llbracket c_2 \rrbracket s' \Rightarrow \Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s) \Rightarrow \Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s')$, with $\Gamma_g; \Gamma_i; p \sqcup T \vdash c_1 \Rightarrow \Gamma_{l_1}; C_1$ and $\Gamma_g; \Gamma_i; p \sqcup T \vdash c_2 \Rightarrow \Gamma_{l_2}; C_2$ and $\Gamma_g; \Gamma_i; \vdash e : T$. The assumption is that constraints $C$ is satisfiable, with $C = C_1 \sqcup C_2$.

If $[e]_s = \text{true}$, then $s \llbracket \text{if } e \text{ then } c_1 \text{ else } c_2 \rrbracket s' = s \llbracket c_1 \rrbracket s'$. $C_1$ is satisfiable as it’s a subset of satisfiable $C$, by applying the induction hypothesis $IH1$, we get $\Gamma_g; \Gamma_i; \vdash \text{WellTyped}(s')$.

If $[e]_s = \text{false}$, then $s \llbracket \text{if } e \text{ then } c_1 \text{ else } c_2 \rrbracket s' = s \llbracket c_2 \rrbracket s'$. $C_2$ is satisfiable as it’s a subset of
satisfiable \( C \), by applying the induction hypothesis \( IH2 \), we get \( \Gamma_g; \Gamma_l \vdash WellTyped(s') \).

Now we have proved that at the end of each branch \( s' \) is well-typed, and our goal is to prove \( \Gamma_g; \Gamma'_l \vdash WellTyped(s') \) at the merging point of the two branches, where \( \Gamma'_l = \Gamma_l \cup \Gamma_{l_2} \). For any variable \( x \) in \( s' \), if it’s a global variable, we can conclude that there exists a function \( f_x \) such that the value of \( x \) can be found as \( f_x(y_{t1}, \ldots, y_{tm}) \), with \( \tau_y \leq \tau_x \) for \( i \in [1..m] \), in the true branch and there exists a function \( f_{x_f} \) such that the value of \( x \) can be found as \( f_{x_f}(y_{t1}, \ldots, y_{fn}) \), with \( \tau_y \leq \tau_x \) for \( i \in [1..n] \), in the false branch. So, there should exists a function \( f_x \) such that the value of \( x \) at the end of the conditional statement can be found as \( f_x(y_{t1}, \ldots, y_{tm}, y_{f1}, \ldots, y_{fn}) \), with \( \tau_y \leq \tau_x \) for \( i \in [1..m] \) and \( \tau_y \leq \tau_x \) for \( i \in [1..n] \). Similarly, if it’s a local variable, the type of its source value domain is \( \leq \Gamma_l(x) \) in true branch and \( \leq \Gamma_{l_2}(x) \) in false branch, so it’s \( \leq \Gamma'_l(x) \) as \( \Gamma'_l(x) = \Gamma_l(x) \cup \Gamma_{l_2}(x) \). Finally, we reach that \( \Gamma_g; \Gamma'_l \vdash WellTyped(s') \).

case 3: \( c \) is \((\textbf{while } e \textbf{ do } c_1)\). As an inductive case, it introduces the induction hypothesis: \( IH : \forall s \ s', C_1 is satisfiable \rightarrow s [c_1] s' \vdash \Gamma_g; \Gamma'_l \vdash WellTyped(s) \rightarrow \Gamma_g; \Gamma_l \vdash WellTyped(s') \), with \( \Gamma_g; \Gamma'_l; p \cup T \vdash c_1 \Rightarrow \Gamma_l; C_1 \) and \( \Gamma'_l = \Gamma_l[x_i \rightarrow \alpha_i][\ldots][x_k \rightarrow \alpha_k] \), where \( \alpha_i \) is the fixed-point type of local variable \( x_i \) that maybe modified within loop body, and \( \Gamma_g; \Gamma'_l \vdash e : T \).

If \( \llbracket e \rrbracket_s = false \), then \( s' = s \), thus \( \Gamma_g; \Gamma_l \vdash WellTyped(s') \) whenever \( \Gamma_g; \Gamma_l \vdash WellTyped(s) \). Our goal is to prove \( \Gamma_g; \Gamma'_l \vdash WellTyped(s') \). For any local variable \( x_i \) that maybe modified within loop body, it holds that \( \Gamma_l(x_i) \leq \Gamma'_l(x_i) \) as generated by the constraint generation rule \( \textbf{while} \), and there exists a function \( f_{x_i} \) such that the value of \( x_i \) in state \( s' \) can be found as \( f_{x_i}(y_{t1}, \ldots, y_{tn}) \), with \( \tau_{y_j} \leq \tau_{x_i} = \Gamma_l(x_i) \) for \( j \in [1..n] \) because of \( \Gamma_g; \Gamma_l \vdash WellTyped(s') \), by transitive relation of \( \leq \) ordering, we get \( \tau_{y_j} \leq \Gamma'_l(x_i) \) for \( j \in [1..n] \). For any global variable, its type keeps the same after the loop iteration, so in state \( s' \), if it’s well typed with respect to \( \Gamma_g; \Gamma_l \) then it should also be well-typed with respect to \( \Gamma_g; \Gamma'_l \). As both local and global variables are well-typed in state \( s' \) with respect to \( \Gamma_g; \Gamma'_l \), it’s natural to get that \( \Gamma_g; \Gamma'_l \vdash WellTyped(s') \).

If \( \llbracket e \rrbracket_s = true \), then \( s [\textbf{while } e \textbf{ do } c_1 \] \) \( s' \) is the same as \( s [c; \textbf{while } e \textbf{ do } c_1 \] \( s' \), and there should exist a state \( s_t \) such that \( s [c] s_t \) and \( s_t [\textbf{while } e \textbf{ do } c_1 \] \) \( s' \). \( C_1 \) is a subset of \( C \), if \( C \) is satisfiable, then \( C_l \) is satisfiable too. In the case of \( \llbracket e \rrbracket_s \) being false, we have proved that \( \Gamma_g; \Gamma'_l \vdash WellTyped(s) \) whenever \( \Gamma_g; \Gamma_l \vdash WellTyped(s) \). By applying the induction hypothesis \( IH \),
we get $\Gamma_g; \Gamma_l \vdash \text{WellTyped}(s_1)$. For any local variable $x_i$ that maybe modified within loop body, it holds that $\Gamma_l(x_i) \leq \Gamma'_l(x_i)$ as shown in the constraint generation rule \texttt{While}, and there exists a function $f_{x_i}$ such that the value of $x_i$ in state $s_1$ can be found as $f_{x_i}(y_1, \ldots, y_n)$, with $\tau_{y_j} \leq \tau_{x_i} = \Gamma_l(x_i)$ for $j \in [1..n]$ because of $\Gamma_g; \Gamma_l \vdash \text{WellTyped}(s_1)$, by transitive relation of $\leq$ ordering, we get $\tau_{y_j} \leq \Gamma'_l(x_i)$ for $j \in [1..n]$. So, after one iteration of while loop, the state $s_1$ is still well-typed for any local variable $x_i$ with respect to $\Gamma_g; \Gamma_l$. For the global variables, if they are well-typed with respect to $\Gamma_g; \Gamma_l$, then they are also well-typed with respect to $\Gamma_g; \Gamma'_l$, as the type of the global variables are stored in $\Gamma_g$, which is kept the same. So we can infer that $\Gamma_g; \Gamma'_l \vdash \text{WellTyped}(s_1)$ whenever $\Gamma_g; \Gamma_l \vdash \text{WellTyped}(s_1)$. Now we have reached the invariant that $\forall \ t \ t', \ t \llbracket c_1 \rrbracket t'$ if $\Gamma_g; \Gamma'_l \vdash \text{WellTyped}(t)$ then $\Gamma_g; \Gamma_l \vdash \text{WellTyped}(t')$, which guarantees the well-typedness for loop ending states after any number of iterations.

case 4: $c$ is $(c_1; \ c_2)$. As an inductive case, it introduces two induction hypothesis:

$IH1 : \forall s \ s', C_1 \ is \ satisfiable \rightarrow s \llbracket c_1 \rrbracket s' \rightarrow \Gamma_g; \Gamma_l \vdash \text{WellTyped}(s) \rightarrow \Gamma_g; \Gamma_l \vdash \text{WellTyped}(s')$, and

$IH2 : \forall s \ s', C_2 \ is \ satisfiable \rightarrow s \llbracket c_2 \rrbracket s' \rightarrow \Gamma_g; \Gamma_l \vdash \text{WellTyped}(s) \rightarrow \Gamma_g; \Gamma_l \vdash \text{WellTyped}(s')$, with

$\Gamma_g; \Gamma_l; p \vdash c_1 \Rightarrow \Gamma_l; C_1$ and $\Gamma_g; \Gamma_l; p \vdash c_2 \Rightarrow \Gamma'_l; C_2$.

As $C = C_1 \cup C_2$, if $C$ is satisfiable, then both $C_1$ and $C_2$ are also satisfiable. Given $s \llbracket c_1; \ c_2 \rrbracket s'$, there should exist $s_1$ such that $s \llbracket c_1 \rrbracket s_1$ and $s_1 \llbracket c_2 \rrbracket s'$. We can get $\Gamma_g; \Gamma_l \vdash \text{WellTyped}(s_1)$ by applying the induction hypothesis $IH1$. Furthermore, by applying the induction hypothesis $IH2$, we will get $\Gamma_g; \Gamma'_l \vdash \text{WellTyped}(s')$ and the proof is done.

case 5: $c$ is $(\textbf{call } f^{\text{Declassify}}(a_1, \ldots, a_k))$. Our goal is to prove $\Gamma_g; \Gamma'_l \vdash \text{WellTyped}(s')$.

For any input argument $a_i$, according to the semantics of procedure call for declassification, we have $D_{\tau_i} := a_i$, with $\tau_i$ being the domain type of the corresponding parameter and $D_{\tau_i}$ being a global variable of $\tau_i$. According to the constraint generation rule \texttt{Procedure-Call}, it will generate type constraint $C_i = \{\tau_{a_i} \leq \tau_i, p \leq \tau_i\}$, where $\tau_{a_i}$ is type of $a_i$. Similar to the proof for assignment, $\forall \Gamma_g; \Gamma_l \ s \ s', \text{if } \Gamma_g; \Gamma_l \vdash \text{WellTyped}(s) \text{ and } s \llbracket D_{\tau_i} := a_i \rrbracket s'$, then $\Gamma_g; \Gamma_l \vdash \text{WellTyped}(s')$ as long as the constraints $C_i$ is satisfiable.

For any output argument $a_j$, we get $a_j := S_{\tau_j}$, with $\tau_j$ being the domain type of the corresponding parameter and $S_{\tau_j}$ being a global variable of $\tau_j$. In constraint generation rule \texttt{Procedure-Call,
if $a_j$ is a local variable, a fresh type variable $\tau'_j = freshType()$ is generated for $a_j$ and $\Gamma_I[a_j \rightarrow \tau'_j]$, with type constraints $C_j = \{\tau_j \leq \tau'_j, p \leq \tau'_j\}$. Similar to the proof for assignment, $\forall \; \Gamma_g; \Gamma_I \vdash WellTyped(s)$ and $s \llbracket a_j := S, \tau_j \rrbracket s'$, then $\Gamma_g; \Gamma_I[a_j \rightarrow \tau'_j] \vdash WellTyped(s')$ as long as the constraints $C_j$ is satisfiable. If $a_j$ is a global variable, then the type constraints $C_j = \{\tau_j \leq \Gamma_g(a_j)\}$, $p \leq \Gamma_g(a_j)$, and the proof for preserving well-typedness can be established in a similar way.

So, given the initial state $s$ before the procedure call such that $\Gamma_g; \Gamma_I \vdash WellTyped(s)$, as both passing in and passing out the values will keep the well-typedness of states, we can finally get $\Gamma_g; \Gamma'_I \vdash WellTyped(s')$.

**case 6: $c$ is (call $f(a_1, \ldots, a_k)$).** $C_f$ is the type constraints between parameter types of procedure $f$ and it’s produced by rule $\text{PROCEDURE-DECL}$ for $\Gamma_g; \Gamma_I; \vdash \text{procedure } f(x_1, \ldots, x_k)\{c_1\} \Rightarrow \Gamma'_g$. Inductively, we have $IH: \forall \; s \; s', \; C_1 \text{ is satisfiable } \rightarrow \; s \llbracket c_1 \rrbracket s' \rightarrow \; \Gamma_g; \Gamma_{l_1} \vdash WellTyped(s) \rightarrow \; \Gamma_g; \Gamma_{l_2} \vdash WellTyped(s')$, with $\Gamma_g; \Gamma_{l_1}; \vdash \; c_1 \Rightarrow \Gamma_{l_2}; C_1$ and $\Gamma_{l_1} = \Gamma_I[x_1 \rightarrow \alpha_1][\ldots][x_k \rightarrow \alpha_k]$, where $\alpha_i = freshType(), \; i \in [1, k]$. In the rule $\text{PROCEDURE-CALL}$ for (call $f(a_1, \ldots, a_k)$), each parameter type is renamed with some fresh type variable and $C_f$ is updated accordingly. As the renaming will not affect the type constraints and its satisfiability, so it’s a safe action. In our proof here, we can assume no renaming for parameter types now.

Given $s \llbracket \text{call } f(a_1, \ldots, a_k) \rrbracket s'$, according to the operational semantics for procedure call, there exist $s_1$ and $s_2$ such that $s \; \text{PassIn}(f(a_1, \ldots, a_k)) \; s_1, \; s_1 \llbracket c_1 \rrbracket s_2$ and $s_2 \; \text{PassOut}(f(a_1, \ldots, a_k)) \; s'$.

For any input argument $a_i$, with type $\tau_{a_i}$, there are type constraints $\{\tau_{a_i} \leq \alpha_i, \; p \leq \alpha_i\}$ being satisfiable. And it’s easy to prove that if $\Gamma_g; \Gamma_I \vdash WellTyped(s)$ then $\Gamma_g; \Gamma_{l_1} \vdash WellTyped(s_1)$.

$C_f$ is a subset of $C$, if $C$ is satisfiable then $C_f$ is also satisfiable, and $\Gamma_g; \Gamma_{l_2} \vdash WellTyped(s_2)$ can be inferred by applying the induction hypothesis $IH$.

For any output argument $a_j$, the type constraints are generated in the same way as for assignment $a_j := x_j$, similar to the proof for assignment, if $\Gamma_g; \Gamma_{l_2} \vdash WellTyped(s_2)$, then the well-typedness after the copying out operation will still hold. Finally, we get $\Gamma_g; \Gamma'_I \vdash WellTyped(s')$.  

98
Table 5.1: Experiment Data

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Constraints#</th>
<th>Simp_Constraints#</th>
<th>SAT</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>if_stmt</td>
<td>28</td>
<td>17</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>if_stmt_implicit_flow</td>
<td>12</td>
<td>6</td>
<td>False</td>
<td>1</td>
</tr>
<tr>
<td>mailbox</td>
<td>29</td>
<td>19</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>encrypt_system</td>
<td>17</td>
<td>5</td>
<td>False</td>
<td>1</td>
</tr>
<tr>
<td>password_check_system</td>
<td>19</td>
<td>8</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>procedure_call</td>
<td>91</td>
<td>57</td>
<td>False</td>
<td>1</td>
</tr>
<tr>
<td>procedure_call2</td>
<td>15</td>
<td>4</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>procedure_call3</td>
<td>20</td>
<td>6</td>
<td>False</td>
<td>2</td>
</tr>
<tr>
<td>seq_stmt</td>
<td>8</td>
<td>3</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>while_stmt</td>
<td>106</td>
<td>89</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>decryption_good</td>
<td>133</td>
<td>107</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>decryption_bad</td>
<td>125</td>
<td>101</td>
<td>False</td>
<td>1</td>
</tr>
<tr>
<td>thumper_good</td>
<td>69</td>
<td>41</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>thumper_bad</td>
<td>114</td>
<td>80</td>
<td>False</td>
<td>1</td>
</tr>
</tbody>
</table>

5.5 Evaluation

We have implemented an automatic verification tool as an associated part of the proposed declassification policy framework to automatically check whether the program conforms to the specified security policy. The evaluation is performed on a collection of small representative examples of SPARK programs. The analysis result for each example program is displayed in Table 5.1.

For each of these examples, Constraint# in Table 5.1 denotes the number of type constraints generated for each program. Simp_Constraint# gives the number of type constraints after simplifications, SAT shows the satisfiability of the constraints, and Bugs gives the number of information flow errors that violates user specified information security policy.

From the Table 5.1, we can see that constraint simplification rules can effectively reduce the number of type constraints to be checked. For the program if_stmt_implicit_flow, the detected information security violation is caused by the implicit control flow from secret information to public information. In program encrypt_system, the encryption function, which is designed to declassify information from secret domain to public domain, is used as an information laundering channel to declassify information from top secret to public. That’s the reason why the program fails the security policy requirement. In program procedure_call, the encryption function is wrapped within another function, and it’s used to declassify information with higher security level than it’s intended security level and the violation of the security policy is reported. In program procedure_call2, a declassification function called Filter_And_Clean is specified for declassifying information from top
secret to public and it demonstrates that it would also be fine to use the same declassification function to declassify the secret information that are less restricted than top secret. In program \textit{procedure\_call3}, a top secret information is allowed to be declassified to the public only if it goes through two separate specific declassification operations, one is for declassification from top secret to secret and another is for declassification from secret to public. There are two information flow within \textit{procedure\_call3} that performs information declassification without following the required security policy.

The \textit{decryption} example, which is adapted from paper\textsuperscript{59}, is shown in the following:

\begin{verbatim}
−− key, msg: secret information
−− cipher, result: public information
msg := cipher * key;
Padding(msg, paddingOk);
if (paddingOk) then
    CheckSum(msg, checkSum);
    if (checkSum /= −1) then
        result := true;
    else
        result := false;
    end if;
else
    result := false;
end if;
procedure Padding(V: in Integer; R: out Boolean) is
begin
    R := (V \text{mod} 256 = 0);
end Padding;
procedure CheckSum(V: in Integer; R: out Integer) is
\end{verbatim}
begin
R := (V / 256) mod 256;
end CheckSum;

It decrypts a ciphertext (public) with a decryption key (secret) and returns the result (public) showing whether the decryption operation is successful or not. It returns true if the cyphertext has a valid padding and passes the integrity check, otherwise, return false. In our security policy framework, we define two security domains: \{public, secret\} and public ≤ secret, with key and msg annotated with secret domain, cipher and result in public domain. Both procedures Padding and CheckSum are specified as declassification functions to be allowed to declassify information from secret domain to public domain. The policy checker will not report any information security bugs for this program as its generated type constraints are satisfiable. However, if there is a mistake within the implementation, e.g.

\[
\textbf{if} \ (\text{key} \neq -1) \ \textbf{then} \quad \textit{use key instead of checkSum}
\]

\[
\text{result} := \text{true};
\]

\[
\textbf{else}
\]

\[
\text{result} := \text{false};
\]

\[
\textbf{end if};
\]

then the policy checker will report a bug for information leak from key (secret) to result (public).

The thumper example is a prototype implementation for a time stamp protocol in SPARK programming language. It’s still an ongoing project with a lot of major components missing. The working mechanism of the protocol is that: Alice needs a certificate or signature to prove later that one of her documents existed at (or before) the current time. She first computes a cryptographic hash of the document using a suitable secure hash algorithm. She then presents this hash to a trusted third party time stamping server. The server appends the current time to the end of the hash, signs the resulting data to make the official time stamp, and returns the time stamp to Alice. One of the major concerns for this example is to make sure the absence of information leak for the signature key. In other words, the server only uses the key for signature and all other information
flow from key is illegal. The following shows part of the implementation code:

```plaintext
--- server side ---

Key: Integer;
Time: Integer;

procedure Tick (Time: in out Integer) is
begin
  Time := Time + 1; -- simulate the clock;
end Tick;

procedure AppendTime(U: in Integer; V: out Integer)
  is begin
    Tick(Time);
    V := U + Time; -- simulate to append time to U
end AppendTime;

procedure Sign(Data: in Integer; Signature: out Integer)
  is begin
    Signature := Data * Key; -- simulate the signature
end Sign;

procedure MakeSignature(Data: in Integer;
  Signature: out Integer) is
  DataWithTime: Integer;
begin
  AppendTime(Data, DataWithTime);
  Sign(DataWithTime, Signature);
end MakeSignature;

procedure Hash(X: in Integer; Y: out Integer) is
begin
```

102
\[ Y := X \mod 100 \; \text{--- simulate the hash function} \]

\textbf{end} Hash;

\textbf{--- client side ---}

Document: Integer;
DocSignature: Integer;

\textbf{--- the mediator between client and server ---}

\textbf{procedure} Mediator \textbf{is}
  
  HashValue: Integer;

begin
  \textbf{--- receives the document from the client,}
  \textbf{--- and returns back the signature for it}
  Hash\text{\textup{ }}(\text{Document, HashValue});
  
  MakeSignature\text{\textup{ }}(\text{HashValue, DocSignature});

\textbf{end} Mediator;

On the server side, it has a secret \texttt{Key} for signature and a timer \texttt{Time} to store the current time. The function \texttt{Tick} is to simulate the clock, \texttt{AppendTime} is to append the current time to a signed document, \texttt{Sign} is to generate signature and \texttt{Hash} to produce hash code. On the client side, it has a \texttt{Document} needed to be signed and stored at \texttt{DocSignature} to be visible to the public. If the program is correct, then there should be no information leak from the secret \texttt{Key} to the Public \texttt{DocSignature}. Similar to the previous example, we define two security domains: \{\texttt{public, secret}\} and \texttt{public} \leq \texttt{secret}, with \texttt{Key} in \texttt{secret} domain, \texttt{DocSignature} in \texttt{public} domain. The procedure \texttt{Sign} is specified as declassification functions to be allowed to declassify information from \texttt{secret} domain to \texttt{public} domain. There is no bug reported by the policy checker as its generated type constraints are satisfiable. If we make an intentional information leak from \texttt{Key} to \texttt{DocSignature},
then it will be detected and reported as a bug.

**Further Assessment:** The current evaluation of the proposed security policy framework is based on a collection of small examples that we have collected and adapted from various examples in related research papers. There are many reasons for why it’s difficult to find bigger examples. Firstly, the SPARK programming language that we are working on is mainly used for the development of safety-critical systems, including commercial aviation, medical and space applications, it’s so safety-critical that their source codes are usually unavailable to the public. Secondly, in the current research of the declassification policy, they mainly focus on the development of different semantic principles for declassification mechanism and their soundness proof, thus leads to a variety of definition of security (from different perspectives of what, when, who and where dimensions). But, as far as we know, none of these work have ever been used in any real applications to report any real defects. Usually, the validity of their proposed methods are proved to be correct based on their proposed semantic principles and illustrated in small examples. Thirdly, the declassification policy is a language-based security policy, which requires the extension of the existing programming languages to syntactically support the declassification policy. That’s another major obstacle to apply the declassification policy in much bigger examples. The other reasons include: most of the current research are working on a toy language and the scalability of the proposed policy and its enforcement to a more complex language would be another challenge.

Since our proposed declassification policy is designed for SPARK programming language, and we have close cooperation with the SPARK designer company, which makes it possible for us do some further assessment on more bigger and practical examples in the future.
Chapter 6

Conclusion

Creating a highly reliable and robust software is a long existing challenge in the domain of safety- and security-critical applications. It’s built based on formal verification technique to ensure that the expected behavior of the program conforms to the software-contract-based specifications. The contributions of this thesis are to make progress towards this direction to increase our confidence in the correctness of high integrity applications from two different perspectives: to guarantee the absence of run-time errors and to ensure the security of information flow.

We have formalized the dynamic/evaluation semantics for a significant subset of the high integrity language SPARK 2014 (which includes run-time checks as an integral part of the language) using the Coq proof assistant, and we have illustrated how this formal semantics can be used in a mechanized proof infrastructure to check that ASTs produced by the GNAT compiler frontend having correctly incorporated decorations for run-time checks. This included developing an optimizer with mechanized proofs of correctness that achieves run-time check placement optimizations equal to or better than GNAT. As the compiler and analyzers all share the AST produced by the frontend, with decorations indicating where run-time checks should be inserted, the certified AST increases the confidence in the GNAT compiler back-end that embeds run-time assertion checking when it emits machine code for testing, as well as in the GNATprove verifier that uses the run-time check decorations to determine what verification conditions to generate. The effectiveness of the
approach was demonstrated using programs from AdaCore test suites.

To control the information flow between different security domains and ensure the information security with respect to security requirements, we have proposed a language-based information security policy specification framework and the associated checking algorithm based on typing system to automatically check the information security of the program. Furthermore, we have demonstrated how the proposed security policy framework can be integrated into SPARK 2014 programming language with the introduction of some new security aspects, and we have implemented a prototype tool to show both the expressiveness of the policy language and the effectiveness of the checking algorithm for automatic enforcement of the security policy.

6.1 Future Work

The work in this thesis opens up a number of interesting opportunities for future work. Some of them are shown as follows:

- Towards the formalization of language semantics for complete SPARK 2014. Now we have formalized the semantics for a core subset of SPARK with support of three major categories of run-time checks. In the future, we are interested in adding more SPARK language features to support more run-time check certifications in a similar way as done in this thesis. Our final goal is to define the formal semantics for the complete language of SPARK 2014 as it’s defined in SPARK reference manual. The formal and unambiguous semantics is beneficial for machine-verified proof of correctness of SPARK static analysis and translation tools.

- Certified SPARK frontend for CompCert certified compiler framework. The mechanized SPARK 2014 semantics (which was designed to align with the approach of CompCert) along with the Jago translator provides the foundation for producing a mechanically proved translation from SPARK into CompCert’s Clight, which would then provide a verified compiler for SPARK 2014 to the target languages supported by CompCert. In addition, the Jago translation also enables one to develop in Coq an integrated verification environment that includes the ability to use Coq to mechanically verify that a SPARK 2014 program conforms...
to its formally specified contracts. In situations where very high confidence is needed, this type of infrastructure could be used directly by verification engineers, or it could enable existing automated tools like Kiasan\textsuperscript{61} or GNATprove\textsuperscript{62} to emit Coq proofs establishing that their verification results for a particular program are correct.

- Conditional declassification policy. The proposed security policy framework in this thesis allows user to specify the trust declassification functions to safely declassify information from one security domain to another. To capture more precise information flow, the policy can be extended with guard conditions to trigger the declassification actions, coming with a more advanced enforcement algorithm. It’s called the conditional information declassification policy that declassification function can be called only when certain conditions are satisfied.
Bibliography


[17] Andrew W. Appel. Verified software toolchain - (invited talk). In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the


Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline.
Programming Language Design and Implementation (PLDI), Vancouver, British Columbia,

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented


P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static anal-
ysis of programs by construction or approximation of fixpoints. In Conference Record of
the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan-


Roderick Chapman, Eric Botcazou, and Angela Wallenburg. SPARKSkein: A formal and fast
reference implementation of Skein. In Adenilso Simao and Carroll Morgan, editors, Formal
Methods, Foundations and Applications, volume 7021 of Lecture Notes in Computer Science,

Guillaume Claret. Falso – A Proof of False in Coq (for, e.g., v8.4pl5; fixed in v8.4pl6).

Pierre Courtieu, Maria-Virginia Aponte, Tristan Crolard, Zhi Zhang, Robby, Jason Belt,
John Hatcliff, Jérôme Guitton, and Trevor Jennings. Towards the formalization of SPARK


