Black-, Grey-, and White-Box Side-Channel Programming for Software Integrity Checking

by

Hong Liu

M.S., Beihang University, China, 2007

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2017

Abstract

Checking software integrity is a fundamental problem of system security. Many approaches have been proposed trying to enforce that a device runs the original code. Software-based methods such as hypervisors, separation kernels, and control flow integrity checking often rely on processors to provide some form of separation such as operation modes and memory protection. Hardware-based methods such as remote attestation, secure boot, and watchdog coprocessors rely on trusted hardware to execute attestation code such as verifying memory content and examining signatures appearing on buses. However, many embedded systems do not possess such sophisticated capabilities due to prohibitive hardware costs, unacceptably high power consumption, or the inability to update fielded components. Further, security assumption may become invalid as time goes by. For Systems-on-Chip (SoCs), in particular, internal activities cannot be observed directly, while in non-SoCs, sniffing bus traffic between constituent components may suffice for integrity checking.

A promising approach to check software integrity for resource-constrained SoCs is through side-channels. Side-channels have been used mostly for attacks, such as eavesdropping from vibration of glass or plant leaves, fingerprinting machines from traffic patterns, or extracting secret key materials of cryptographic routines using power consumption measurements. In this work, side-channels are used to enhance rather than undercut security. First, we study the relationships between the internal states of a target device and side-channel information. We use the uncovered relationships to monitor the internal state of a running device and determine whether the internal state is an expected one. An unexpected state may be a sign of incorrect execution or malicious activity.

To further explore the possibilities inherent in side-channel-based software integrity checking, we investigate various hardware platforms, representative of different degrees of knowledge of the hardware from the side-channel profiling point of view. In other words, side-channel information is extracted by black-, grey-, and white-box analysis. Each one involves unique challenges requiring different techniques to successfully derive "side-channel profiles". We can use these profiles to detect unexpected states with extremely high probability, even when an adversary knows that their code may be subject to side-channel analysis, i.e., the methodology is robust to side-channel-aware adversaries.

The research includes

1. Constructing systematic approaches for black- and grey-box profiling of side channels (and comparing them to white-box analysis);
2. Designing custom measurement instrumentation; and
3. Developing techniques for monitoring and enforcing software integrity utilizing side-channel profiles.

We introduce the term "side-channel programming" to refer to techniques we design in which developers explicitly utilize side-channel characteristics of existing hardware to optimize run-time software integrity checking, creating executable code which is more conducive to side-channel-based monitoring. Compared with other software integrity checking techniques, our approach has numerous benefits. Among them are that the measurement process is non-invasive, non-interruptive, and backward-compatible in that it does not require any hardware modification, meaning our approach works with processors that do not include security features. Our method can even be used to augment existing protection mechanism, as it works even when all security mechanisms internal to the device fail.

Black-, Grey-, and White-Box Side-Channel Programming for Software Integrity Checking

by

Hong Liu

M.S., Beihang University, China, 2007

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2017

Approved by:

Major Professor
Eugene Y. Vasserman

Copyright

© Hong Liu 2017.

Abstract

Checking software integrity is a fundamental problem of system security. Many approaches have been proposed trying to enforce that a device runs the original code. Software-based methods such as hypervisors, separation kernels, and control flow integrity checking often rely on processors to provide some form of separation such as operation modes and memory protection. Hardware-based methods such as remote attestation, secure boot, and watchdog coprocessors rely on trusted hardware to execute attestation code such as verifying memory content and examining signatures appearing on buses. However, many embedded systems do not possess such sophisticated capabilities due to prohibitive hardware costs, unacceptably high power consumption, or the inability to update fielded components. Further, security assumption may become invalid as time goes by. For Systems-on-Chip (SoCs), in particular, internal activities cannot be observed directly, while in non-SoCs, sniffing bus traffic between constituent components may suffice for integrity checking.

A promising approach to check software integrity for resource-constrained SoCs is through side-channels. Side-channels have been used mostly for attacks, such as eavesdropping from vibration of glass or plant leaves, fingerprinting machines from traffic patterns, or extracting secret key materials of cryptographic routines using power consumption measurements. In this work, side-channels are used to enhance rather than undercut security. First, we study the relationships between the internal states of a target device and side-channel information. We use the uncovered relationships to monitor the internal state of a running device and determine whether the internal state is an expected one. An unexpected state may be a sign of incorrect execution or malicious activity.

To further explore the possibilities inherent in side-channel-based software integrity checking, we investigate various hardware platforms, representative of different degrees of knowledge of the hardware from the side-channel profiling point of view. In other words, side-channel information is extracted by black-, grey-, and white-box analysis. Each one involves unique challenges requiring different techniques to successfully derive "side-channel profiles". We can use these profiles to detect unexpected states with extremely high probability, even when an adversary knows that their code may be subject to side-channel analysis, i.e., the methodology is robust to side-channel-aware adversaries.

The research includes

1. Constructing systematic approaches for black- and grey-box profiling of side channels (and comparing them to white-box analysis);

2. Designing custom measurement instrumentation; and

3. Developing techniques for monitoring and enforcing software integrity utilizing side-channel profiles.

We introduce the term "side-channel programming" to refer to techniques we design in which developers explicitly utilize side-channel characteristics of existing hardware to optimize run-time software integrity checking, creating executable code which is more conducive to side-channel-based monitoring. Compared with other software integrity checking techniques, our approach has numerous benefits. Among them are that the measurement process is non-invasive, non-interruptive, and backward-compatible in that it does not require any hardware modification, meaning our approach works with processors that do not include security features. Our method can even be used to augment existing protection mechanism, as it works even when all security mechanisms internal to the device fail.

Table of Contents

List of Figures . . . xi

List of Tables . . . xiii

Acknowledgements . . . xiii

Dedication . . . xiv

1 Introduction . . . 1
  1.1 Organization . . . 4

2 Background . . . 5
  2.1 Hardware platforms and security problems . . . 5
  2.2 Software integrity checking . . . 7
    2.2.1 External verifier versus internal verifier . . . 8
  2.3 Side channels . . . 12
    2.3.1 White-box analysis of side-channels . . . 13
    2.3.2 Side-channel analysis with limited knowledge of devices . . . 15
  2.4 Statistics and mathematical background . . . 18
    2.4.1 Pattern matching . . . 18
    2.4.2 Mathematical modeling . . . 21
    2.4.3 Variable selection . . . 23

3 Black-box Analysis . . . 26
  3.1 Related work . . . 26

3.2 Problem definition 28
3.2.1 Threat model 28
3.2.2 Experiment setup 30
3.3 A systematic approach for instruction-level side-channel analysis 31
3.3.1 Semantic models 32
3.4 Side-channel models 39
3.4.1 Side-channel models of instruction operations 40
3.4.2 Side-channel models of internal states 42
3.4.3 EM measurement 51
4 Side-channel Programming 54
4.1 Side-channel constrained transitions 55
4.2 Heuristics 62
4.3 Graph of side-channel programs 65
4.3.1 Random initial GPRs 65
4.3.2 Zero initial GPRs 66
4.4 Other heuristics 67
5 Grey-box Analysis 70
5.1 Background 70
5.2 Related work 72
5.2.1 White-box analysis of FPGA side-channels 72
5.2.2 Empirical models of FPGA side-channels 72
5.3 Problem definition 73
5.3.1 Threat model 74
5.4 Experimental setup 75
5.5 The test code and SoC test targets 78
5.5.1 System A: NIOS II-based SoC 78

5.5.2 System B: Resource-constrained NIOS II-based SoC 79
5.5.3 System C: OpenMSP430-based SoC 79
5.5.4 Test code 79
5.6 Modeling side-channels 80
5.6.1 Black-box analysis 81
5.6.2 Grey-box analysis 82
5.7 Validation 85
5.8 Applying results to software integrity checking 94
6 Discussion and Future Work 101
Bibliography 104

List of Figures

2.1 General software integrity checking problem with explicit communication channels and side-channels 8

2.2 Lumped-C model of a CMOS inverter[1] 13

3.1 The instruction set of PIC16F687[2] 29

3.2 Measurement setup 31

3.3 A single-captured waveform of executing "`MOVLW 0x69`" and "`ADDWF 0x40,F`" 40

3.4 Difference of two measurements with the same program but different register values. 43

3.5 A typical single-captured EM radiation of the target chip. 51

4.1 The average size of search trees for one instruction cycle: the x-axis is $(q2, q3, q4)$ (ranges $(0, 0, 0), (1, 0, 0), \ldots, (8, 14, 10)$); the y-axis is the average number of nodes of the resulting search trees. 63

4.2 The size of search trees after one instruction cycle versus $q2$, $q3$, and $q4$, respectively. The x-axis is $q2$, $q3$, or $q4$; the y-axis is the number of nodes of the resulting search trees. 64

5.1 FPGA architecture with logic blocks depicted as blue squares; logic blocks are surrounded by routing channels; several logic blocks have a dedicated memory column in-between. 71

5.2 Electromagnetic measurement 76

5.3 Measurements of system A when a subset of internal signals are identical . . 84

5.4 Pearson's r for model validation, with the profiling (modeling) results to the left and the testing (on another chip) results to the right; from the darkest bar to the lightest are ridge regression, PLS, PCR, and stepwise regression. . 86

5.5 Spearman's rho for model validation, with the profiling (modeling) results to the left and the testing (on another chip) results to the right; from the darkest bar to the lightest are ridge regression, PLS, PCR, and stepwise regression. . 87

5.6 Correlation of model prediction and measurement with sliding time window during profiling; the x-axis is the time offset, and the y-axis is Pearson's r computed from the actual measurement and the model prediction which has an offset in time. 89

5.7 Model prediction and measurements for the best PLS model of system A . . 90

5.8 The coefficients of the best models for system A, excluding the constant; from the top to bottom are coefficients of ridge, PLS, PCR, and stepwise regression, respectively. 91

5.9 The coefficients of the best models for system B, excluding the constant. . . 92

5.10 The coefficients of the best models for system C, excluding the constant. . . 93

5.11 EM measurements of instructions grouped by operations for system A; x-axis is the number of clock cycles; y-axis is the EM measurement. 97

5.12 EM measurements of `add r3,r16,r16` for system A; red cross indicates the same instance of executing `add r3,r16,r16`. 99

List of Tables

3.1 Regression analysis of power consumption in Q2 46
3.2 Regression analysis of power consumption in Q4 48
3.3 Regression analysis of EM radiation in Q2 52
3.4 Regression analysis of EM radiation in Q4 53

5.1 False positive rates (%) for a(n) (aligned) single-captured EM trace 95
5.2 False negative rates (%) for an aligned single-captured EM trace 95
5.3 False negative rates (%) for an arbitrary single-captured EM trace 95
5.4 Class separability J for system C 98

Acknowledgments

I owe a lot of thanks to my academic advisor, Professor Eugene Y. Vasserman. Without him, I would not have entered the amazing world of cyber security. He has been so helpful and supportive throughout my time as his PhD student. I would not have accomplished anything without his inspiring advice, encouragement, and patience. I feel very lucky to have such a great advisor.

I would also like to thank many of the professors of the Computer Science Department, as well as the professors of the Electrical and Computer Engineering Department for their great teaching, valuable discussion and help. I owe many thanks to Professor Bala Natarajan, Professor William Hsu, and Professor Steven Warren for their suggestions and concrete help in improving my research. I would also like to thank Professor Mitchell Neilsen and Professor Yurii Maravin for reading my dissertation and giving me insightful comments.

I would like to thank my laboratory mates and friends at the Computer Science Department for their generous help both in my research and in my life. They have brought so much fun to my PhD study and have made my life so much easier.

I would like to thank my family for their support and love, without which it would not have been possible for me to devote so much time and energy to my study and research. My love to them is the drive of my work.

Finally, I would like to thank the NSF grants that have made the research possible: NSF CNS 1253930 and NSF CNS 1224007.

Dedication

Dedicated to the humane treatment of animals

Chapter 1

Introduction

Electronic devices have become an integral part of our everyday life. From computers to electric cars to insulin pump to Process Logic Controls (PLCs) in industry, most electronic devices nowadays are cyber systems which are composed of one or more processors and are controlled by software. Since many of these devices are essential to our safety and well-being, verifying their integrity is an important task. Developers traditionally focus on realizing device functionality, while ignoring the fact that an attacker can change the behavior of a device by overwriting its program and/or data locally or remotely.[3]

Verifying software integrity is challenging. First, cyber devices are of great variety. They may be composed of general-purpose microprocessors, graphics processing units (GPUs), microcontrollers (μCs), digital signal processors (DSPs), complex programmable logic devices (CPLDs), field programmable gate arrays (FPGAs), or diverse application-specific integrated circuits (ASICs). The processor architecture may be von Neuman or Harvard, the instruction set may be reduced instruction set computer (RISC) or complex instruction set computer (CISC), the word size may be 8-bit or 64-bit, pipelining may be none or super-scalar, caches may be zero or several megabytes, etc. While the variety of cyber systems helps in preventing "single point of failure" in which a security breach of one processor will affect all systems, it also limits the applicability of any security mechanisms. For example, the security problems of FPGAs are very different from those of microprocessor-based systems.[4;5]

Second, many cyber devices are designed to realize special functionality, and are often of very restricted resources. These devices are loosely referred to as "embedded systems". The term "embedded" means the device is a system embedded within a larger system. Typical embedded systems are the control unit of an electronic thermostat that determines when to stop boiling the water, the Electronic Brake Control Module of a car that controls how to lock brakes, and the reactor shutdown unit of a nuclear power plant that decides whether to terminate the reactor in an emergency so that the entire town can be saved from meltdown. Since these devices are designed to accomplish very specific tasks, they are often very small and highly optimized systems that may consist of an 8-bit processor and several kilo-bytes of memory. The simplicity of these systems, especially the common missing of security mechanisms in legacy devices, facilitates tampering. To guarantee integrity of these systems, without updating them with sophisticated but costly processors or security modules, requires innovative work.

In this thesis, several integrity checking approaches are presented with the goal of ensuring software integrity on various hardware platforms. These approaches are based on "side-channels" of a device. Side-channels can be defined as any channels that are not the main communication channel of a device. The term "side" is used to infer that the channel carries information as a side-effect of running desired functionality on a physical device. Common side-channels are power consumption, electromagnetic radiation, timing and quantity of traffic, thermal and light imaging, object vibrations and movement, etc.[6–10] There are also some uncommon side-channels, such as processor cache misses and the /proc virtual filesystem. Historically, side-channels have been used for a long time to penetrate a seemingly closed system. Here they are used instead for constructive purpose – checking software integrity of diverse devices in which security mechanisms have not ever been built in.

There are two types of integrity checking schemes that are proposed in this thesis. One is pure passive, which does not require the target device to modify its hardware *or* software. The other is a design-for-security scheme in which software of a device is rewritten based on the side-channel characteristics of its hardware so that the integrity of the software can be verified during device execution. Both schemes do not require modification of hardware

or interruption of device execution. The second scheme is useful in scenarios where the first scheme is not applicable due to side-channel features of the target hardware.

The side-channel-based software integrity checking schemes proposed in this thesis are external to the target device (c.f. Section 2.2.1). The idea is to first extract side-channel information and correlate it with the internal state of a running device, and then verify real-time side-channel emanations of the target device against desired values, without any explicit checking procedures within the device. Compared with other software integrity checking techniques, this approach has numerous advantages, such as incurring low or no overhead to existing systems, easy to deploy and upgrade the verification algorithms, no interruptions of normal execution, low or no visibility to attackers who penetrate the target device, applicable to legacy and deployed systems. An external verifier is more robust than internal verifiers which are confined by the hardware resources of the target device. A verifier that constantly observe the runtime behavior of the target device is able to defend against more sophisticated attacks such as code-reuse attacks and data-based attacks. In particular, security assumptions always weakens with time. The side-channel-based methods can detect tampering and verify validity of security assumptions without modifying original systems, given that side-channel analysis of the system proves so.

However, this approach poses significant challenges. First, we usually only have very limited knowledge of the target hardware platforms. Very little information can be derived for side-channel profiling from the released documents by the hardware manufacturers. Second, noise plays an important role in the success of our study. We must carefully design experimental instrumentation and data processing algorithms to minimize the effect of noise. Third, our integrity checking approach must be accurate enough to deal with compact malware that is potentially as short as a single instruction. Fourth, we must be able to secure against side-channel-aware attackers who may write malware that minimizes side-channel deviations. Last but not least, our verification approach must be realistic. Although it is theoretically possible to verify every aspect of an integrated circuit through micro-probing and microscoping, it is not practical to perform such analysis on every device that needs integrity checking, due to the high cost of testing equipment and time and expertise needed

to perform such verification.

In this thesis, the feasibility and generality of using side-channels for software integrity checking in diverse hardware platforms is explored. Since increasing side-channel leakage may violate EMI/EMC requirements. We have not made any effort to increase side-channel emanations but have instead measured the inherent side-channels of a target device. A systematic approach is present for efficiently analyzing side-channel characteristics of a target device over the entire instruction set. The effectiveness of this approach is shown by clock-cycle-accurate side-channel profiling of various SoCs with different degrees of knowledge on the system. The performance of the proposed side-channel-based software integrity checking schemes is quantified by the very low probability that a tampering is not detected from side-channel measurements, given both conventional attackers and side-channel-aware attackers who actively try to evade side-channel-based integrity checking.

1.1 Organization

Chapter 1 gives a brief introduction of the motivation. The background information and common related work are given in Chapter 2. Chapter 3 describes the approach to profile the side-channel emanations of a microcontroller with high accuracy that enables side-channel-based software integrity checking. Based on the side-channel-model obtained in Chapter 3, Chapter 4 proposes a novel Design-for-Security approach that utilizes side-channel characteristics of a black-box device to write code with guaranteed integrity. Chapter 5 presents the research of side-channel-based software integrity checking on a distinct hardware platform – FPGA. Three different SoCs implemented on the FPGA are studies to show the effectiveness, efficiency, and generality of the proposed approach. Finally, Chapter 6 summarizes the thesis and gives the future work.

Chapter 2

Background

2.1 Hardware platforms and security problems

One of the major difficulties in securing cyber systems is diversity. Early systems are composed of separate components, and it is easy to observe the structure and model of the system. It is also easy to analyze the functionality of the system by sniffing the signals between components (chips). From the perspective of software integrity checking, this system organization is beneficial in that the designers and testers can identify tampering and faults conveniently by using low-cost tools such as oscilloscopes and logic analyzers. On the other hand, attackers can reverse-engineer and clone the system in the same way.

With the emergence of microcontrollers, the CPU, memory, and common peripherals become integrated in a single chip package. Such a system is called System-on-a-Chip (SoC) and has become very common in cyber devices, especially embedded systems. Today's SoCs are composed of not only the traditional digital parts such as the processor, memory and popular I/O controllers, but also analog and mixed-signal parts including oscillators and phase-locked loops (PLL), analog-to-digital converters (ADCs), digital-to-analog converters (DACs), temperature sensors, and radio-frequency units to interface with WiFi, Bluetooth, and ZigBee wireless networks. Programmable hardware, such as CPLDs and FPGAs, has also become more favorable to SoC designs, by integrating "hard-core" processors and analog

components such as ADCs and DACs. By combining most components into one silicon substrate and by controlling the device with software stored in internal memory, SoCs are hard to reverse-engineered using traffic sniffing. Debugging and examining runtime signals are at the meantime difficult, if not using built-in debugging mechanisms such as JTAG.

However, it is not at all impossible to observe the internal structure and signals of SoCs. At the lowest level, wafer fabrication plants and chip manufacturers are equipped with process control and failure analysis tools such as scanning electron microscopes (SEMs), transmission electron microscopes (TEMs) and focused ion beam (FIB) machines that are able to examine and modify nanometer scale structures.[11;12] Microprobing can measure the state of a SRAM cell or even transistor.[13–15] Often the plastic package of the chip is etched off in a corrosive acid solution, e.g., hot fuming nitric acid. For hermetic and ceramic packages, mechanical or thermal treatment may be used.[12] The silicon may be accessed from the back side of the chip, or the metal layers need to be removed layer by layer.[11] These techniques are the "ultimate" tools for analyzing or modifying target cells or transistors, especially memory cells that store secret keys or security fuses. They however require invasive access to the DUT and very expensive equipment which is not available for ordinary companies and university labs. Another disadvantage is that these techniques do not scale well for large circuits when design details are unknown. Today's ultra-large-scale integration (ULSI) circuits may be composed of hundreds of millions of CMOS transistors. The cost of a full chip reverse engineering is estimated to be around 100 thousands of Euros for a 130nm technology chip containing 100k logic gates.[16]

Semi-invasive analysis only requires depackaging the chip with the passivation layer intact.[11;17;18] Semi-invasive techniques require less expensive equipment but can achieve similar degree of control over the DUT. It is possible to extract information from internal memory such as SRAM and EEPROM, and to modify SRAM content and change the state of any individual CMOS transistor.[11]

By using invasive and semi-invasive analysis, it is fundamentally possible to verify the integrity of any cyber device, since every internal signal can be read out given enough time and equipment. In practice, however, such analysis is too expensive, especially for long-

term and bulk verification. In addition, it is not practical to apply semi-invasive analysis or invasive analysis for software integrity checking of deployed devices.

One solution is to combine invasive and semi-invasive analysis with non-invasive analysis. Non-invasive analysis often utilizes "side-channel" emanations of a device to infer its internal activities. Side-channel information such as power consumption, temperature, and electromagnetic emanations can be easily measured using low-cost equipment and non-invasive access to the chip. In[19], authors propose the use of side-channel profiles of chips to test against a "golden sample" for hardware trojan detection. The golden sample is a reference chip whose genuineness is verified by invasive analysis. Very often a great amount of information can be obtained solely from non-invasive analysis. In this thesis, all experiments are performed by using non-invasive access, low-cost equipment, and passive measurement, and are favorable to legacy and deployed systems.

2.2 Software integrity checking

We define the general software integrity checking problem as follows. There are two parties: a prover P, a device-under-test (DUT) running the target application software S, and a verifier V, a trusted entity who would like to determine whether P runs S or S', which may be a different piece of code or original code but in an unintended execution state. P and V communicate over an explicit channel C and a side-channel E, as shown in Figure 2.1. V bases its judgment on evidence that P provides directly (e.g., by signatures) over C or indirectly (e.g., by timing or EM radiation), over E. V knows the initial configuration of P, including hardware and software.

The problem model can be instantiated in numerous ways. In a microcontroller-based system, as considered in Chapter 3 and 4, S is naturally the software running on the microcontroller, and P is the hardware including the microcontroller chip and the printed circuit board (PCB). In an FPGA-based system, as considered in Chapter 5, the situation is a bit more complex, since for FPGAs, both the hardware and the software are programmable. The FPGA configuration logic describes both the hardware (processor, memory, IO, etc.) of

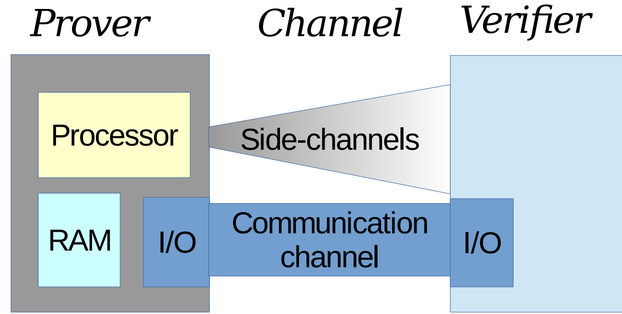


Figure 2.1: General software integrity checking problem with explicit communication channels and side-channels

the system implemented on an FPGA chip, as well as the application software that runs on the system. Since there are many methods to prevent reconfiguration of an FPGA device (c.f. Section 5.3), S is therefore defined as the application software, and P incorporates the PCB, the FPGA chip, and the FPGA configuration logic describing the hardware of the system.

2.2.1 External verifier versus internal verifier

Many approaches have been proposed to enforce software integrity. They can be classified by where the verifier resides. An *internal* verifier performs integrity checking in the same device with the target software and thus shares hardware resources with the software, whereas an *external* verifier does its job outside the device and therefore often has unbounded resources.

Internal verifiers

Internal verifiers can be either software or hardware. Conventional anti-virus software and intrusion detection/protection systems are most common software-based internal verifiers that protect or detect tampering of system software and memory by identifying software signatures, verifying checksums, analyzing traffic and software behaviors, etc. Hypervisors[20–22], separation kernels[23], mandatory access control[24;25], and control flow integrity[26–28], are more recent internal software-based approaches that provide more confined environments for un-

trusted software and/or restrict software behaviors in a finer granularity. The common property of software-based internal verifiers is that the verifier shares the same hardware with the target software. A verifier that runs on the main CPU of a system therefore cannot detect tampering on the peripherals (e.g., the Ethernet card[29]). In addition, usually software-based internal verifiers detect or prevent "anomalous behavior" of programs by utilizing hardware security features that provide some form of separation such as different operation modes and memory protection. While modern computers possess sophisticated processors that have rich security features, cyber devices as a whole often cannot afford such a processor because of cost, power, or space constraints. It is not uncommon that an embedded system only has an 8- or 16-bit processor that does not even support integer multiply operation. Many software-based approaches that rely on hardware-supported separation are therefore not applicable to such devices.

Software Symbiote[30] is a noteworthy software-based internal verifier that does not rely on sophisticated processors. Its idea is to insert verification code duplicates, which perform self-checksumming, into host software so that the code gets executed from time to time and will trigger an alarm when checksums are different. The major weakness of Symbiote is that the verification code can be inactivated or even removed by a sophisticated attacker who is aware of the existence of such a protection mechanism, as no protection on the verification code itself can be guaranteed if a processor lacks appropriate security features.

Another weakness that all software-based internal verifiers share is that the verifier competes for resources with the target software. For cyber devices that perform real-time tasks, the designers must carefully balance security and real-time requirements, in which case security is often the one that is sacrificed.

Unlike software-based approaches, hardware-based internal verifiers build dedicated hardware components for software integrity checking and are more or less invisible to the software under test. Example verifiers include watchdog coprocessors[31;32], secure boot, and dynamic root of trust.[33] Often the trusted hardware implements some cryptographic routines and stores sensitive information such as secret keys. The hardware is responsible for verifying checksums and signatures of code, such as BIOS code and firmware, and executes the code

only if the checksum/signature is valid. It may also send out alarms to remote servers if tampering is detected. In particular, trusted platform module (TPM)[34;35] is a security chip that exists in most enterprise-grade personal computers and costs for only around 1 dollar.[35] A TPM measures itself, BIOS code, the master boot record, etc., and stores integrity checksums in Platform Configuration Registers (PCRs), which is a protected memory. A verifier asks the TPM for a signed copy of the PCRs to evaluate the boot sanity. As we all see, the existence of TPMs does not guarantee that our computers are not tampered. This is because TPM is a passive device that is operated by code running on the main CPU.[35] At best, it can only be used to guarantee that the boot procedure of a system is clean. If the system is breached after boot, secure boot with the TPM cannot detect or prevent malicious behavior. Instead, dynamic root of trust such as Intel TXT[36] and AMD SVM[37] performs attestation on the current state of the software by utilizing specific CPU instruction which resets the TPM PCRs. Nevertheless, verification on solely code integrity is vulnerable to code-reuse attacks, such as return-into-libc[38], Return-Oriented Programming[39], and Jump-Oriented Programming[40], which execute original code but in undesired order. It is also vulnerable to the even more sophisticated data-only attacks[41], which execute original code in normal order but with wrong data. At a finer granularity, a coprocessor can be used to examine the runtime code that appears on buses and verify integrity by matching code patterns. If carefully designed, such a coprocessor can detect code-reuse attacks and data-only attacks. In general, hardware-based internal verifiers are more powerful than software-based internal verifiers, at the cost of additional hardware and/or more complex CPU and chipset support. It is also more difficult to deploy new hardware or fix hardware flaws.

External verifiers

The verifier can also be outside of the DUT. Compared to internal verifiers, external verifiers do not compete for resources with the normal software, or require additional hardware, and it is much easier to deploy and update an external verifier. External verifiers also have minimal visibility to attackers who penetrate the device. Moreover, a verifier that is external to a

device is the only possible integrity verifier when all the security mechanisms internal to the device fail, or do not even exist.

On the other hand, since the verifier is external to the DUT, it is possible to perform "proxy attack" in which the attacker asks another device/component (proxy), instead of the tampered one, to help provide integrity proof to the verifier. Some mechanism must be used to authenticate the source of the integrity proof.

Software attestation[42–44] is a software integrity checking scheme that relies on an external verifier. Compared to other integrity checking schemes, it has an advantage in requiring no sophisticated hardware security features or hardware modification, which makes it applicable to legacy devices. Software attestation assumes that an external verifier has direct access to the DUT – the device and integrity measurement channel are authenticated out-of-band by for example visual inspection. When attestation begins, the verifier asks the DUT to put the device to a known state and compute a checksum of the memory of the device, using a nonce provided by the verifier. The verifier tests the returned checksum and the time for the DUT to compute the checksum, so that any tampering of the memory (including all available registers and caches) will be reflected in the checksum and the device does not have enough time to remove malware before checksum computation. Software attestation is able to detect compact malware – malware composed of very few instructions – and is secure against side-channel-aware attackers. Here timing is a key side-channel information to make sure that the attacker is unable to evade detection, even if she is able to profile the computation time of the target device.

Remote attestation[45–47] aims to perform software attestation on a remote device. The major problem is to prevent proxy attacks, as the timing side-channel cannot be used as a reliable detector of proxies in this scenario. Current solutions are based on a trusted hardware similar to TPM that is implemented on the DUT, and proxy is prevented by using the secret key shared by the trusted hardware and the verifier. Remote attestation relies on the somewhat unrealistic assumption that a remote device cannot be tampered physically (the trusted hardware is secure) and the secret key cannot be leaked. As shown in later sections, side-channel analysis, microprobing, reverse engineering, and fault injection can

fundamentally reveal any secret of a device.

In addition, there are some common weaknesses of software attestation and remote attestation. They both attest on known memory configuration, and are therefore vulnerable to code-reuse attacks, self-delete malware, data-only attacks, and time-of-check and time-of-use (TOCTOU) attacks (in which the memory configuration is tampered and then restored between two attestations).

The software integrity checking schemes proposed in this thesis are external approaches. The proposed approaches have the same benefits as an external verifier has: low or no visibility to penetrators, low or no overhead to normal software, and easy deployment and upgrade. Furthermore, the proposed schemes in this thesis, unlike software attestation and remote attestation, measure runtime software integrity passively, without explicit attestation procedures, and therefore do not interrupt normal execution or have the TOCTOU problem. Like software attestation, they do not require CPUs with sophisticated security features, or modification of the hardware platforms. This makes our approaches favorable to legacy and deployed devices. To achieve these goals, our approaches utilize non-invasive passive side-channel emanations of electronic devices.

2.3 Side channels

It is common-sense that electronic devices consume power, cause heat, and emit electromagnetic (EM) radiation during execution. Power consumption, temperature, and EM measurements are resulted from the internal activities of the complementary metal-oxide-semiconductor (CMOS) circuits that comprise most of modern electronic devices. Therefore these side-channels, beside the explicit communication channel of a device, also carry information about the runtime state of a device.

2.3.1 White-box analysis of side-channels

Why can power consumption, thermal and light imaging, and EM radiation carry information about a running device? It lies in the physics of electronic devices. An electronic device is often composed of analog components and digital components. For software integrity checking, only the digital part associated with a typical cyber system is concerned. Digital circuits are always built based on logic cells, which are commonly implemented by using CMOS.[1] Figure 2.2 shows a CMOS implementation of an inverter, a basic unit of digital circuits.

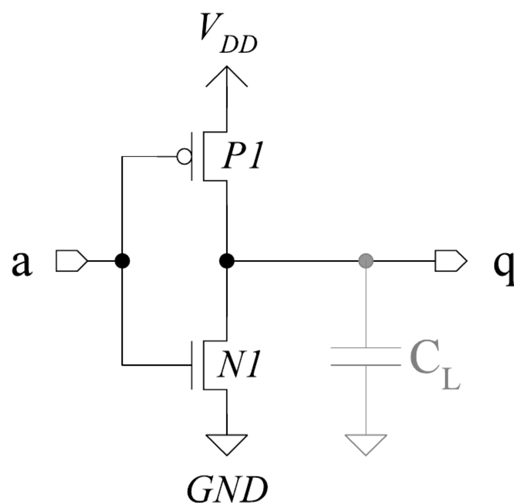


Figure 2.2: Lumped-C model of a CMOS inverter[1]

Power consumption of an inverter is composed of static power consumption and dynamic power consumption. CMOS inverter is designed such that there are no direct connections between `Vdd` and `Gnd`, except a small leakage current flowing through the MOS transistor that is off. This leakage current is the static consumption, which is usually 1pA. Dynamic power consumption is caused by internal or output signal switches. For the inverter, dynamic power consumption is incurred when input is changed. There are two types of dynamic power consumption: switching power and short-circuit power.[1;48–50] When the input switches from 1 to 0 (i.e., output from 0 to 1), the load capacitance of the cell needs to be charged, and therefore current flows from the `Vdd` through the load capacitance. Load capacitance incor-

porates the internal capacitance connected to the output, the capacitance of wire connected to other cells, and the input capacitance of these cells. Typical load capacitance ranges from 1fF to 1pF. When the input switches from 0 to 1 (i.e., output from 1 to 0), the load capacitance is discharged, and current flows in opposite direction in the load capacitance. Internal switches can be ignored because output causes much larger power consumption. Short-circuit power consumption is the temporary short circuit between the two complementary transistors (PMOS and NMOS) during switching.

Dynamic power consumption of a circuit is often modeled as a sum of power consumed by all the nodes[50]:

$$P_{sw} = \frac{1}{2} f V_{dd}^2 \sum_{i=1}^{n} C_i \cdot E_i \tag{2.1}$$

$$P_{sc} = \alpha_{sc}(t_r) P_{sw} \tag{2.2}$$

where P_{sw} is the switching power consumption, f is the clock frequency, C_i is the load capacitance of node i, E_i is the transition density (frequency of switches) of node i, P_{sc} is the short-circuit power consumption, $\alpha_{sc}(t_r)$ is a linear function of the input transition time t_r. $\alpha_{sc}(t_r)$ is decided by a linear curve fitting of simulated dynamic power.

More accurate power models also consider the cross-talk (interference) between signals of neighboring wires[51]:

$$E_{total} = \sum_{i=1}^{n} (C_L + C_{eff,i} C_I) \cdot \Delta V_i \cdot V_i \tag{2.3}$$

$$C_{eff,i} = abs(C_L \delta_i + C_I \delta_{i,i-1} + C_I \delta_{i,i+1}) \tag{2.4}$$

where E_{total} is the total energy consumption for a given bus transition, C_L is load capacitance seen by the driver, C_I is the inter-wire coupling capacitance between adjacent signal lines, $C_{eff,i}$ is the effective total capacitance of the driver of i-th line, $\delta_i \in \{0, 1\}$ is the normalized voltage change on i-th line, $\delta_{i,i\pm1} \in \{0, \pm1, \pm2\}$ is the normalized relative voltage change on

i-th line relative to the $(i \pm 1)$-th line, $\Delta V_i \in \{0, \pm V_{dd}\}$ is the voltage change on i-th line.

These power models are often used in combination with SPICE[52] simulation in company with detailed manufacture parameters to estimate dynamic power of small-scale circuits.[48–50;53–55] In practice, however, the complexity and lack of knowledge of cyber devices prevent such computation. Researchers in turn try to analyze side-channel properties from real measurements.

2.3.2 Side-channel analysis with limited knowledge of devices

People have studied side-channel emanations from empirical measurements in diverse areas. Historically side-channels have been used mostly for attacks. In recent years, researchers start to try using side-channels for constructive purposes.

Power-efficient device design Empirical study of side-channel emissions of various embedded systems have long been studied to optimize power usage and perform EMI/EMC analysis. Researchers usually obtain power consumption of a target component of a cyber system by averaging the energy consumed when executing the component for a long time. This can be done at different levels. For example, application developers usually read the voltage of the battery of a mobile phone from time to time to determine the power consumption of the system so that some power-saving strategy can be applied. Researchers have also studied the empirical power consumption of instructions of various devices (most commonly mobile device) to evaluate the efficiency of the processor design or to guide power-efficient software development. In such study, researchers often execute the target instruction, e.g., multiply, for many times and compute the empirical power consumption or other side-channel information such as EM cartography by averaging the accumulative side-channel emanations. This research does not concern runtime side-channel emanations for individual instruction execution.[56–58]

Side-channel analysis for cryptographic hardware More recently, side-channel analysis has been used mostly for attacks. Numerous papers are published each year on how to

utilize power consumption or EM radiation of cryptographic hardware to break the embedded secret keys. Attackers are naturally of very limited knowledge of the target device, which yet cannot prevent them from correlating the side-channel emissions of the cryptographic device with the secret key materials by executing the same cryptographic routine (whose implementation details or even algorithm is unknown) a large number of times (typically thousands of times).[1;59–62]

Covert channel Another (mis-)use of side-channels is data exfiltration, in which side-channels act as covert channels that silently transmit information about a running device. Tempest radiation is known as the stray RF emitted by electronic devices that can be used by an opponent to reconstruct the information about the data being processed.[63] Again, attackers do not need to build complete side-channel models for the target device in order to achieve their goals. One notable example dates back to 1980s when a researcher described how to reconstruct the picture on a VDU at a distance using a modified TV set.[64] More recently, researchers showed that it was possible to recover speech from the vibrations of a potato-chip bag photographed from 15 feet away through soundproof glass.[65]

Fingerprinting Empirical study of side-channels has also been used for device fingerprinting. Browser response patterns allow websites to track users without the need of client-side identifiers.[66] Ethernet devices can be uniquely identified by analyzing variations in their analog signal caused by hardware and manufacturing inconsistencies.[67] For constructive purpose, IC fingerprinting profiles the power, temperature, and EM (including light) characteristics of an IC family and further uses the profiles to detect hardware trojans (malicious circuits substituting the genuine ones).[6–10;68] The side-channel measurement of a DUT is compared with that of a "golden-model" which is invasively examined to ensure that no trojans exist. Hardware trojans are composed of sequential circuits and/or combinational circuits, and trojans of pure combinational circuits are hard to detect since they are barely triggered except for specific input pattern. On the other hand, if it is possible to detect combinational trojans, then so for the sequential trojans. With combinational trojans as the target, research

on IC fingerprinting often scans emissions of the IC for enough time so that untriggered trojan circuits can be exposed from little bias in side-channel measurements. IC fingerprinting therefore does not concern with runtime side-channel emanations.

Side-channel disassemblers Side-channel analysis of runtime emanations has been studied for different purposes. One is to recognize instruction operations from side-channel measurements, referred to as "side-channel disassemblers" in[69]. Power consumption or EM radiation of some smart cards and microcontrollers are collected when running instructions using random data input, and statistic and pattern matching techniques, such as Principal Component Analysis (PCA) and template analysis, are used to classify among side-channel measurements of different instruction operations.[17;18;69–74] In[17] the authors claimed a relevant recognition rate of 96.24% on test data and 87.69% on real code by using multi-position localized EM emissions and semi-invasive access to a PIC microcontroller. In[74], a 100% classification rate was reported by using power measurements on a AVR microcontroller. However, neither[17] nor us have succeeded in repeating the authors' results on a PIC microcontroller, which is a simpler microcontroller.

Integrity checking Another use of runtime side-channel information is to verify system integrity, as this thesis is concerned. Researchers have analyzed passive system-wide power measurements of general programs in the hope of detecting anomalous behaviors and/or malware.[73;75–78] These methods, however, assume malware (code) to be sufficiently long and not written to conceal its side-channel profiles. To use side-channels for rigorous integrity checking, we must consider compact and side-channel-aware malware. Software attestation[42–44;79] utilizes the timing side-channel and is capable of detecting malware at such precision. However, the device must support such attestation, and carrying out the process requires interruption of the device execution, a particular drawback for legacy and actively-used systems (see also Section 2.2.1).

Micro-probing At the lowest level, it is possible to measure the EM radiation of a single transistor or SRAM cell by using intrusive measurement and microscopic probing, as mentioned in Section 2.1. This method however does not scale well for integrity checking of a complex system that is composed of tens of thousands and more transistors. It does not solve the problem of efficiently verifying the software of a modern embedded system in practice.

2.4 Statistics and mathematical background

The proposed approaches in this thesis are composed of two procedures: side-channel modeling (profiling) and tampering detection. Side-channel modeling utilizes statistics, mathematical modeling, and pattern matching techniques to build relationships between the internal state of a target device and side-channel measurements. Tampering detection is to, based on the resulting side-channel model, determine from a new side-channel measurement whether the internal state of the device is a desired one or not. This section will briefly introduce the statistic and mathematical knowledge that is commonly used in the following chapters.

2.4.1 Pattern matching

Pattern matching/machine learning techniques are widely used in side-channel analysis since the interested internal states of a physical device are usually finite. For example, the secret sub-keys are often only 8-bit long, and at most 256 classes are needed (see[80] for various ways of classifying an 8-bit sub-key). So side-channel profiling can usually be applied with general classifiers. Popular techniques include Support Vector Machines (SVMs), template analysis (TA), and neural networks.[70;80–82] Template analysis has turned out be the most powerful one and are more relevant with the proposed approaches in this thesis.

Template analysis

Template analysis solves a classification problem by first build a set of reference patterns (templates) and then decide which one of the reference patterns matches best with a new test pattern; some measure is used to define the distance between reference patterns and the test pattern.[83] Template analysis in the context of attacking cryptographic hardware often assumes that the side-channel measurements (usually a sampled trace of power consumption) can be modeled as multi-variate Gaussian signals. Then templates are built by estimating the parameters of the Gaussian signals for each of the interested internal states (classes) from executing the device for a large number of times.[69;81] For each class ω_i, select l samples in side-channel measurements for modeling, the templates are l-dimensional Gaussian distributions with parameters estimated from power consumption observations when executing the target device under ω_i:

$$\begin{aligned} p(\boldsymbol{x}|\omega_i) &= \frac{1}{(2\pi)^{l/2}|\boldsymbol{\Sigma}_i|^{1/2}}\exp\Big(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)^T\boldsymbol{\Sigma}_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i)\Big) \\ \boldsymbol{\mu}_i &= \frac{1}{N_i}\sum_{j=1}^{N_i}\boldsymbol{x}_{ij} \\ \boldsymbol{\Sigma}_i &= \frac{1}{N_i-1}\sum_{j=1}^{N_i}(\boldsymbol{x}_{ij}-\boldsymbol{\mu}_i)(\boldsymbol{x}_{ij}-\boldsymbol{\mu}_i)^T \end{aligned} \tag{2.5}$$

where $\boldsymbol{x}_{ij}$ is an l-dimensional observation of executing the device under ω_i in the modeling data, N_i is the number of such observations in the modeling data.

When given a new observation $\boldsymbol{x}$, its internal state $\hat{\omega}$ is estimated by applying the Bayes rule, which is the ω_i that gives the maximum a posteriori probability.

$$\hat{\omega} = \underset{\omega_i}{\operatorname{argmax}}\, p(\omega_i|\boldsymbol{x}) = \underset{\omega_i}{\operatorname{argmax}}\, p(\boldsymbol{x}|\omega_i)P(\omega_i) \tag{2.6}$$

Template analysis has been shown to be the most powerful tool in breaking cryptographic hardware, as it can discover secret keys when given only a single test trace.[69;82;84] Other techniques usually requires hundreds or thousands of traces. Template analysis however

has some limitations. First, template building (sometimes referred to as "device profiling") requires full access to the target device, which is not always possible. Second, the profiling step requires executing the device for a very large number of times under each interested state, which may make the approach impractical, especially if the state space is large (which is not uncommon). Third, the performance of the approach will degrade significantly if the assumed model cannot accurately describe the real measurements or if the pre-selected l-dimensional observation is not well related with the interested states.

Class separability

A metric that is useful to examine whether the pre-selected observation (aka features in pattern matching literature) are representative to solve the template analysis problem is the class separability measure. It tells how likely the classification problem will be solved, since if classes are not well separable given selected features, then no techniques can achieve good classification performance.

The upper bound of the minimum attainable error of the Bayes classifier is[83]:

$$P_e \leq \epsilon_{CB} = \sqrt{P(\omega_i)P(\omega_j)} \int_{-\infty}^{\infty} \sqrt{p(\boldsymbol{x}|\omega_i)p(\boldsymbol{x}|\omega_j)} d\boldsymbol{x} \tag{2.7}$$

where P_e is the classification error of the Bayes classifier, ϵ_{CB} is known as the Chernoff bound, $p(\boldsymbol{x}|\omega_i)$ is the probability density function of features $\boldsymbol{x}$ under class ω_i.

For multi-variate Gaussian signals, as assumed in template analysis and commonly in modeling power consumption and EM radiation:

$$\epsilon_{CB} = \sqrt{P(\omega_i)P(\omega_j)} \exp(-B_{ij}) \tag{2.8}$$

where

$$B_{ij} = \frac{1}{8}(\boldsymbol{\mu}_i - \boldsymbol{\mu}_j)^T (\frac{\boldsymbol{\Sigma}_i + \boldsymbol{\Sigma}_j}{2})^{-1} (\boldsymbol{\mu}_i - \boldsymbol{\mu}_j) + \frac{1}{2} \ln \frac{|\frac{\boldsymbol{\Sigma}_i + \boldsymbol{\Sigma}_j}{2}|}{\sqrt{|\boldsymbol{\Sigma}_i||\boldsymbol{\Sigma}_j|}} \tag{2.9}$$

where μ_i is the mean of class ω_i, Σ_i is the covariance of ω_i. B is known as the Bhattacharyya

distance and used for class separability measure of Gaussian signals. The smaller B is, the larger classification error will be incurred. See Section 3.4 for the Bhattacharyya distance of power consumption of a microcontroller grouped by instruction operations.

Another class separability measure that does not assume Gaussian distribution is to use the within-class and between-class scatter matrices[83]:

$$J = \frac{trace(S_m)}{trace(S_w)} \tag{2.10}$$

$$S_m = S_w + S_b \tag{2.11}$$

$$S_w = \sum_{i=1}^{M} P_i \Sigma_i \tag{2.12}$$

$$S_b = \sum_{i=1}^{M} P_i(\mu_i - \mu_0)(\mu_i - \mu_0)^T \tag{2.13}$$

where P_i is the a prior probability of class ω_i, M is the total number of classes, μ_i is the mean of class ω_i, Σ_i is the covariance of ω_i, and $\mu_0 = sum_{i=1}^{M} P_i \mu_i$ is the global mean vector. See Section 5.6 for the J values of EM radiation of FPGA-implemented SoCs grouped by instruction operations.

2.4.2 Mathematical modeling

When the modeling target is not finite, then classification techniques are not applicable. In Section 3.4 and 5.6, the side-channel measurement, which is naturally of continuous values, is the response variable, and depends on internal states of the target device. Note that literature in attacking cryptographic hardware often takes the side-channel measurements as the predictor variables (features) and the internal states as the response variables (classes). When the response variable is continuous, regression techniques are the tools to establish the relationships between side-channel measurements and internal activities.

Assume the observation Y_t at time t depends on a set of variables $\vec{x_t} = (x_{t1}, \dots, x_{tp})$ at t, which are often controllable in experiments:

$$Y_t = f(\vec{x_t}) + N_t \tag{2.14}$$

where N_t encloses remaining components in the EM radiation including noise and time-dependent components, Y_t and N_t are necessarily random variables. $x_{tj}(j = 1, \dots, p)$ are called the predictor variables and Y_t the response variable.

Regression techniques estimate the function $f(\cdot)$ from a large number $n > p$ of experiments to exercise the controlled predictor variables $\{\vec{x_i} | i = 1, \dots, n\}$ and collect the corresponding observations $\{Y_i | i = 1, \dots, n\}$. Commonly assumed form of $f(\cdot)$ is linear:

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_{q-1} x_{i,q-1} + N_i \tag{2.15}$$

where $\beta_j (j = 0, \dots, q-1)$ are constants.

The power consumption model of dynamic switching power 2.1 is similar to this form when we consider the transition density E_i as the predictor variables and load capacitance C_i as the constants, or vice versa.

$x_{ij}(j = 1, \dots, q-1)$ may be derived from $x_{ij}(j = 1, \dots, p)$. For example, a polynomial regression model, which is a special case of linear regression, obtain the q predictor from a polynomial function of $x_{ij}(j = 1, \dots, p)$. A second-order polynomial regression model with two predictor variables is:

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_{11} x_{i1}^2 + \beta_{22} x_{i2}^2 + \beta_{12} x_{i1} x_{i2} + N_i \tag{2.16}$$

This form is familiar as the power consumption model of bus signals 2.3 that consider interference between neighboring buses has the "interaction term" $x_{i1} x_{i2}$ too.

If the assumed linear function is a good approximator of the real function $f(\cdot)$, then least-square estimation can be used to estimate $\vec{\beta} = (\beta_0, \dots, \beta_{q-1})$. Least-square estimation does not make any assumption on the distribution of $x_{ij}(j = 1, \dots, p)$, and the resulting

estimate $\hat{\vec{\beta}}$ is the optimal unbiased estimator of $\vec{\beta}$, given $E(N_i) = 0$, $Var(N_i) = \sigma^2$, where $E(X)$ is the expectation of random variable X, and $Var(X)$ is the variance of X.

$$\hat{\vec{\beta}} = (X'X)^{-1}X'\vec{y} \tag{2.17}$$

where $X = \{x_{ij}\}_{n \times q}$ is the matrix of the predictor variables. However, as shown in Section 5.6, directly applying least-square estimation does not always work in practice. The most notable problem encountered in this thesis is the multicollinearity among the predictor variables $x_j (j = 1, \ldots, q)$. Multicollinearity is the phenomenon where some of the predictor variables are correlated with each other, which will lead to singularity of $(X'X)$. Several techniques can be used to eliminate multicollinearity, and will be discussed in details in Section 5.6.

2.4.3 Variable selection

A common problem that is encountered both in pattern matching/classification and in regression is variable/feature selection. Sometimes variable/feature selection techniques are also used for dimensionality reduction or noise reduction. Very often in our experiments, one observation (response) is associated with many predictor variables (i.e., features) and it is difficult or impractical to exclusively control a small set of variables while keeping the others constant. One may think with many predictor variables it is unlikely to omit any important variables that greatly impact the response. However, too many predictor variables are detrimental rather than beneficial to the classification/regression problem. First, many more observations are needed to exercise the space of the predictor variables. Second, the increased computation complexity may make the problem unsolvable. Third, the correlation among the predictor variables may significantly worsen the model performance.

One popular technique that appears in attacking cryptographic hardware and also in pattern matching is principal component analysis (PCA). PCA transforms the original data to a new space spanned by a set of orthogonal vectors, along the first of which the data have the largest variance, along the second the second largest variance, etc. The set of orthogonal

vectors are computed from the singular value decomposition (SVD) of the empirical sample covariance matrix Σ of the original data X[85]:

$$\Sigma = U * \Lambda * U^{-1} \tag{2.18}$$

$$\Sigma = S(X_i, X_j)_{p \times p}, (i = 1, \ldots, p, j = 1, \ldots, p) \tag{2.19}$$

$$S(X_i, X_j) = \frac{1}{n-1} \sum_{k=1}^{n} (x_{ki} - \bar{X}_i)(x_{kj} - \bar{X}_j) \tag{2.20}$$

where $X_i (i = 1, \ldots, p)$ is the i-th vector of X, each X_i has n observations (i.e. X has n rows), $\bar{X}_i$ is the sample mean of X_i. Since Σ is a symmetric matrix, Λ is a diagonal matrix and U is an orthogonal matrix of eigenvectors of Σ. The eigenvector corresponding to the largest eigenvalue is the first principal component, along which the original data have the largest variance; the eigenvector corresponding to the second largest eigenvalue is the direction along which the original data have the second largest variance, etc (see[85]). If we can keep the eigenvectors corresponding to the first $k < p$ largest eigenvalues, then we only need to process k variables that have the majority of variance. The dimension of the variables is therefore reduced. In addition, if some variables in the original matrix are correlated, then some of the eigenvalues are zeros. We can then ignore the directions that correspond to these eigenvalues. The transformed data X' that keep only the first k eigenvectors are computed from:

$$X' = XW \tag{2.21}$$

where W is a $p \times k$ matrix that keeps only the first k columns of U. X' instead of X will be used for further processing. If X' is used for the normal least-square-estimation of a regression model, then the procedure is called principal component regression (PCR), which is a useful technique to eliminate multicollinearity among original variables. Other common use of X' is for pre-processing in pattern matching. The purpose is to reduce dimensionality

while retaining most representative signals, as the transformed data that corresponding to small variances are discarded for further processing. It is in particular useful in template analysis, as the covariance matrix of the input variables needs to be computed.

Chapter 3

Black-box Analysis

[1]

The first device studied to explore feasibility of using side-channels for software integrity checking is a PIC microcontroller (μC). PIC16F687 is chosen as the DUT, because most previous research in side-channel analysis for general programs has been done on this IC.[17;69;72;74]

3.1 Related work

Chapter 2 has already introduced literature on existing software integrity checking approaches that are either internal or external to the target device, as well as side-channel analysis for different purposes. The problem of side-channel-based software integrity checking for microcontrollers is distinct from side-channel analysis in other domains in that

- The analysis is on a single-captured side-channel measurement that represents a one-time execution of some code;
- The analysis is on the entire instruction set instead of a few special instructions;
- The analysis is on the entire trace in one capture instead of a few special occasions in time;

[1]This chapter is based on work already published[86]

- The analysis is on a black-box device the design detail of which is unknown;
- The analysis must consider compact malware that may be composed of a single instruction, or a change of original instruction only on the operands;
- The analysis must consider side-channel-aware attackers who actively attempts to evade detection by computing alternative code that has near indistinguishable side-channel measurement from that of the original code.

Side-channel analysis proposed in this chapter for software integrity checking is an "on-line" verification method in which the runtime state of a device is checked dynamically against desired one via side-channel measurements, so that "transient" malware or faulty states can be detected. This approach is in contrast with detection methods for hardware trojans, which are based on static scanning of the side-channel emissions, such as light, of a circuit.

Previous research on side-channel-based software integrity checking that performs run-time verification is either at a rather coarse granularity (e.g., function-level)[73;75;76;78], or ignorant of side-channel-aware attackers[77;78], or the effects of data, even operands of instructions.[17;18;69–74] Software attestation[42–44;79] utilizes the timing side-channel and is capable of detecting malware which may be as compact as one instruction. However, the device must support such attestation, and carrying out the process requires interruption of the device execution, a particular drawback for legacy and actively-used systems. Furthermore, software attestation cannot detect transient faulty states, code-reuse attacks, data-only attacks, and suffers from the TOCTOU problem (c.f. Section 2.2.1).

As shown in the following sections, the runtime side-channel emanations of a device are not only determined by the instructions, but also, if not more significantly, by data (context) that are involved in the instruction execution, such as register and memory content. More generally, the actual internal activity of the device, instead of solely the code determines side-channel emanations. To guarantee that a single execution of malware is detected, side-channel analysis must be at the granularity of at least instruction cycles, taking into

consideration of both code and data. Previous research often tries to profile side-channels according to instruction operations by using pattern matching/classification methods.[18;69–73;80] We will show in this chapter that such analysis is unlikely to be successful due to the nature of side-channel emissions, and without considering operands and other context, it cannot solve rigorously the integrity checking problem.

3.2 Problem definition

PIC16F687 is an 8-bit RISC μC in Harvard architecture. It has a 14-bit program bus, which is connected to the program flash, and an 8-bit data bus, which is connected to RAM, EEPROM, PORTs, ADC, etc. The instruction set has 35 operations, all executed in single instruction cycle, except branches. Figure 3.1 shows the instruction set and the opcodes. The processor has a two-stage pipeline. Each instruction execution is overlapped with the next instruction fetch. Unconditional and conditional branches take two instruction cycles if a branch is taken. A `NOP` is inserted after the branch instruction and replaces the original instruction following immediately after the branch instruction. The working register is one of the two operands of the ALU. There is a 128-byte register file including general-purpose registers (GPRs) and special function registers (SFRs). Besides this basic information, PIC16F687 is a black-box to the developers as no details on its architecture implementation are available in the public documents.

3.2.1 Threat model

Attacker

We assume that the attacker is able to modify the software of the DUT. The attacker is also able to profile the side-channel emissions of the DUT non-invasively and to modify the software in a fashion that minimizes side-channel deviation from the authentic code. The attacker is however unable to inject faults or modify the hardware, including the IC design

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb … LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff	C, DC, Z	**1, 2**
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff	Z	**1, 2**
CLRF	f	Clear f	1	00 0001 1fff ffff	Z	**2**
CLRW	–	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff	Z	**1, 2**
DECF	f, d	Decrement f	1	00 0011 dfff ffff	Z	**1, 2**
DECFSZ	f, d	Decrement f, Skip if 0	1**(2)**	00 1011 dfff ffff		**1, 2, 3**
INCF	f, d	Increment f	1	00 1010 dfff ffff	Z	**1, 2**
INCFSZ	f, d	Increment f, Skip if 0	1**(2)**	00 1111 dfff ffff		**1, 2, 3**
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	**1, 2**
MOVF	f, d	Move f	1	00 1000 dfff ffff	Z	**1, 2**
MOVWF	f	Move W to f	1	00 0000 1fff ffff		
NOP	–	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	**1, 2**
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	**1, 2**
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff	C, DC, Z	**1, 2**
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff		**1, 2**
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	**1, 2**
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		**1, 2**
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		**1, 2**
BTFSC	f, b	Bit Test f, Skip if Clear	1 **(2)**	01 10bb bfff ffff		**3**
BTFSS	f, b	Bit Test f, Skip if Set	1 **(2)**	01 11bb bfff ffff		**3**
LITERAL AND CONTROL OPERATIONS						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk	C, DC, Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	k	Call Subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	–	Clear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$, $\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	–	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	–	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	–	Go into Standby mode	1	00 0000 0110 0011	$\overline{TO}$, $\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk	C, DC, Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	

Figure 3.1: The instruction set of PIC16F687[2]

and the PCB on which the DUT is mounted. [2] The attacker is not an insider of the chip manufacturer either.

Verifier

We assume that the verifier knows the initial hardware and software configuration of the device. The verifier is able to profile the side-channel emissions of the DUT non-invasively. During the integrity checking process, the verifier cannot interrupt the device execution or interfere with the device in any way that violates the Electromagnetic Compatibility (EMC) requirements (e.g., by removing the noise decoupling capacitor on the board) so that the device will not execute wrongly due to side-channel measurement. The verifier is not an insider of the chip manufacturer either so is only able to perform black-box analysis on the DUT.

3.2.2 Experiment setup

First, the power consumption of the chip is chosen for side-channel analysis. The ground pin of the chip is connected to an 82Ω shunt resistor, as shown in Figure 3.2. Voltage drop across the shunt resistor is captured by a PicoScope 5444B 200MHz USB oscilloscope. The ground pin, instead of the power supply pin, is used due to limitations of the oscilloscope. To mitigate the low-pass filtering effects from the chip itself[1;87], we set the processor frequency to 125 kHz. The sample rate is 31.3 MS/s. Higher frequency suffers more from the low-pass filtering effects and does not work with the oscilloscope. However, the result is repeatable with higher processor frequency, given an oscilloscope with higher bandwidth. The experiment setup is low-cost and reflects a worse-case scenario from the verifier's point of view.

The result of power consumption is later compared with that of EM measurement, as shown in Section 3.4.3.

[2] In general, if an attacker has invasive access to the chip, she can change the chip in any fashion.

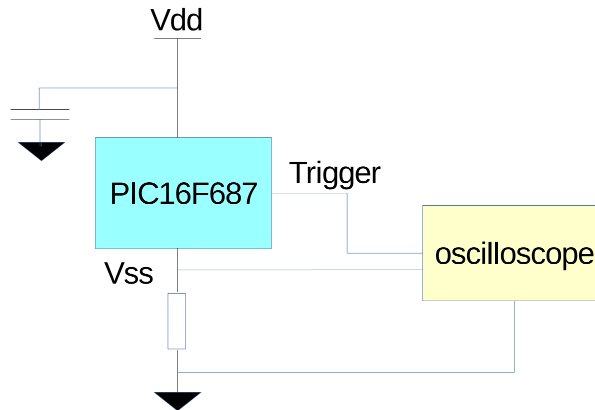


Figure 3.2: Measurement setup

3.3 A systematic approach for instruction-level side-channel analysis

Because there are so many factors that may affect side-channel emanations, an ad hoc experiment will soon become unmanageable. After several attempts, we have developed a systematic approach for black-box side-channel analysis at instruction-level: [3]

- Build semantic models of the instruction set, using known architecture information.
- Generate testing code that is long enough to execute each instruction operation many times with different operands.
- Calculate the internal runtime state according to the semantic model for each instruction.
- Cross-validate side-channel measurements and the semantic models with respect to the predicted runtime states.

[3]The actual model is at (the finer) clock cycle level.

3.3.1 Semantic models

Building semantic models of an instruction set includes elaborating the hypothetical detailed internal activities that happen during an instruction execution, such as fetch, decode, and data read/write. Based on the limited architecture information described in the PIC16F687 datasheet[2;88], we deduce that potential data that may appear on buses, and therefore are likely to cause the major power consumption, include values of the program counter (PC), the operands and opcode of instructions, the working register, the GPRs, and the `STATUS` SFR.

We initially assume the internal state of the device is:

$$T = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type, OPRD1, OPRD2, B, D) \tag{3.1}$$

where

- W is the working register,
- C is the STATUS register,
- F is the GPRs (indexed from 0x40 to 0x7f),
- PC is the program counter,
- I_{prev} is the previous instruction (incl. instruction operation and operands),
- I_{curr} is the current instruction,
- I_{next} is the next instruction,
- $Type$ is the type of the operation of the current instruction,
- $OPRD1$ is the *content* of first operand,
- $OPRD2$ is the *content* of second operand,
- whether a branch is executing $B (B \in \{\text{True}, \text{False}\})$,

- D is the result of the current instruction.

Type is one of the instruction categories we summarize from the instruction set document[2;88]:

- byte-oriented file register operation with the working register as destination, e.g., `ADDWF f,W` (abbr. `wfw`);
- byte-oriented file register operation with the GPR as destination, e.g., `ADDWF f,F`, `CLRF f`, `MOVWF f` (abbr. `wff`);
- the goto operation `GOTO` (abbr. `goto`);
- the call operation `CALL` (abbr. `call`);
- the return operation `RETURN, RETLW` (abbr. `ret`);
- bit-oriented increment/decrement branch operation with the working register as destination `INCFSZ f,W and DECFSZ f,W` (abbr. `fszw`);
- the bit-oriented increment/decrement branch operation with the GPR as destination `INCFSZ f,F and DECFSZ f,F` (abbr. `fszf`);
- the bit-oriented test branch operation `BTFSC and BTFSS` (abbr. `btfs`);
- the bit-oriented set/clear operation `BCF and BSF` (abbr. `bxf`);
- the literal operation, e.g., `ADDLW k` (abbr. `lw`);
- the nop operation `NOP` (abbr. `nop`);
- the literal clear operation `CLRW` (abbr. `clrw`); and
- the inserted nop operation when branch is taken (abbr. `brnop`).

We then build the hypothetical semantic model for each instruction and compute the associated internal activity. For example,

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{ADDWF f,W}}$$

$$T_1 = (W' = mod(W + (f), 256), C', F, PC + 1, I_{curr}, I_{next}, I_{next-next}, \text{wfw}, W, (f), \text{False}, W')$$

which means, if current instruction is not skipped, then after executing `ADDWF f,W`, the internal state of the device will change from T_0 to T_1, with (1) the working register is updated with the truncated value of $(W+(f))$, where (f) is the content of the GPR f; (2) the `STATUS` register is updated according to the result of $(W+(f))$; (3) the GPRs remain the same; (4) the program counter increments; (5) next instruction is read; (6) the type of the operation is updated to `wfw`; (7) the first operand is the working register; (8) the second operand is the content of the GPR f; (9) no branch happens; and (10) the result of the operation is W'. The state transition of `ADDWF f,F` can be defined similarly, with the exception that the result is stored back to f instead of W. In this way we build the hypothetical semantic models for `ADDWF f,d`, `ANDWF f,d`, `COMF f,d`, `DECF f,d`, `INCF f,d`, `IORWF f,d`, `RLF f,d`, `RRF f,d`, `SUBWF f,d`, `SWAPF f,d`, `XORWF f,d`, where `d` is `W` or `F` indicating whether the result is stored to the working register or the GPR. Note that for `SUBWF f,d`, the result is the truncated value of $((f)-W)$. One of the operands can be both W or the two's complement of W. We have tried both cases in further processing. After cross-validation, we find that the original W is used.

The hypothetical semantic model for `CLRF f` is defined as:

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{CLRF f}}$$

$$T_1 = (W, C', F', PC + 1, I_{curr}, I_{next}, I_{next-next}, \text{wff}, 0, \text{NA},$$

$$\text{False}, 0)$$

where the `STATUS` register is updated with `Z` set to 1 and the content of GPR f of F is set to 0. NA means the value is not available and/or irrelevant.

The hypothetical semantic model for `CLRW` is defined as:

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{CLRW}}$$

$$T_1 = (0, C', F, PC + 1, I_{curr}, I_{next}, I_{next-next}, \text{wfw}, 0, \text{NA},$$

$$\text{False}, 0)$$

where the `STATUS` register is updated with `Z` set to 1 and W is set to 0.

The hypothetical semantic model for `MOVWF f` is defined as:

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{MOVWF f}}$$

$$T_1 = (W, C, F', PC + 1, I_{curr}, I_{next}, I_{next-next}, \text{wff}, W, (f),$$

$$\text{False}, W)$$

where the content of GPR f of F is set to W, the `STATUS` register is not affected, and the

second operand is (f), although the value of f is not used.

For conditional branch instructions of type `fszw`, take the example of `INCFSZ f,W`:

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{INCFSZ f,W}}$$

$$T_1 = (W' = mod((f)+1, 256), C, F, PC+1, I_{curr}, I_{next}, I_{nextnext}, \text{fszw}, (f), \text{NA}, \begin{cases} \text{False} & \text{if } W' \neq 0 \\ \text{True} & \text{if } W' = 0 \end{cases}, W')$$

which means, if current instruction is not skipped, after executing `INCFSZ f,W`, the internal state of the device will change from T_0 to T_1, with (1) the working register is updated with the truncated value of $((f)+1)$, where (f) is the content of the GPR f; (2) the `STATUS` register is *not* affected; (3) the GPRs remain the same; (4) the program counter increments (see below); (5) next instruction is read (see below); (6) the type of the operation is updated to `fszw`; (7) the first operand is the content of the GPR f; (8) the second operand is not available; (9) branch (skip) will happen if $W' = 0$; and (10) the result of the operation is W'.

The semantic model of conditional branches of type `fszw`, `fszf`, `btfs` are defined similarly.

For instructions of type `bxf`, take the example of `BSF f,b`:

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{BSF f,b}}$$

$$T_1 = (W, C, F', PC + 1, I_{curr}, I_{next}, I_{next-next}, \text{bxf}, (f), b,$$

$$\text{False}, (f)|(1 << b))$$

which means, if current instruction is not skipped, the content of GPR f is updated with the b-bit set and the `STATUS` register is not affected.

For the literal operation of type `lw`, take the example of `ADDLW k`:

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{ADDLW k}}$$

$$T_1 = (W' = mod(W + k, 256), C', F, PC + 1, I_{curr}, I_{next}, I_{next-next}, \text{bxf}, W, k,$$

$$\text{False}, W')$$

The semantic models for other literal operations are similarly defined. Note that for `SUBLW k`, the result is the truncated value of $(k - W)$. One of the operands can be both W or the two's complement of W. We have tried both cases in further processing. After cross-validation, we find that the original W is used.

For the `NOP` operation,

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{False}, D)$$

$$\xrightarrow{\text{NOP}}$$

$$T_1 = (W, C, F, PC + 1, I_{curr}, I_{next}, I_{next-next}, \text{nop}, W, 0,$$

$$\text{False}, W)$$

which means, if current instruction is not skipped, `ADDLW 0` is executed with the exception that no `STATUS` flags are affected.

Now we consider the situation in which the current instruction is skipped, for example when the previous instruction is a conditional branch and branch result is `True`, or when the previous instruction is an unconditional branch such as `GOTO`:

$$T_0 = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, Type0, OPRD1, OPRD2, \text{True}, D)$$

$$\xrightarrow{\text{any instruction}}$$

$$T_1 = (W, C, F, PC + 1, \text{NOP}, I_{next}, I_{next-next}, \text{brnop}, W, 0,$$

$$\text{False}, W)$$

which means, if current instruction *is* skipped, then a `NOP` operation is executed instead of the original instruction.

Because the architecture information is not complete, our semantic models are only conjectures, which can be cross-validated with the side-channel measurements. This is multipurpose: first, it is necessary for predicting branches during code generation; second, analyzing the measurements with respect to the runtime state reveals effects of data versus those of processing; third, waveform can also be checked against the predicted runtime state in order to guarantee that the chip functions correctly. The measurement equipment will inevitably introduce interference to the DUT. For example, using a large shunt resistor for power con-

sumption measurement eases the experiment by eliminating the amplification circuits which requires additional power supply and increases cost. However, the shunt resistor also introduces common impedance coupling and narrows the voltage drops between V_{DD} and V_{SS}, which may cause the device to malfunction. Comparing the waveform against the predicted state helps to choose the right resistor value.

We generate random assembly code traces and calculate internal activities from the hypothetical semantic models. Random code is used instead of real code to evenly sample the code space, and avoid overfitting to any specific code base. There is a potential risk that high-order side-channel characteristics which exists only among some particular instruction pairs/blocks will be averaged out when using random code. While we may lose some information for particular instruction blocks, side-channel properties that are applicable to arbitrary programs still retain. Once the first-order characteristics is discovered, we can continue to analyze high-order properties which may or may not exist.

3.4 Side-channel models

The target μC PIC16F687 has 2kB memory. We therefore generate around 1400 random instructions each time. To have enough samples per operation, file register access is limited to 12 general-purpose registers and the `STATUS` SFR. `CALL`, `RETFIE`, `RETLW`, and `RETURN` are manually inserted in multiple places so that the program can execute normally. `SLEEP` (put device to standby mode) and `CLRWDT` (clear watchdog timer) are excluded. One thousand power traces are collected for each program, among which 50% are used for modeling and 50% are used for testing. A typical waveform is shown in Figure 3.3. PIC16F687 has an instruction cycle of four clock cycles, denoted as Q1 to Q4. The waveform exhibits sharp peaks at clock rising/falling edges, showing that the low-pass effects are not prominent with our experiment setup. We observe prominent clock shifting in the waveform, and therefore align samples for each clock cycle according to the peak values at clock rising edges.

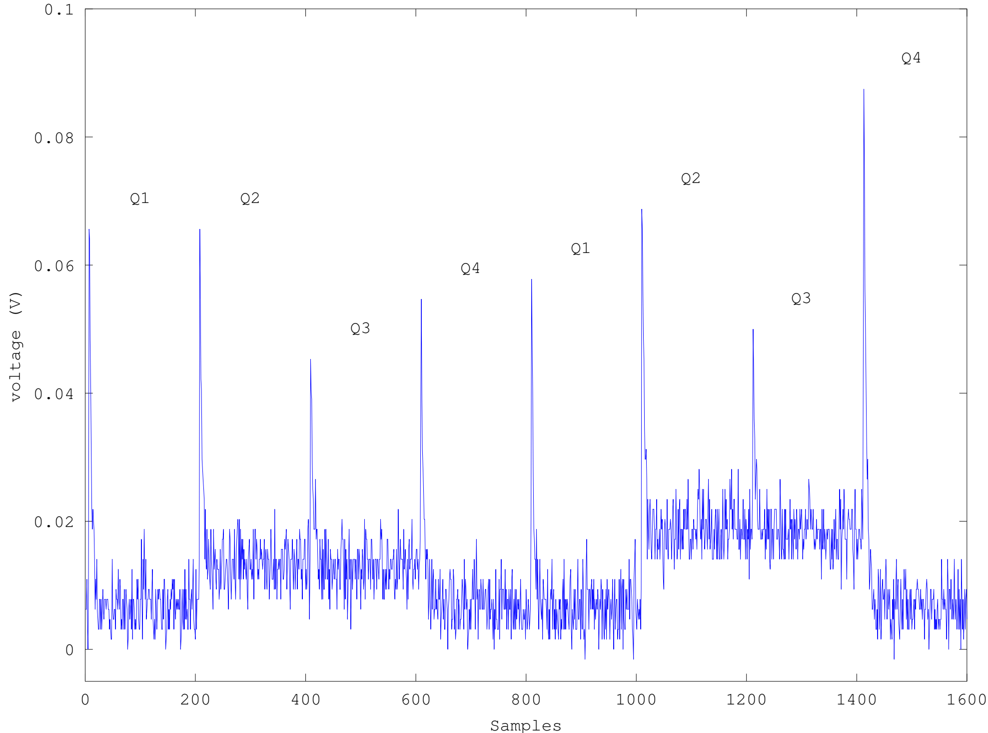


Figure 3.3: A single-captured waveform of executing "`MOVLW 0x69`" and "`ADDWF 0x40,F`"

3.4.1 Side-channel models of instruction operations

We first try to analyze the side-channel measurement with respect to the instruction operations, as done in previous research.[17;18;69;72;74] The problem can be formalized as: given a single trace of power samples of four clock cycles, the verifier tries to recognize one out of 33 instruction operations based on the profiling model – a typical pattern recognition/classification problem. We have applied various classifiers, including the naive Bayes, k-nearest neighbor (kNN), SVM, Multilayer Perceptron, and template analysis. Power samples are with/without feature selection by PCA, mutual information, and linear discriminant analysis (LDA). The best recognition rate is obtained by using template analysis.[69;81] The power consumption is approximated as multi-variate Gaussian signals. One template is built for each instruction operation ω_i (e.g., `ADDWF`, `BTSSC`). For branches, only the first instruction cycle is modeled, and the second instruction cycle of inserted `NOP` when branch is taken is modeled separately. When selecting l samples in one instruction cycle for modeling, the templates are l-dimensional Gaussian distributions with parameters estimated from power

consumption observations when executing ω_i (repeating Equation 2.5 for readability):

$$
\begin{aligned}
p(\boldsymbol{x}|\omega_i) &= \frac{1}{(2\pi)^{l/2}|\boldsymbol{\Sigma}_i|^{1/2}} \exp\Big(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i)\Big) \\
\boldsymbol{\mu}_i &= \frac{1}{N_i}\sum_{j=1}^{N_i} \boldsymbol{x}_{ij} \\
\boldsymbol{\Sigma}_i &= \frac{1}{N_i - 1}\sum_{j=1}^{N_i} (\boldsymbol{x}_{ij}-\boldsymbol{\mu}_i)(\boldsymbol{x}_{ij}-\boldsymbol{\mu}_i)^T
\end{aligned}
$$

where $\boldsymbol{x}_{ij}$ is an l-dimensional observation of executing operation ω_i in the modeling data, N_i is the number of such observations in the modeling data. When given a new observation $\boldsymbol{x}$, the instruction operation is estimated by applying the Bayes rule, which is the ω_i that gives the maximum a posteriori probability.

$$
\hat{\omega} = \underset{\omega_i}{\operatorname{argmax}}\, p(\omega_i|\boldsymbol{x}) = \underset{\omega_i}{\operatorname{argmax}}\, p(\boldsymbol{x}|\omega_i)P(\omega_i)
$$

For integrity checking, the a priori distribution $P(\omega_i)$ is meaningless, since the verifier is unlikely to know with which instruction the attacker may use to replace the original code. We therefore assume the a priori distribution is uniform, thus reduce Bayes rule to the maximum likelihood criterion.

$$
\hat{\omega} = \underset{\omega_i}{\operatorname{argmax}}\, p(\boldsymbol{x}|\omega_i) \tag{3.2}
$$

One template is built for each operation. For file-register operations, each template is built for writing to the file/working register. In total, 47 templates are built. The resulting average recognition rate is 45.6%, which is comparable to unoptimized results of[69;81] and the single-location result of[17]. While some operations still have acceptable recognition rates, such as `CLRW` (99.0% recognition rate), `GOTO` (97.8%), and `COMF f,F` (95.7%), other operations, such as `CLRF`, `DECFSZ f,W` and `IORWF f,F`, are almost always misclassified.

To explore the sources of recognition errors, we perform the same template analysis

but now build one template for each instance of instruction execution. The models thus incorporate power consumption caused by execution with different operands and runtime state. For the same data, we build 1435 templates. Applying again the maximum likelihood criterion, the average recognition rate is surprisingly 99.90%, in contrast with 0.0678% for random guess. This result however cannot be directly used for model building because this implies every combination of instruction and data must be exercised, which is impractical even for an 8-bit processor. We must try a different strategy to solve the problem.

Separability

The difference in recognition rates can be explained by the separability of templates. The Bhattacharyya distance of power consumption grouped by instruction operations is computed (c.f. Section 2.4.1) to measure class separability.

Building templates for instances of instruction execution, the 30 errors in 30,000 tests correspond to 4 out of 1,028,895 pairs that have the smallest Bhattacharyya distances (from 3.98 to 11.45), showing that the multi-variate Gaussian models are good approximators of the signals. In contrast, for templates of instruction operations, the Bhattacharyya distances of the majority of template pairs, especially logic and arithmetic operations, are near zero, corresponding to recognition rates near to those of random guess.

3.4.2 Side-channel models of internal states

Previous modeling attempts imply that the side-channel emanations are related not only with the instruction operation, but also, probably more strongly, with other internal variables during instruction execution such as operands and the content of operands. To discover the effects of runtime states, we change testing programs by modifying only the initial values of registers. Because register values affect results of conditional branches, code near conditional branches is adjusted, so that only the instruction immediate after each conditional branch test is different, while majority of instruction execution stays the same. We rerun the experiment and record the measurements of the (nearly) same program but with different

data. The difference between the two resulting measurements is shown in Figure 3.4.

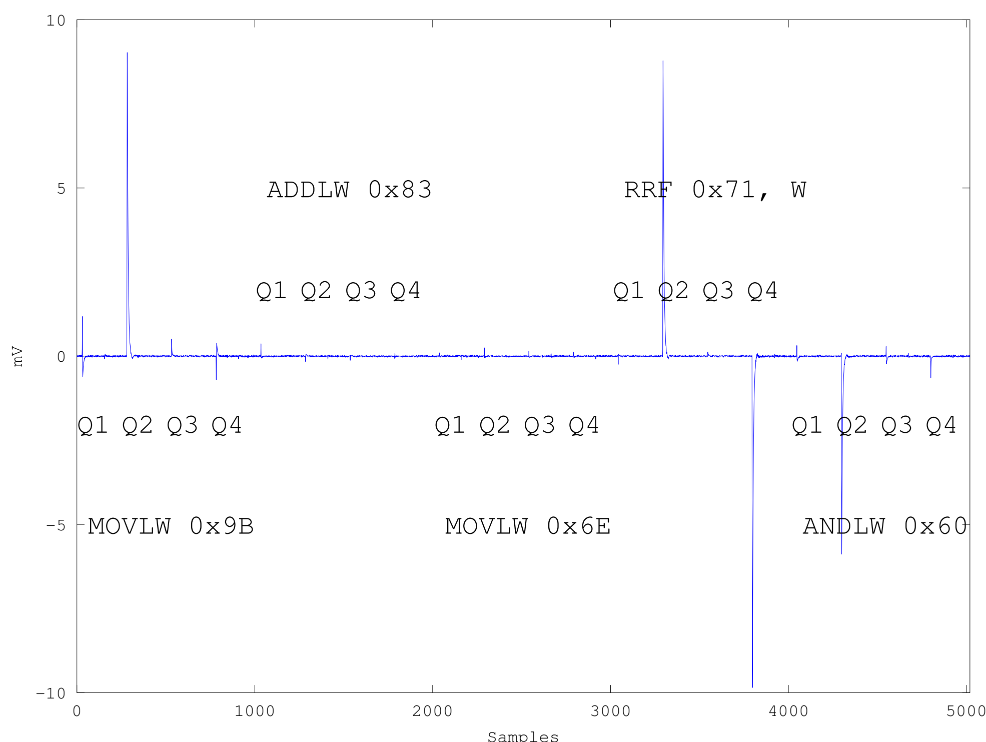


Figure 3.4: Difference of two measurements with the same program but different register values.

The measurements of "`MOVLW 0x9B`" have significant difference at the edge of Q2. After executing "`MOVLW 0x9B`", the measurements of "`ADDLW 0x83`" and "`MOVLW 0x6E`" are nearly identical. Executing "`RRF 0x71,W`" differs at Q2 and Q4, whereas executing "`ANDLW 0x60`" has significant difference at Q2 and slight difference at Q4. Q1 and Q3 are on the other hand almost the same at all time. This phenomenon coincides with the architecture description in[2]: for instruction execution, instruction is latched in Q1, data memory is read in Q2 (operand read), data is processed in Q3, and in Q4 data memory is written (destination write). After executing "`MOVLW 0x9B`", the working register and the `STATUS` register are the same, [4] and we deduce that the traffic on the data bus during operand read and destination write is therefore the same, which leads to the same side-channel measurements. The contents of the file register `0x71` are different, which results in different traffic on the data bus and accordingly different measurements at Q2 and Q4. The result of "`RRF 0x71,W`" is written

[4]The `STATUS` register is affected by previous code that is not shown.

to the working register, and thus causes further differences at Q2 and Q4 when executing "`ANDLW 0x60`". On the other hand, Q1 and Q3 do not show significant difference, even though the manual claims that data are processed in Q3.

We therefore turn to another distinct strategy for side-channel modeling. Instead of building templates from side-channel measurements, we try to find regression functions which can reveal the linkage between internal activities and side-channel measurements. Note that in pattern matching/classification, the side-channel measurement is the predictor or independent variable x, and the resulting class is the response y, whereas in regression analysis, the predictor is internal activity, and the response is side-channel measurement.

Let runtime state at time t be a vector of random variables $\vec{T}_t$, we assume the power consumption at t is a random variable Y_t and depend on $\vec{T}_t$:

$$Y_t = f_q(\vec{T}_t) + N_t \tag{3.3}$$

where N_t encloses remaining components in the power consumption at time t including time-dependent components and noise, as in[89]. N_t and Y_t are random variables and T_t is a result of executing instructions and therefore controllable. $f_q(q = 1, 2, \ldots)$ indicates that there might be a set of functions for different stages of instruction execution.

We use the variables of T in the hypothetical semantic model to regress with the power consumption. In addition, we also include the Hamming distance (HD) and the Hamming weight (HW) of the variables in the predictors. The HD counts the number of bit differences between two binary values, and the HW counts the number of 1s of a binary value. We have intentionally added many variables in the semantic model which are probably more than necessary (not all shown in Section 3.3.1). This does not affect regression in the sense that adding more predictor variables will always give smaller mean square errors (MSE).[90] Unnecessary predictors can be pruned afterwards, by using for example t-test of the regression coefficient for individual predictor.

It turns out that there are strong linear relationships between runtime internal state and power consumption measurements. For any instruction cycle t, there are four regression

functions, corresponding to four execution stages:

$$Y_q = \vec{T}_t \vec{\beta_q} + b_q + N$$

where $q \in [1, 4]$ indicates the stage, $\vec{\beta_q}$ is a vector of weights (regression coefficients) and b_q is a constant. The noise component N is assumed to be independent of time and stage.

The actual regression models are built for individual instruction categories. We use the Pearson's correlation coefficient and the Spearman's correlation coefficient to measure the performance of the regression models. For two random variables $X = \vec{T}_t \vec{\beta_q}$ and Y, the Pearson correlation coefficient is a measure of linear dependence between X and Y:

$$r = \frac{cov(X, Y)}{\sigma_X \sigma_Y} = \frac{\sigma_X}{\sqrt{\sigma_X^2 + \sigma_b^2}} \tag{3.4}$$

where the covariance of the two random variables are estimated by using the empirical sample variance. r tends to ± 1 as σ_b^2 tends to 0. Spearman's rank correlation is the Pearson correlation between weakly-ordered values. Spearman's correlation is able to measure non-linear relationships between two variables. If the two variables are linearly related, then the two correlations are identical. Spearman's correlation is more sensitive to outliers.

Regression analysis shows that the power consumption of the chip can be accurately modeled using linear functions of internal activities. Several interesting phenomena can be observed.

First, the HD of PC and (PC+1) influences the peak amplitude in Q1, regardless of the operation of the instruction. This corresponds to the fact that the pipeline depth of the DUT is two: each instruction execution is overlapped with fetching of the next instruction, and the PC increments in Q1 for instruction fetch.

Second, the content of data already on some internal data bus (which is the result of previous instruction) and the operand loaded for current instruction influence the peak in Q2. In Q2, different types of operations (c.f. Section 3.3.1) will load different operands. And the peak amplitude in Q2 is linearly related to the HD of the data on some internal bus and the operand. Table 3.1 shows the resulting regression models for different types of

operations.

Table 3.1: Regression analysis of power consumption in Q2

	File register operations	
Predictors	$HD(\text{previous result}, (f))$	Constant
Coefficients	2.88	-15.30
r	0.97	
rho	0.97	
	Literal operations	
Predictors	$HD(\text{previous result}, operand)$	Constant
Coefficients	2.86	-19.34
r	0.92	
rho	0.92	
	`SUBLW`	
Predictors	$HD(\text{previous result}, operand)$	Constant
Coefficients	1.73	-17.99
r	0.95	
rho	0.88	
	`NOP`	
Predictors	$HD(\text{previous result}, 0)$	Constant
Coefficients	2.49	-19.63
r	0.90	
rho	0.90	
	`GOTO`	
Predictors	$HD(\text{previous result}, operand)$	Constant
Coefficients	2.38	-22.09
r	0.92	
rho	0.89	

The file register operations in Table 3.1 include all the other operations not listed in the following entries, including `ADDWF`, `INCF`, `INCFSZ`, `BSF`, `BTFSC`, etc. The literal operations in Table 3.1 include `ADDLW`, `ANDLW`, `IORLW`, `XORLW`, and `MOVLW`. The table shows that overall, the peak amplitude of power consumption in Q2 is proportional to the Hamming distance of the value already on the bus, and the data loaded for current execution:

$$Y_{q2} = \beta * HD(\text{previous result, data for current}) + b$$

where Y_{q2} is in mV, same unit for the following Ys.

There are some interesting discoveries:

- For bit-oriented file register operations and byte-oriented file register operations such as `ADDWF`, `RRF`, `INCF`, `INCFSZ`, and `BTFSC`, the content of the file register is loaded, regardless of whether the instruction is a conditional branch or not;
- For operations including `CLRF`, `CLRW`, `MOVWF`, the content file register is still loaded even if its content is not useful;
- `CLRW` is actually implemented as `CLRF 0x7f,W` – that is, the content of GPR 0x7f is loaded in Q2;
- The target address, i.e., the operand of `GOTO` is loaded in Q2;
- The literal, i.e., the operand of literal operations such as `ADDLW` is loaded in Q2; The operand of `SUBLW`, not its two's complement, is loaded;
- `NOP` is implemented as `ADDLW 0`, except that no `STATUS` flags will be affected. The value zero is therefore loaded in Q2.

The plateaus following the peaks in Q2 and Q3, are linear to the Hamming weight (HW) of the next instruction, *regardless of instruction operations*:

$$Y_{plateaus} = 0.836 * HW(I_{next}) - 14.41$$

with $r = 1.00$ and $\rho = 0.99$. Note that the next instruction is still the instruction immediately following the current instruction even if a branch will be taken after execution (e.g., when the current instruction is `GOTO` or `BTFSS f,b` and bit `b` of GPR `f` is 1).

The peak amplitude in Q3 is linear to the Hamming weight of next instruction and the Hamming weight of current instruction, *regardless of instruction operations*:

$$Y_{q3} = 1.32 * HW(I_{curr}) + 0.828 * HW(I_{next}) - 31.57$$

with $r = 1.00$ and $\rho = 1.00$. Note that the current instruction is `NOP` if current instruction cycle is an inserted cycle after a branch is taken. The $HW(I_{curr})$ is therefore zero, regardless of neighboring instructions. So I_{curr} of the current instruction cycle is not necessarily equal to I_{next} of the previous instruction cycle.

The regression analysis of the peak amplitude in Q4 is shown in Table 3.2. Overall, the peak amplitude in Q4 is linear to the Hamming distance between data loaded in Q2 and the result of the current instruction, and the Hamming weight of the next instruction:

Table 3.2: Regression analysis of power consumption in Q4

	W as destination		
Predictors	HD(data loaded in q2, result)	$HW(I_{next})$	Constant
Coefficients	2.93	2.15	-25.09
r	0.99		
rho	0.99		
	f as destination		
Predictors	HD(data loaded in q2, result)	$HW(I_{next})$	Constant
Coefficients	3.60	2.15	-23.78
r	1.00		
rho	1.00		

The operations with the working register as destination in Table 3.2 include: (1) literal operations such as `ADDLW`, `NOP`; (2) the `NOP` inserted after branch; (3) the unconditional branch `GOTO`; (4) conditional branches with the working register as destination, such as `DECFSZ f,W` and `BTFSC f,b`; (5) `CLRW`; and (6) file register operations with the working register as destination, such as `ADDWF f,W`. The operations with the GPR as destination include: (1) conditional branches with the file register as destination, such as `DECFSZ f,F`; (2) file register operations with the file register as destination, such as `ADDWF f,F`; and (3) `BCF f,b` and `BSF f,b`.

There are also some interesting discoveries:

- The correlation coefficients rs and $rhos$ are very close, if not equal, to one, meaning that the linear relationship between the internal state and power consumption measurements is very strong;

- Operations with the file register as destination consume more power than those with the working register as destination;

- The resulting data of `MOVLW` is the new W, which has the same value with the operand of the instruction; the Hamming distance is therefore always zero;

- The resulting data of `GOTO` is still the operand (i.e., branch address); the Hamming distance is therefore always zero; the next instruction of `GOTO` is the immediate instruction followed, and is not the instruction at the goto address;

- The resulting data of conditional branches, `BTFSC f,b` and `BTFSS f,b`, are always zero, regardless of whether branch is to happen or not; the Hamming distance is therefore $HD((f), 0) \equiv HW((f))$;

- The resulting data of conditional branches `DECFSZ f,d`, and `INCFSZ f,d` are, in contrast, the incremented or decremented (f) values; the Hamming distance is therefore $HD((f), mod((f) + 1, 256))$;

- The resulting data of the nop inserted after a branch is taken is W, as if a normal `NOP` is executed; the next instruction of the inserted nop is the branch destination;

- The resulting data of literal operations, such `ADDLW` and `NOP`, is the new W; the Hamming distance for `NOP` is therefore equal to $HW(W)$;

- The file register operations `COMF f,d` and `MOVF f,d` always have a constant Hamming distance, regardless of the destination `d`; the Hamming distance for `COMF f,d` is always eight, and for `MOVF` is always zero.

The regression analysis also explains the findings of previous template analysis. `COMF f,F` has a 95.7% recognition rate which is much higher than other operations. This can

be explained by the high power consumption cause by the largest Hamming distance value (eight) in Q4, and also by the destination being a file register. `COMF f,F` therefore has the highest power consumption in Q4 on average. `CLRW` has a 99.0% recognition rate, which is likely to be caused by the Hamming weight of its instruction being always one.

Double-checking. To increase the potential SNR of operation-related signals, we generate testing programs composed of instructions of the same Hamming weight. Except `GOTO` and instruction types that cannot have the target Hamming weight (e.g. `NOP` and `CLRW`), all logic and arithmetic operations are included. Repeating the experiment, we find that previous conclusions on the linear models and data dependencies still hold. Measurements in Q3 have nearly the same value, which can be shown by the small standard deviations (σ) among peak amplitudes. For execution instances, the maximum σ, occurring at the peak of Q3, is 0.324 mV, in contrast with the maximum σ in previous experiments, which is 4.193. For instruction operations, the maximum σ is 0.121, in contrast with the maximum σ in previous experiments, which is 2.495. This implies that Q3 does not yield sufficient margins for classification. Applying various pattern recognition techniques, the best average recognition rate is 33.16% for instruction operations, obtained by SVM with polynomial kernel, five-fold cross-validation. The recognition rate is still much worse than that obtained by template analysis for instruction execution instances, which is 99.53%.

The relationships reveal several valuable sources of side-channel leakage that can be utilized for different verification purposes. First, they reveal that side-channel measurements have strong dependencies on data and weak dependencies on instruction operations. This explains why templates of logic and arithmetic operations have small Bhattacharyya distances: they have small differences in the Hamming weights of their opcode spaces and the distributions of operands and results (except for `COMF`, whose Q4 always has large power consumption since the HD(old value of file register, new value of working/file register) is always eight). While not helping in template analysis with respect to instruction operations, data dependencies in peaks of Q2 and Q4 help to match data values with operations. Second, the strong linear relationships also help to validate our semantic models. Third, the dependency

in instruction opcodes through Q2 to Q4 leaks information about the control flow. While not directly revealing the neighboring instructions, this helps to identify certain instructions such as `NOP` (having the unique zero opcode) and the `NOP` executed after each branch. Fourth, the regression coefficients are in the unit of mV per bit, and are large enough to be resilient to measurement noise.

3.4.3 EM measurement

We repeat the same experiment by using EM measurement. The EM radiation of the chip is captured by a hand-made loop probe, which is the same EM probe used in Section 5.4. The loop probe is place over the power line from the V_{SS} pin which also forms a loop (with a opening) to the power supply. The magnetic field generated by the power line will cause voltage drop in the loop probe, which is amplified by a 20 dB amplifier and then sampled by an oscilloscope, both same to the one used in Section 5.4. An example single-captured waveform of EM radiation of the chip is shown in Figure 3.5.

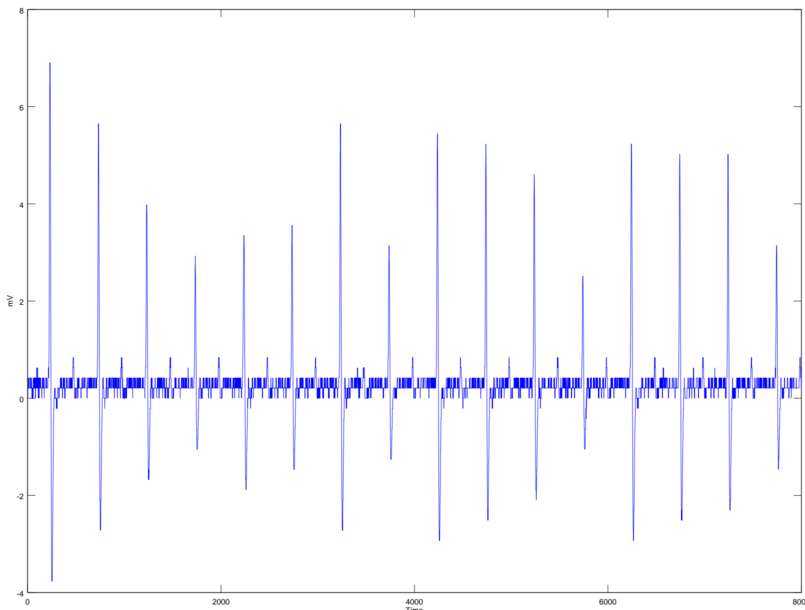


Figure 3.5: A typical single-captured EM radiation of the target chip.

Although the waveform of EM radiation appears to be different from that of power consumption, it is interesting to observe that regression analysis still works well for profiling the EM radiation. The peak amplitude at each clock rising/falling edge is linearly related with the internal activities of the chip. The resulting regression models for individual stages are slightly different from those of power consumption. However the linear relationships between EM radiation and the Hamming distances of the data loaded for execution are still prominent, as shown in Table 3.3 and 3.4.

Table 3.3: Regression analysis of EM radiation in Q2

	File register operations	
Predictors	$HD(\text{previous result}, (f))$	Constant
Coefficients	3.46	37.08
r	0.96	
rho	0.96	
	Literal operations	
Predictors	$HD(\text{previous result}, operand)$	Constant
Coefficients	3.62	32.09
r	0.91	
rho	0.91	
	`SUBLW`	
Predictors	$HD(\text{previous result}, operand)$	Constant
Coefficients	2.24	33.85
r	0.94	
rho	0.88	
	`NOP`	
Predictors	$HD(\text{previous result}, 0)$	Constant
Coefficients	3.12	31.80
r	0.89	
rho	0.89	
	`GOTO`	
Predictors	$HD(\text{previous result}, operand)$	Constant
Coefficients	3.09	27.69
r	0.91	
rho	0.88	

Table 3.3 shows that in Q2, the linear relationship between the side-channel measurement and the Hamming distance of previous result and data loaded for current instruction still holds. The only difference is the values of regression coefficients and the model performance rs and rhos of the EM models are slightly smaller than those of the power consumption models.

It can be observed from Figure 3.5 that there is no plateaus following the peaks at Q2 and Q3, which is not surprising since EM measurement will filter out near DC components.

The peak amplitude of EM measurement at Q3 is linear to the Hamming weight of current instruction, *regardless of instruction operations*:

$$Y_{q3} = 1.48 * HW(I_{curr}) + 18.32$$

with $r = 0.97$ and $\rho = 0.96$. Note that the EM measurement in Q3 is not related with the Hamming weight of next instruction, unlike the case in power consumption.

Table 3.4: Regression analysis of EM radiation in Q4

	W as destination		
Predictors	$HD(\text{data loaded in q2, result})$	$HW(I_{next})$	Constant
Coefficients	3.33	0.90	25.63
r	0.91		
rho	0.90		
	f as destination		
Predictors	$HD(\text{data loaded in q2, result})$	$HW(I_{next})$	Constant
Coefficients	4.42	0.87	27.38
r	0.99		
rho	0.99		

Table 3.4 shows again that in Q4, the linear relationship between the side-channel measurement and the Hamming distance of data loaded for current instruction and the resulting data still holds. The only difference is the value of regression coefficients and the model performance r and rho of the EM models are slightly smaller than those of the power consumption model, as the case in Q2.

Chapter 4

Side-channel Programming

[1]

Experiments in Chapter 3 show that the side-channel emanations of a PIC16F687 μC, both power consumption and EM radiation, are determined by a few internal data transitions.[2] This implies that side-channel profiling according to instruction operations by using general classifiers is unlikely to be successful, because of the weak linkage between operations and side-channel emissions. Regression model can instead provide highly accurate models. Profiling the side-channel emanations of the instruction set is just the first step, now we aim to design a software integrity checking scheme based on the discoveries of the relationships between side-channels and internal activities.

Now the major obstacle is that, although there are very strong linear relationships between waveform and data read in Q2 and destination write in Q4, it is the Hamming distance rather than the exact data that is involved. For a side-channel-aware attacker, it is easy to compute data pairs that have the same HD with previous data on the bus in Q2, go through different operations, and again have the same HD with the operands in Q4, thus evading side-channel-based checking. This is feasible even when considering the Hamming weight relationships through Q2 to Q4, since the coding of instructions is quite compact.

The good news is that a change in data may have cascading effects: in order to tamper

[1] Part of this chapter is based on work already published.[86]

[2] This is likely to be caused by an internal data bus of large load capacitance.

with data in one instruction, previous and next instructions must be modified accordingly. The developers of the μC may utilize the linear relationships between waveform and HW and HD to guarantee tamper detection, not on the code but on the internal states. Given an initial state T_0 and a desired final state T_1, if the developer can find a trace of side-channel measurements $\{Q2_i, Q3_i, Q4_i\}, i = 1, \ldots, n$ that guarantees the transition from T_0 in n steps must end with T_1, then the device must function as desired. Any tampering with the program can either be detected from side-channel measurements, or lead to the same resulting state.

4.1 Side-channel constrained transitions

Since we can identify the unconditional branch operations `CALL`, `RETURN`, `RETFIE`, and `GOTO` separately by template analysis and also by the `NOP` instruction that always follows, we only need to consider the literal operations, the file-register operations that perform arithmetic and logic computation, and the conditional branch operations, for which template analysis fails and we instead have discovered the linear relationships with the side-channel measurements and data processing in Q2, Q3, and Q4. This includes 28 operations. For conditional branches, both branch and no branch are considered.

Since there is no ways to distinguish operations on file registers of the same HW (e.g., 0x41 and 0x50), we define the *distinguishable* runtime state S as:

$$S = (W, C, FS, D, B) \tag{4.1}$$

where

- W is the working register,
- C is the `STATUS` register,
- $FS = fs_{wi} | wi = 1, \ldots, 7$ is a set of sets of GPRs grouped by HWs, i.e., the 64 GPRs (index i is from 0x40 to 0x7f) are divided into seven sets fs_{wi} according to the HW

of index $wi = HW(i)$; the GPRs of the same wi are in one set and therefore do not distinguish index order. For example, fs_1 and fs_7 contains only one member (i.e., GPR 0x40 and 0x7f, respectively).

- D is the result of current instruction,
- whether a branch is to execute $B (B \in \{\text{True,False}\})$,

Note the difference in the definition of the internal runtime state here and the definition of the hypothetical internal state for semantic modeling in Section 3.3.1.

We use the regression models of power consumption for further discussion. Similar analysis can be applied for EM radiation. Since the side-channel measurement is determined only by a few internal data transitions, for each instruction cycle we only need to consider three values: (1) $q2 = HD(\text{last result}, \text{data loaded for current operation})$, (2) $q3 = HW(\text{current instruction})$, and (3) $q4 = HD(\text{data loaded for current operation}, \text{new result})$. $q2 \in [0, 8]$ corresponds to the power measurement in Q2 of current instruction cycle, $q3 \in [0, 14]$ corresponds to the power measurement both in Q3 of current instruction cycle and Q2 through Q4 of previous instruction cycle, and $q4 \in [0, 10]$ corresponds to the power measurement in Q4 of current instruction cycle, which is adjusted from $[0, 8]$ as instructions with the file register as destination consumes more power than instructions with the working register as destination (c.f. Table 3.2).

For a given side-channel constraint of $(q2, q3, q4)$, we exhaustively compute all the possible instructions and resulting runtime states. For efficient computation, all the possible runtime states for any instruction cycle is stored as a search tree T keyed by the runtime state S. The record V of each node of key S is a set of previous state S_{0i} and instructions P_{0i}: $V = \{(S_{01}, P_{01}), \ldots, (S_{0k}, P_{0k})\}$ that result in S. The skeleton of the algorithm that computes a series of possible runtime states, given a trace of side-channel constraints, is shown from Algorithm 1 to 5.

Based on Algorithms 1, it is possible to compute all the possible final state T_k given the initial state T_0 and a series of side-channel constraints $\{q_1, q_2, \ldots, q_k\}$, $q = (q2, q3, q4)$ for

Data: $q2$, $q3$, $q4$, $T_0 = \{(S_0, null)\}$
Result: T_1

```
onecycle(q2, q3, q4, T0)
begin
    T1 ⟵ ∅
    for each node (S,V), S = (W,C,F,D,B) in T0 do
        if this cycle is branch (i.e., nop executed) then
            if q2 = HW(D) and q3 < 15 and q4 = HW(W) then
                compute resulting S1 of executing NOP
                addnode(S1, S, branchNOP, T1)
        else
            procliteral(q2, q3, q4, S, T1)
            procbyte(q2, q3, q4, S, T1)
            procbit(q2, q3, q4, S, T1)
```

Algorithm 1: `onecycle`: Compute instructions that satisfy a given side-channel constraint $(q2, q3, q4)$.

Data: S_1, S_0, P_0, T_1
Result: T_1

```
addnode(S1, S0, P0, T1)
begin
    if S1 already exists in T1 then
        get the record V1 of node S1 in T1
        add (S0, P0) to V1
    else
        create empty record V1
        add (S0, P0) to V1
        add node (S1, V1) to T1
```

Algorithm 2: `addnode`: add initial state S_0 and a possible instruction P_0 to the search tree of next state T_1.

Data: $q2$, $q3$, $q4$, $S = (W, C, F, D, B)$, T_1
Result: T_1

```
procliteral(q2, q3, q4, S, T1)
begin
    for each literal operation op do
        for each operand opr0 that satisfies HD(opr0, D) = q2 do
            if q3 = HW(opr0) + HW(op) then
                compute resulting S1 of executing op on the working register
                if adjusted q4 = HD(opr0, D1) then
                    addnode(S1, S, op, T1)
    if q3 = 0 and q2 = HW(D) and q4 = HW(W) then                    // NOP
        compute resulting S1 of executing NOP
        addnode(S1, S, NOP, T1)
    if q3 = 1 and q2 = HD(D,file register 0x7f) and q4 = HW(file register 0x7f)
    then                                                             // CLRW
        compute resulting S1 of executing CLRW
        addnode(S1, S, CLRW, T1)
```

Algorithm 3: `procliteral`: find all possible literal instructions satisfying $(q2, q3, q4)$. Some optimization is omitted.

Data: $q2$, $q3$, $q4$, $S = (W, C, F, D, B)$, T_1
Result: T_1

```
procbyte(q2, q3, q4, S, T1)
begin
    for each file-register operation op do
        d ⟵ W
                                        // result stored in the working register
        for each file-register f in G_w of w = q3 − HW(op) do
            if HD(D, f) = q2 then
                compute resulting S1 of executing op on f
                if q4 = HD(f, D1) then
                    addnode(S1, S, op, T1)
        d ⟵ F
                                        // result stored in GPR
        q4' ⟵ adjusted q4 for each file-register f in G_w of w = q3 − 1 − HW(op) do
            if HD(D, f) = q2 then
                compute resulting S1 of executing op on f
                if q4' = HD(f, D1) then
                    addnode(S1, S, op, T1)
```

Algorithm 4: Algorithm 4: `procbyte`: find all possible byte-oriented file register instructions (including conditional branch `DECFSZ` and `INCFSZ`) that satisfy $(q2, q3, q4)$. Some optimization is omitted.

Data: $q2$, $q3$, $q4$, $S = (W, C, F, D, B)$, T_1
Result: T_1

```
procbit(q2, q3, q4, S, T1)
begin
    if q4 = 0 or 1 then
        for each bit operation op do
            d ⟵ W
                                        // result stored in the working register
            for each file-register f do
                if HD(D, f) = q2 and w = q3 − HW(f) − HW(op) ∈ [0, 3] then
                    for each bit b ∈ [0, 7] of weight w do
                        compute resulting S1 of executing op on bit b of f
                        if HD(D1, f) = q4 then
                            addnode(S1, S, op, T1)
    for each BTFSC/BTFSS operation op do
        for each file-register f do
            if HW(f) = q4 and HD(D, f) = q2 and
            w = q3 − HW(f) − HW(op) ∈ [0, 3] then
                for each bit b ∈ [0, 7] of weight w do
                    compute resulting S1 of executing op on bit b of f
                    addnode(S1, S, op, T1)
```

Algorithm 5: `procbit`: find all possible bit-oriented file register instructions (including conditional branch `BTFSC` and `INCFSZ`) that satisfy $(q2, q3, q4)$. Some optimization is omitted.

k instructions. Knowing the original code and initial state, $\{q_i\}$ can be computed for each instruction cycle. If we find that the side-channel measurements are the same with $\{q_i\}$, then we can derive that the final state is one of the state in T_k. Further, if T_k only contains one record, then we can guarantee that the desired state is reached.

For example, when the initial state is

$$S_0 = (16, 000b, \{fs_i | fs_i = \{f_{ij} | f_{ij} = 0\}\}, 32, \text{False})$$

where $000b$ means three binary zeros.

Given side-channel constraint is $\{(1, 8, 1), (7, 10, 6), (1, 7, 4)\}$ for three instruction cycles, then only one final state is reached, namely

$$S_3 = (7, 001b, \{fs_4 = \{8, 0, \dots, 0\}, fs_i | fs_i = \{f_{ij} | f_{ij} = 0\}, i \neq 4\}, 7, \text{False})$$

There are 20 possible three-instruction traces that satisfy the side-channel constraints, e.g., "`BSF 0x47,3; ANDLW 0xE7; DECFSZ 0x47,W`", "`BSF 0x65,3; ANDLW 0xE7; DECFSZ 0x65,W`", and "`BSF 0x78,3; ANDLW 0xE7; DECFSZ 0x78,W`". However all the traces lead to the unique final state S_3.

Side-channel Programming This leads to the idea of "side-channel programming", in which internal state of a program can be verified by simply checking the side-channel emanations. If the entire program can be rewritten in a way that all alternative programs that cause the same side-channel measurements will result in the same final state then the device can be guaranteed to function as desired, in spite of probably different intermediate instructions. This notion of integrity is different from conventional notion of verifying the code. But as mentioned in Chapter 1, verifying the code solely cannot rigorously guarantee the device is not tampered, due to the existence of code-reuse attacks and data-only attacks.

Unfortunately, because the instruction set of the target μC is compact in instruction coding and many instructions having the same HW perform completely different operations, it is not obvious how to side-channel program an arbitrary program, or even whether it is

possible to side-channel program. To answer these questions, we compute the statistics of side-channel emanations versus runtime state and instructions to obtain some hint.

4.2 Heuristics

First, we compute the resulting states given 1,000 random initial states and all the possible qs for one instruction cycle. The average size of the resulting search trees versus q is shown in Figure 4.1. It is interesting to observe that most qs are not valid to produce any final states. For the remaining valid qs, the tree sizes are Gaussian/binomial-like. This can be further illustrated in Figure 4.2, in which the tree sizes versus $q2$ and versus $q3$ are Gaussian/binomial-like, whereas the tree size versus $q4$ is much more uniform. This implies $q4$ does not play a very important role in controlling the number of resulting final states, which is out of expectation. The combination of $q2$ and $q3$ determines, on average, how many different final states are possible given the same initial state and side-channel constraint q.

The statistics on average is only a primitive result for side-channel programming. Only those qs that lead to a unique final state is of value to us. We find that for a given initial state, there are around 7.58% of all possible qs leading to a unique final state for one instruction cycle. If at each instruction cycle we transform runtime states using one of the 7.58% qs, then we can guarantee that for each cycle the device computes the desired value.

However, it may not suffice to use only 7.58% qs to transform an initial state to any desired final state, before the physical program memory runs out. We can assume that some heuristic function can generate a suitable q based on current runtime state, and the heuristic function will guarantee (at high probability) that the desired unique final state is reached within a few steps. For example, the heuristics may allow more than one possible states are reached at intermediate steps while forcing the intermediate states to converge to a unique desired final state. This algorithm is shown in Algorithm 6, where the heuristics $genq(\cdot)$ generates a q for one instruction cycle based on context. The simplest $genq(\cdot)$ is to generate a random q at each step regardless of current context. An optional throttle value is chosen to limit the size of the search tree at each instruction cycle.

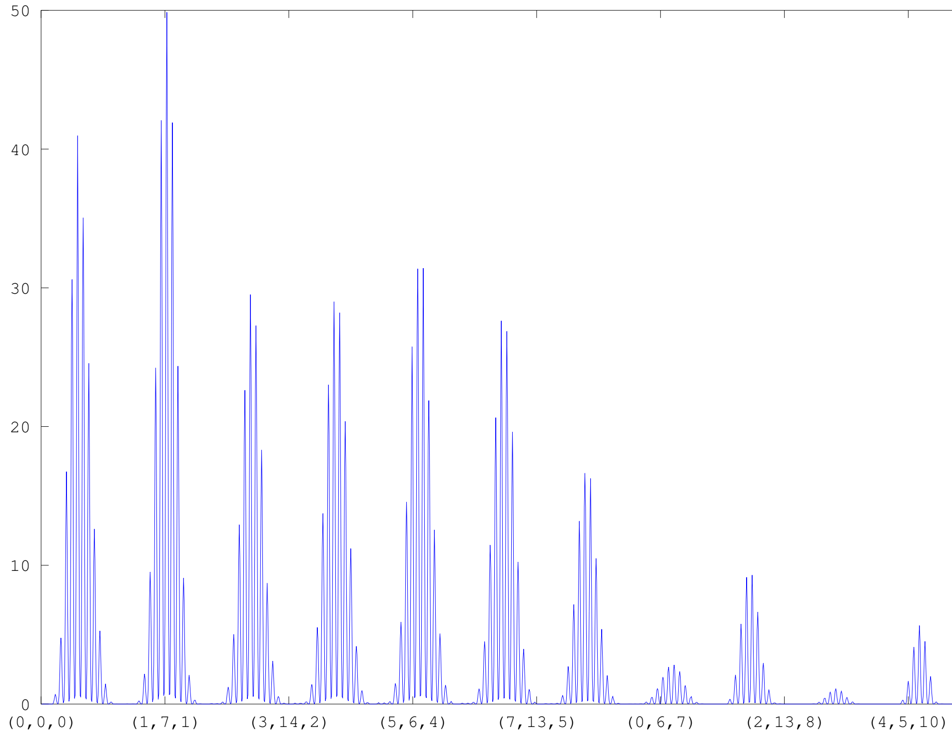


Figure 4.1: The average size of search trees for one instruction cycle: the x-axis is $(q2, q3, q4)$ (ranges $(0,0,0), (1,0,0), \ldots, (8,14,10)$); the y-axis is the average number of nodes of the resulting search trees.

Data: S_0, n
Result: S_1

`uniquetx`(S_0, n)
begin
- $T_i \longleftarrow \emptyset, i = 0, \ldots, n$
- add $(S_0, null)$ to T_0
- **while** $i \leq n$ **do**
  - $q \longleftarrow$`genq`($context$)
  - $T_{i+1} \longleftarrow$ `onecycle`($q2$, $q3$, $q4$, T_i)
  - **if** *size of* T_{i+1} *is one* **then**
    - break
  - **else if** *size of* T_{i+1} < *threshold* **then**
    - $i \longleftarrow i + 1$

Algorithm 6: `uniquetx`: Compute programs that transform a given initial state to a unique final state.

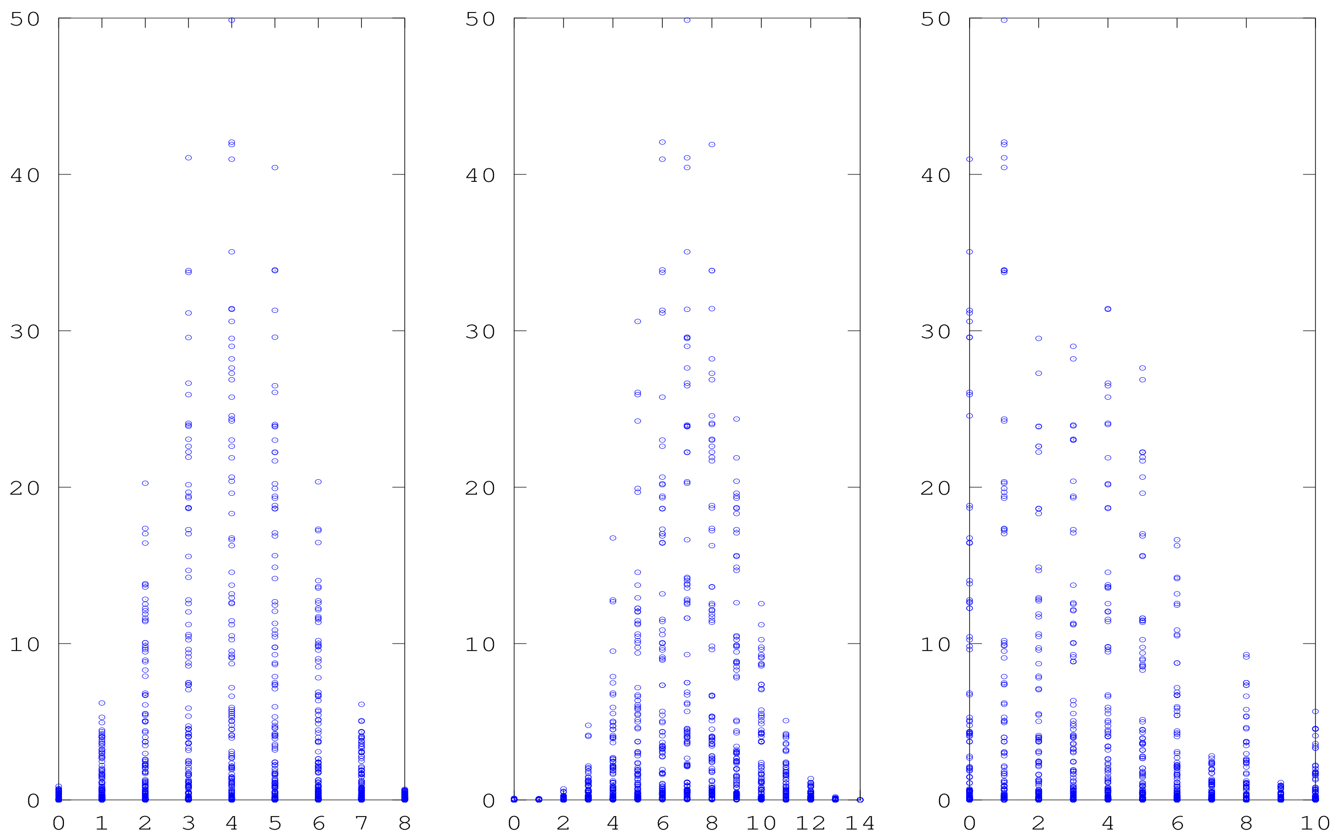


Figure 4.2: The size of search trees after one instruction cycle versus $q2$, $q3$, and $q4$, respectively. The x-axis is $q2$, $q3$, or $q4$; the y-axis is the number of nodes of the resulting search trees.

4.3 Graph of side-channel programs

Since it is not clear how to design the heuristics that can uniquely transform from any initial state to any desired final state, we turn to a reductionist method. First, we design a simple $genq(\cdot)$ that is able to uniquely transform a given initial state to some final state. Second, we try to link each unique transformation into a program that results into the desired final state. To do so, we compute a directed graph of runtime states connected by side-channel programs that guarantee unique transformations of runtime states from verifying side-channel emissions. That is, each node i of the side-channel programming graph is a state S_i, and an edge exists from S_i to S_j only if at least one side-channel program (of any instructions) exists to uniquely transform S_i to S_j. Since computing the entire graph that connects all possible runtime state is impractical (and unnecessary), we only test the effectiveness of this approach by computing the graphs for representative values.

4.3.1 Random initial GPRs

First we consider the initial state to be

$$S_w = (w, 000b, F_0, w, \text{False}) \tag{4.2}$$

where F_0 is set to random value. We compute the graphs of side-channel programs for 20 different F_0s with $w \in [0, 255]$ for each F_0. For each S_w, we execute Algorithm 6 for 200 times, with $genq(\cdot)$ generates random qs and $threshold = 100$. We obtain several interesting results.

First, it is surprisingly easy to obtain side-channel programs. Given the maximum number of instruction cycles n to be 2, 4, 8, and 16, the number of side-channel programs generated for any S_w is on average 70, 95, 120, and 136, respectively, for 200 times of trials. The number of new programs decreases near exponentially with the number of instruction cycles. When n is 2, 4, 8, or 16, the side-channel programs result in 51, 74, 99, and 115 distinct final state values, respectively, for any S_w on average. This shows that the side-channel programs

compute diverse final states for any given initial state.

Among the 51, 74, 99, and 115 distinct final state values S', there are 26, 31, 33, and 33 state values, respectively, that have the same values of FS (i.e., $FS' = F_0$). Others have different FS' that differ from F_0 for one to several GPR values. There are therefore statistically both side-channel programs that uniquely transform any initial state to resulting states having different GPRs, and also side-channel programs that uniquely transform any initial state to resulting states having the same GPRs.

In addition, there are 21, 26, 28, and 28 state values, respectively, that not only have $FS' = F_0$, but also have the resulting data D' equal to the working register W'. There are 10, 12, 12, and 12 state values, respectively, that not only have $FS' = F_0$, $D' = W'$, but also have the `STATUS` register equal to $000b$.

We can therefore compose a graph of side-channel programs with each node as S_w, $w \in [0, 255]$ (note that $w = D$). Given n as 2, 4, 8, 16, there are 228, 236, 238, and 238 out of the total 256 S_w that can traverse the entire graph (i.e., transformed to any possible S_w through side-channel programs), by repeating Algorithm 6, with $genq(\cdot)$ generates random qs. Note that the graph is a lower bound of the actual graph with nodes of the form S_w because it does not include the edges that connect two nodes of the form S_w indirectly through nodes not of the form S_w.

Furthermore, since there are 25, 43, 66, and 82 distinct final state values, respectively, that have different FS', any S_w can transform to a different FS' through side-channel programs, thus connecting different graphs of S_w. Combining the graphs of the same GPRs FS and of different GPRs will produce a side-channel program that statistically traverses any internal states.

4.3.2 Zero initial GPRs

Second, we consider special initial states that have all the GPRs of zero values, since after system reboot, all GPRs are initialized with zero. The initial side-channel distinguishable states are of the form

$$S_w = (w, 000b, \{fs_i | fs_i = \{f_{ij} | f_{ij} = 0\}, i = 1, \ldots, 7\}, w, \text{False})$$

where $w \in [0, 255]$.

For each initial state S_w, we execute Algorithm 6 for 100 times with $n = 8$ and $threshold = 100$. $genq(\cdot)$ generates random qs. Then we connect S_i to a final state S_j only if a side-channel program exists resulted in a unique S_j. Note that again the graph does not contain any nodes not of the form S_w and therefore excludes the edges that connect two nodes of the form S_w indirectly through nodes not of the form S_w.

We find that 242 out of the total 256 nodes in the graph of S_w can traverse the entire graph, showing that the majority of the runtime states of the form S_w can be reached by side-channel programming. Executing 100 more times of Algorithm 6 for each S_w connects 10 more nodes. Executing more times of Algorithm 6 for each S_w is likely to add more edges to the graph and connect the remaining 4 unconnected nodes. Considering nodes of more general forms or increasing n and/or $threshold$ is also likely to connect the remaining nodes.

4.4 Other heuristics

The results in Section 4.3 are obtained with the simplest heuristic function: $genq(\cdot)$ uniformly generates q at each step, without considering context. It is free to choose any other heuristic functions to obtain desired properties. Algorithms 7 and 8 show two other heuristics we have tried for $genq(\cdot)$.

We consider the initial state to be in the form of 4.2, with random initial GPRs. Executing Algorithm 6 by using Algorithm 7 and 8, respectively, we again compute side-channel graphs for 20 different F_0s with $w \in [0, 255]$ for each F_0. For each S_w, we repeat the computation for 200 times, with the maximum number of instruction cycles n set to 8. The numbers of resulting side-channel programs are on average 74 and 180, respectively (i.e., the probability of obtaining a side-channel program of maximally eight instruction cycles is 37% by using Algorithm 7 and is 90% by using Algorithm 8). The numbers of distinct final states (as

Data: C, n
Result: $q2$, $q3$, $q4$

```
genqec(C, n)
begin
    if current step obtained from context C is less than half of maximum number of
    steps n then
        q2 ⟵ rand([3, 5])
        q3 ⟵ rand([5, 9])
    else
        q2 ⟵ rand([0, 2] ∪ [6, 8])
        q3 ⟵ rand([0, 4] ∪ [10, 14])
    q4 ⟵ rand([0, 10])
```

Algorithm 7: `genqec`: Generate qs by considering context C.

Data:
Result: $q2$, $q3$, $q4$

```
genqless()
begin
    if current step obtained from context C is less than half of maximum number of
    steps n then
        q2 ⟵ rand([3, 5])
        q3 ⟵ rand([5, 9])
    else
        q2 ⟵ rand([0, 2] ∪ [6, 8])
        q3 ⟵ rand([0, 4] ∪ [10, 14])
    q4 ⟵ rand([0, 10])
```

Algorithm 8: `genqless`: Generate qs that have fewer possible instructions.

different side-channel programs may result in the same final state) are on average 66 and 95, respectively (i.e., 33% and 47%, respectively).

Among the 66 distinct final states by using Algorithm 7, on average 6 (actually 5.65) have the same values of FS (i.e., $FS' = F_0$). Others have FS' that differ from F_0 for one to several GPR values. 5 out of 6 final states not only have $FS' = F_0$, but also have the resulting data D equal to the working register. 2 out of 5 final states further have the `STATUS` register equal to $000b$. The entire side-channel graph for the same F_0 has on average 25 out of the total 256 states that are connected.

For Algorithm 8, on average 44 out of 95 final states have $FS' = F_0$. 35 out of 44 further have the resulting data D equal to the working register. And 13 out of 35 further have the `STATUS` register equal to $000b$. The entire side-channel graph for the same F_0 has on average 204 out of the total 256 states that are connected. Note that above graphs are obtained by considering only the 200 times of executing Algorithm 6 with nodes of the same form of S_w.

Compared to the results in Section 4.3.1 which uses random qs at each instruction cycle and ignores the context, Algorithm 8 seems to be almost as efficient in finding side-channel programs as random qs, while Algorithm 7 is surprisingly less effective. Algorithm 8 is however more advantageous than Algorithm 7 and random qs in that it takes much less time to compute.

Chapter 5

Grey-box Analysis

[1]

In previous chapters we have shown that it is possible to accurately profile the single-captured side-channel emanations of a microcontroller and to perform side-channel-based integrity checking. One important question is whether or not we can apply the same approach in general. To explore the feasibility of using side-channels for software integrity checking on different devices, we choose FPGAs as the next hardware platform to study. FPGAs provide a uniform platform to implement various SoCs, so we can test whether an approach that works for one SoC still works for another. Three different Systems-on-Chip (SoCs) have been studied in this chapter in order to test the effectiveness, efficiency, and generality of our approach.

5.1 Background

There are several types of FPGAs, among which the most widely used is SRAM (static memory)-based FPGAs, as this thesis is concerned. FPGAs consist of an array of programming logic blocks, which generally include logic elements, memory, and multipliers (See[92] for detailed description). A programmable routing circuit connects the logic blocks to realize certain functionality. Altogether, the array of logic blocks is surrounded by programmable

[1] This chapter is based on work already published[91]

input/output (I/O) blocks. Modern FPGAs also include memory blocks and hardwired components such as hard processor cores, as shown in Figure 5.1.

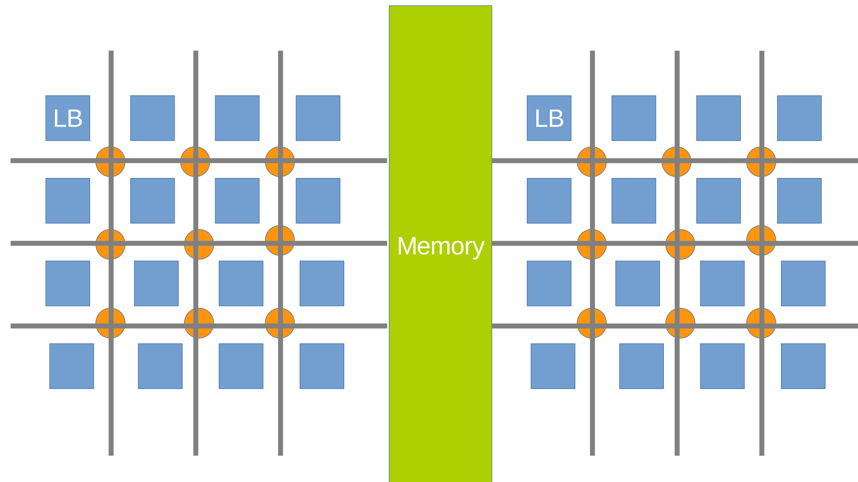


Figure 5.1: FPGA architecture with logic blocks depicted as blue squares; logic blocks are surrounded by routing channels; several logic blocks have a dedicated memory column in-between.

The basic building block of an FPGA device is the logic element, which is essentially composed of a look-up-table (LUT), a flip-flop, and a multiplexer. Several logic elements compose a logic block. Logic blocks are surrounded by routing channels composed of programmable switches and wire segments with different fixed lengths. In SRAM-based FPGAs, SRAM is used to provide configurability by selecting lines to multiplexers and by storing data in LUTs. After powered up, the configuration circuit of an FPGA device will initialize all the SRAM cells with a user provided "configuration logic", which is in general compiled from code written in a high-level "hardware description language" by using a series of CAD tools.

Diverse circuits can be realized by programming the generic LUTs, multiplexers, and interconnects of an FPGA, including an entire system that consists of a sophisticated pro-

cessor, memory, I/O controllers, etc. In that case, the configuration logic will describe both the "hardware" (e.g., the processor circuit) and the "software" which is the content of the memory.

5.2 Related work

To the best of our knowledge, no previous work has been done on side-channel-based software integrity checking for FPGA-implemented SoCs. Note that fundamentally, it is possible to examine every aspect of an integrated circuit, such as state of transistor or SRAM cell, by using micro-probing and microscoping techniques.[93] It is however impractical to use such techniques on a large scale due to the time and expertise that are required, and to the costly equipment that is not available to ordinary companies and university laboratories.

5.2.1 White-box analysis of FPGA side-channels

If the detailed design and manufacturing parameters of an FPGA circuit is known, it is theoretically possible to compute power consumption of an FPGA-implemented circuit. This method has been used primarily for designing power-efficient FPGA architectures and power-aware CAD tools.[48;49;53–55] Dynamic power consumption is in general modeled as the aggregation of power consumed by each node inside an FPGA whose load and parasitic capacitance is charged and discharged at signal transitions, as well as short-circuit power that occurs in CMOS inverters.[48;49;54] For FPGA-implemented SoCs, however, low-level analysis is impractical or inefficient, and relies on the knowledge of detailed design information, which is in general impossible for FPGA developers.

5.2.2 Empirical models of FPGA side-channels

Researchers have in turn tried to profile side-channel characteristics of a target system from empirical measurements of real boards. Senn et al.[58] measured system-level power consumption of the NIOS II soft processor core. Zipf et al.[57] performed a hybrid functional- and

instruction-level power analysis of LEON2, another soft-core processor. However, these estimation models profile the average side-channel emissions of an embedded system rather than trying to infer system state (which program is running and its runtime state) from side-channel measurements, and therefore cannot be applied to the integrity checking problem.

Passive side-channel emissions of FPGA-implemented cryptographic routines have been utilized intensively to extract secret materials such as keys that are embedded in an FPGA circuit.[59–62] FPGA-implemented cryptographic hardware can be either based on a general processor or on a specially designed cryptographic co-processor/peripheral. However, such analysis concentrates on a few leakage points of the keys and cannot present a comprehensive picture of the dynamic side-channel emissions of an entire embedded system (see also Section 2.3.2 and 3.1).

Research on passive side-channel emissions of FPGA devices has also been focused on hardware trojans.[9;10;68] The methods used are either similar to differential power/EM analysis for cryptographic hardware, or are based on accumulative side-channel emanations so that untriggered malicious circuits can be detected. These methods therefore cannot be applied to solve the software integrity checking problem, which requires to detect transient and compact malware that exists for a short time in the memory and does not cause significant difference in long-term side-channel emanations.

5.3 Problem definition

The threat model for FPGA-implemented SoCs is different from that for microcontrollers due to the special properties of FPGA architecture. See[4;5] for a broader discussion of FPGA security problems.

5.3.1 Threat model

Attacker

We assume that the attacker is unable to modify or inject faults into the PCB and the FPGA chip of the target embedded system. This is enforceable in practice using physical tamper-resistance or tamper-proof techniques.[94] Moreover, the attacker is not an insider of the FPGA or IP core manufacturers, and cannot tamper the FPGA IC design, IP cores, or the CAD tools, on which we rely to establish the ground truth.

We further assume that the attacker is unable to modify the FPGA hardware configuration, i.e., cannot reconfigure the hardware of the FPGA. This can be simply achieved by observing the side-channel emissions of the FPGA. During reconfiguration, the waveform of power consumption or EM radiation will change drastically. For example, the main clock of the target SoC will be lost. It is therefore straightforward to check that a device is being reconfigured. Furthermore, this can also reinforced by authenticating the configuration bit-stream[4], or by removing configuration peripherals before deployment.

However, we assume that the attacker *is* able to modify the *application software*, e.g., modifying the RAM of the device through buffer overflows, data-only attacks, etc. We consider two types of attackers: conventional attackers who cannot profile the side-channels of the target device, and attackers who are side-channel-aware and thus actively attempt to evade detection, by crafting the modified software in a fashion that minimizes side-channel deviations from the original one. Nonetheless, the attacker cannot *invasively* profile the side-channels of the target device, which can be again enforced by physical tamper-resistance/proof facilities. Note that fundamentally, if an attacker is able to invasively profile a device, then she can reverse-engineer and modify every aspect of the circuit.[11]

Verifier

We assume the verifier is of very limited capability, so that our approach can be applied to more general scenarios. We assume that the verifier knows the initial configuration of a target device and is able to profile the side-channel characteristics only on a *different* device

of the same model. The verifier can only perform non-invasive measurements on the profiling device, which is important for this method to be applied to deployed devices.

We emphasize that the verifier is completely external to the target device, and *cannot modify the device hardware or software* to change the nature or magnitude of side-channel emissions, so that the verifier has the advantage of invisibility to attackers. This is also to avoid increase EM radiation of the existing device that violates EMC requirements. For example, the verifier may remove the shielding enclosure for measurements, but may not remove the noise decoupling circuits, which may cause errors in device execution. The verifier can only passively measure the target device with equipment that incurs minimal impact on EMC.

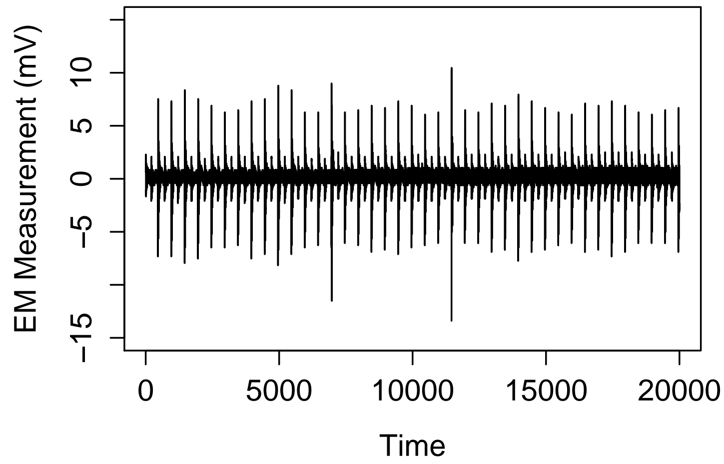
5.4 Experimental setup

For a representative legacy and deployed system, we choose a general-purpose development board for the Altera Cyclone III FPGA EP3C5E144C8 as the target device. The FPGA chip is designed for low-cost, small-scale applications. We choose the EM side-channel for non-invasive measurement, because unlike power analysis, EM measurement can be easily conducted on deployed systems, as no insertion of shunt resistors or similar current measurement components is required. Our experiments measure the inherent EM radiation of the target SoC, and make no effort to increase EM emissions to avoid introducing additional EMC problems. Preliminary test on the chip shows that it is not EM-shielded, which eases our experiment.

We implement one SoC on the FPGA at a time in the way that the only observable I/O is a parallel peripheral I/O (the memory chip on the development board is not used). Two different chips (i.e., boards) of the same model are used, one for profiling and another for testing.

It is interesting for us to find that several positions on the board emit strong EM signals of similar waveform. The conjecture is that the EM measurement is similar to power consumption, since a large dimension probe tends to measure global radiation. We have tried

(a) The SoC and probe



(b) A typical waveform

Figure 5.2: Electromagnetic measurement

both far-field and near-field measurements of EM radiation, and obtained the best result from near-field measurements with a shielded loop probe similar to the EMC probe in[95]. The probe measures the global radiation of the FPGA chip. The resulting setup, with the probe position near one of the power regulators, is shown in Figure 5.2a. Probes of different dimensions, with perimeters varying from 1 cm to 2.5 cm, both shielded and unshielded, provide very similar waveform after denoising, which implies a certain robustness of the acquisition equipment.

The output of the probe is amplified by a 20 dB amplifier with bandwidth from 1 kHz to 1 GHz, and then is sampled by a PicoScope 5244B oscilloscope, which has a 200 MHz bandwidth and a maximally 500 MS/s sample rate for each channel. We use the 20 MHz integrated hardware filter of the oscilloscope to avoid aliasing. The clock frequency of the processor therefore should be set far lower since the EM signals are of much higher frequency than the main clock. We set the clock to 1 MHz, and the EM signals should retain for our 200 MHz oscilloscope. Our results should be repeatable at higher frequencies using more costly oscilloscopes that support higher bandwidths. The position and orientation of the probe is then adjusted to gain signals of the maximal signal-to-noise ratio (SNR). Probe location is re-adjusted for each SoC. The resulting SNR of the EM traces, computed by the ratio of the variance of the signal and noise, is around 15 dB. A typical single-captured waveform is shown in Figure 5.2b.

We have intentionally used low-cost signal acquisition equipment in order to show that verification of low-end and/or legacy systems can be accomplished with only modest resources. It is unrealistic to protect low-end and/or legacy systems using high-end equipment, unlike the case in attacking. The most costly component in our experiment setup is the off-the-shelf USB oscilloscope, at about $2,000; putting the total system cost at under $2,100. Combining all components into a single "software integrity measurement device" and manufacturing at scale is likely to further reduce costs.

5.5 The test code and SoC test targets

We evaluate our approach on three SoCs, implemented in turn on our FPGA test-bed: a NIOS II-based system capable of running an operating system; a simpler NIOS II-based system with a more constrained resource configuration; and an OpenMSP430-based system that is also operating system-capable.

5.5.1 System A: NIOS II-based SoC

Our first experiment is on a NIOS II-based SoC. NIOS II is a general-purpose 32-bit RISC soft processor core from Altera.[96] We choose the NIOS II/e Quartus II 13.1 web edition (the latest edition for the target FPGA). NIOS II/e is designed for simple control logic applications and/or inexpensive systems. It supports over a hundred instruction operations, executed in a variable number of clock cycles, ranging from six to 38. (Our experiments show it is actually seven to 39, contrary to specifications.) The HDL source is not available, and the processor offers only limited configurability. The architecture contains no cache or memory management units and does not perform branch prediction. It does not support different operating modes or memory protection, so modern security mechanisms that rely on processor-enforced separation cannot be used.

The NIOS II-based system is composed of the core, 40 kB M9K RAM, a timer, and a 16-bit parallel I/O connected using the Avalon bus. The system can run the FreeRTOS operating system[97] and several application tasks. The entire FPGA-implemented SoC consists of the NIOS II-based system, a small control unit that supplies clock and reset signals to the core, and a phase-locked loop (PLL). Programs are loaded and executed directly in RAM, forming a complete SoC, i.e., with no bus interfaces outside the chip except the parallel I/O.

In addition, we remove the JTAG interface of the processor, as it is unlikely to be present in a deployed device. The 1 MHz clock is obtained by using a PLL core connected to an external 25 MHz clock source. We do not make any effort to enhance the side-channel emissions when generating the system, so the experiment measures the typical EM radiation of a NIOS II-based system.

5.5.2 System B: Resource-constrained NIOS II-based SoC

The second SoC is also NIOS II-based, but simpler, to represent a "bare-boned" system that does not have enough resources to run an operating system. It has only a 16 kB M9K RAM for program and data memory, and an 8-bit parallel I/O. No timers are present. Otherwise is identical to system A.

5.5.3 System C: OpenMSP430-based SoC

The third system is based on OpenMSP430, a 16-bit open-source MSP430 family-compatible processor.[98] It supports 27 core (many instructions are emulated) instructions and seven addressing modes. Any valid combination of source and destination addressing modes is possible in an instruction, unlike NIOS II, which uses explicit load and store operations. Instructions of OpenMSP430 can be byte or word operations, whereas NIOS II supports only 32-bit word operations for instructions other than load and store. The number of clock cycles required for an instruction is variable (from one to six), depending both on the instruction type and addressing modes.

The SoC consists of the processor, 32 kB M9K RAM for program memory and 4 kB for data memory, a timer, and a 16-bit parallel I/O. Otherwise configuration is the same for all three systems. Programs are compiled using MSP430-GCC, then binaries are converted to an FPGA RAM initialization file, loaded and executed directly in RAM, forming a complete SoC.

5.5.4 Test code

Ideally, we should exercise all the possible internal states of the target SoCs to build side-channel models (i.e., template analysis). However, this is impractical since our preliminary tests show that the EM radiation depends on instruction operations, operands, and the content of the registers, memories, etc. (see Section 5.8). We can only build side-channel models from a very limited number of programs and data configurations, compared with the entire state space of the system. The validity of the resulting model is tested both by

the reasonableness of its form and by the predictive power for side-channel emissions of new programs and data. Our test code is a integration of the FreeRTOS operating system port for each core (except system B) and re-implementation[2] of a part of the CoreMark benchmark suite[99]: integer matrix multiplication to test common operations like array access, loops, and multiplication; floating-point multiplication for the floating point library; greatest common divisor for integer division and recursion; quick-sort of vector data for array access and recursion; list find for pointer operations; list quick-sort for list sort operations; string hash for iterative and pointer operations; a finite state machine for control logic; and random assembly code we generate for each core that avoids memory access and bypasses the native compilers: one composed of logic/arithmetic instructions, and one composed of only five types of logic/arithmetic instructions. The program binaries execute in a similar number of clock cycles. We do not model the EM radiation of I/Os, or code that change system-level behaviors, e.g., the timer intervals.

5.6 Modeling side-channels

The EM radiation of a CMOS circuit theoretically contains information about its internal states[1;100;101], but it is challenging to extract. Computing the EM field of complex circuits is not only impractical, but also relies on knowledge of detailed design parameters.[48;53;102] Both processors have complex architectures, the design details of NIOS II are unknown, and the systems are mapped onto the FPGA base array. The resulting complexity and obscurity pose a great challenge for side-channel profiling.

Our preliminary tests show that EM emissions of different operations largely overlap (c.f. Section 5.8). Profiling EM emissions by using general classifiers as done in[17;69;70;72] is unlikely to succeed. Furthermore, knowing only the instruction operations does not guarantee integrity since an attacker may write malware by varying only the operands and content of registers/memory, while keeping the operations the same.

Chapter 3 has shown that the power consumption of a PIC microcontroller can be accu-

[2] To fit into available memory

rately described as a set of linear functions of a few internal data-dependent activities, and the contribution of different operations is negligible. We repeat the same experiment but using EM measurements with the above probe, as shown in Section 3.4.3. The result shows that, the linear model still holds for EM radiation! The only difference is the values of the regression coefficients and omission of a near-DC component, which is linear to the HW of instructions in the power model. This strongly suggests that we may be able to build similar regression models for the FPGA-implemented SoCs.

We assume that the EM sample Y_t at time t can be again modeled as a function of internal states $\vec{x_t} = (x_{1t}, \ldots, x_{pt})$ at t:

$$Y_t = f_q(\vec{x_t}) + N_t$$

where N_t encloses remaining components in the EM radiation including noise and time-dependent components. If we further assume that N_t is not dependent on time, then we can build the model by regression techniques.

Since both processors execute instructions in a variable number of clock cycles, depending on operands and bus traffic, we build side-channel models for each clock cycle and ignore the stages. The sample rate of the oscilloscope is 500 MS/s, meaning that at least 500 regression models *can* be built. In practice, however, most information is found at clock rising edges. Y_t is therefore the peak amplitude at rising edges of clock t. A sum of 26 points near the peak gives slightly better results than using the single peak value. We denoise the traces for use in regression by averaging over only 100 EM traces. Now the problem is to select representative predictor variables $\vec{x_t}$.

5.6.1 Black-box analysis

Switches of internal signals and voltage differences of neighboring signals are a promising initial choice for $\vec{x_t}$, supported by research on power consumption of FPGAs and general circuits.[48;51;53–55] Because the design details of NIOS II are not available, we initially treat

the system as a black box and attempt to reason about internal activities directly from the instruction set documents, as done in Chapter 3.4. However, the EM samples correlate poorly with predictions. This is not surprising, as the target SoC systems are much more complex than the PIC microcontroller. In particular, the FPGA is unlikely to possess regular long wire segments as the dominant power-consuming memory interface in the PIC chip. To find potential internal signals that may correlate with EM radiation, we turn to the simulation models of the processor cores.

5.6.2 Grey-box analysis

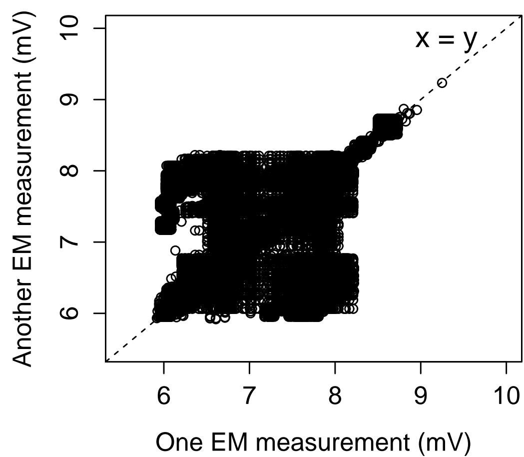
For system A, register-transfer level (RTL) simulation gives runtime values of thousands of signals, such as 35 bus signals. Gate-level post-fit simulation gives runtime values of at least 8259 1-bit internal signals. For system Band system C, there are over 5800 and 12606 1-bit gate-level signals, respectively. The simulation result can be stored in a Value Change Dump File (`.vcd`). EM radiation must be related with these signals in some form. However, directly estimating $f(\cdot)$ does not work due to the sheer number of signals, and also due to the multicollinearity among the signals (many signals are highly correlated with each other, and thus only one signal in a correlated set may be a useful predictor). Some signals are even identical – a simulation artifact. More variables are identical when considering switches of signals (transitions from 0 to 1 or vice versa). However, removing duplicate variables does not eliminate multicollinearity, showing that more complex correlations exist among the signals and signal switches.

As a first step in selecting representative signals for $\vec{x_t}$, we test whether the EM radiation has similar amplitudes when a subset of internal signals stay the same while others vary. If so, we need only to retain the subset of signals for model building. Figure 5.3a shows pairs of EM measurements (peak amplitudes) when bus signals are identical while other signals vary. The x-axis is one EM measurement, and y-axis is another EM measurement. Significant difference in EM measurements can be observed. Figure 5.3b shows pairs of EM measurements when signal transitions (0-1 and 1-0 are regarded as different transitions) of

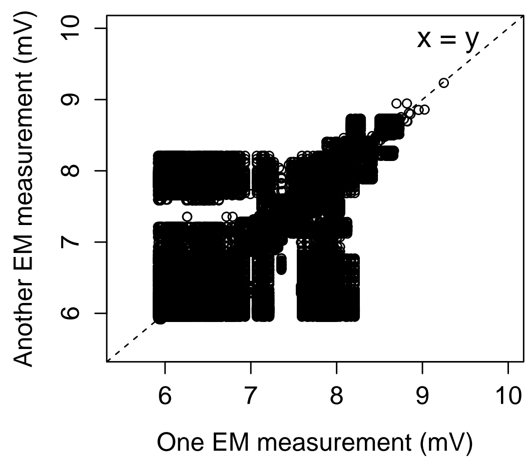
bus signals are identical. These two figures mean that, *no matter what the form of* $f(\cdot)$ *is*, the EM radiation is not determined only by the bus signals. This is in contrast with the PIC chip, whose EM radiation is dominated by the Hamming distance of bus signals. Therefore, we must include additional variables in $\vec{x_t}$. Because it is impractical to try arbitrary subsets of signals, we have to turn to the gate-level signals, as RTL signals are optimized out in final layout of the SoC system. However there are thousands of gate-level signals. To select the most representative ones, we utilize the vendor-provided power estimation tool PowerPlay[103], therefore taking the grey box modeling approach, since PowerPlay encodes manufacturer's knowledge of the FPGA design.

PowerPlay is a tool for developers to estimate power consumption of an FPGA system to allow selection of power supply and heat dissipation scheme. Total thermal power estimates are claimed to have ±20% accuracy to silicon. However, since PowerPlay only reports comparatively rough estimates of accumulative power consumption, it cannot be directly used to solve our problem, which requires side-channel models for at least each instruction. PowerPlay can, however, generate a set of signal names for use in a gate-level simulation (which is in turn used for power estimation). It is reasonable to assume that these signals contribute more significantly to power consumption (thus causing more EM radiation). For system A, 1778 (out of 8259) gate-level signals are selected by PowerPlay, a huge reduction in variables requiring post-processing.

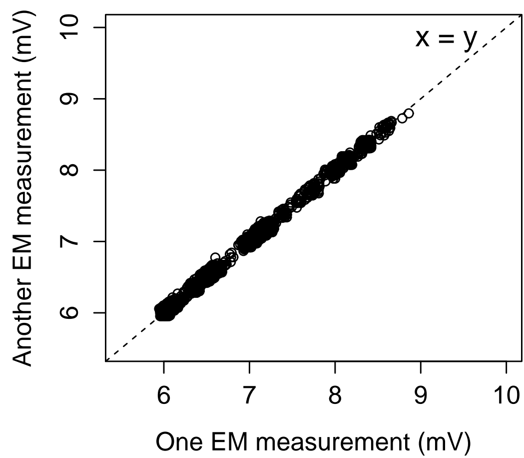
We again test whether EM radiation is similar when the 1778 signals are identical while others vary. Figure 5.3c shows pairs of EM measurements when signal transitions are identical for the 1778 variables, and Figure 5.3d shows pairs of EM measurements when the HDs (which do not distinguish between 0-1 and 1-0) are identical. We do not find enough pairs whose absolute values of the 1778 variables are identical. Nevertheless, Figure 5.3d already illustrates that it is reasonable to select HD of the 1778 variables as $\vec{x_t}$, regardless the real form of $f(\cdot)$ (Note that the set of points in Figure 5.3c is a subset of those in Figure 5.3d). After modeling is finished, we retry using the original 8259 gate-level signals, and find that indeed modeling with the 1778 signals gives the model better predictive power, validating our choice. For system Band system C, the number of selected signals is 1280 and 2715,



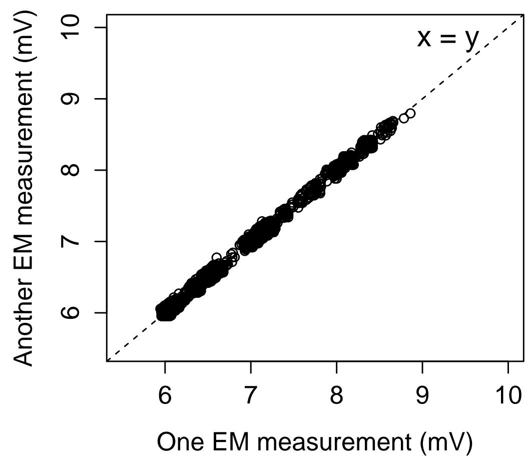
(a) Identical bus signals



(b) Identical *transitions* of bus signals



(c) Identical transitions of selected *gate-level* signals



(d) Identical *Hamming distance* of selected gate-level signals

Figure 5.3: Measurements of system A when a subset of internal signals are identical

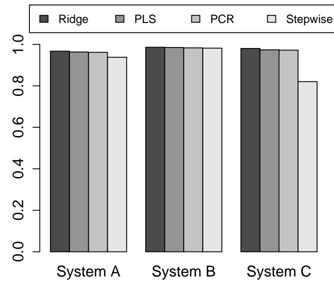
respectively.

However, multicollinearity still exists among the selected variables. Since further dividing the selected variables based on the SoC structure does not lead to improvement and exhaustive search is computationally impractical, we turn to statistical techniques. Several techniques can deal with predictors that have multicollinearity: ridge regression, partial least-square regression (PLS), principal component regression (PCR), and stepwise regression. For the selected variables, it turns out that all regression techniques produce similarly good regression models in terms of the coefficients of determination R^2, MSE, and F-tests in the model building step. To select better models, we perform model validation to measure model reasonableness and predictive power.

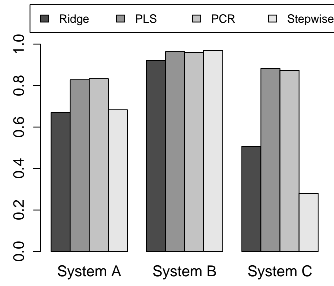
5.7 Validation

We perform five-fold cross-validation to test the ability of the regression model in predicting EM radiation. Among the test programs (Section 5.5.4), half are used for modeling and the other half for testing (recall that they execute in a similar numbers of clock cycles). One FPGA chip is used for building the EM model and a different FPGA chip of the same model is used for testing the model. This stricter five-fold (compared to the common seven-fold or even ten-fold) validation scheme is used because it is impractical to perform exhaustive exploration and associated physical measurements. We are forced to use a limited set of programs for side-channel profiling and derive a model which accurately predicts the experimental results from all other possible programs. The goal is to evaluate the validity of using above variable selection approach and regression techniques for side-channel profiling, rather than to obtain a specific "best" model. Since some of the combinations of modeling/testing code may yield better results, cross-validation eliminates this problem by repeating the modeling and testing procedure using different programs for modeling and testing each time. We exhaustively compute all 252 possible combinations.

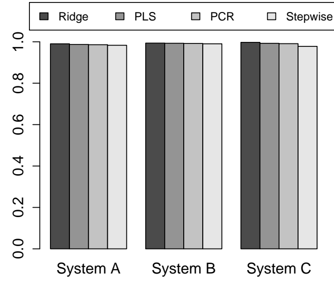
We initially assume $f(\cdot)$ can be approximated as a first-order linear function. Pearson's r and Spearman's ρ are used to evaluate the quality of our models – the larger the correlation,



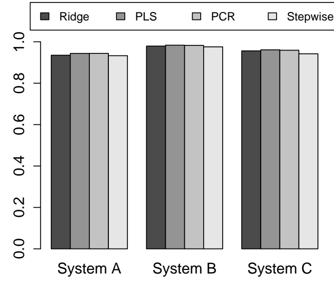
(a) Minimum r of profiling



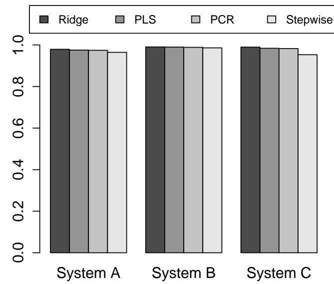
(b) Minimum r of testing



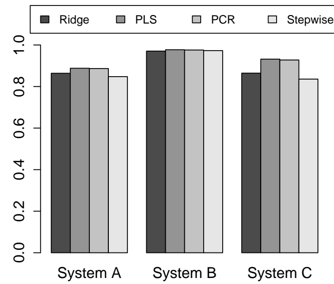
(c) Maximum r of profiling



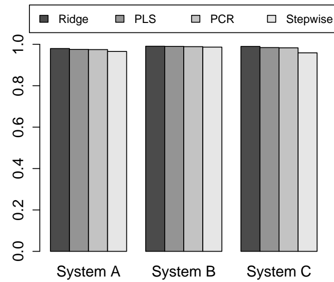
(d) Maximum r of testing



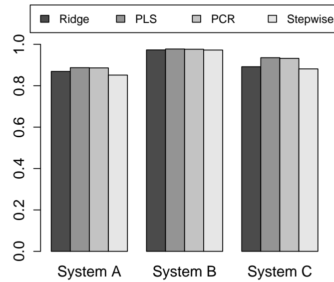
(e) Mean r of profiling



(f) Mean r of testing

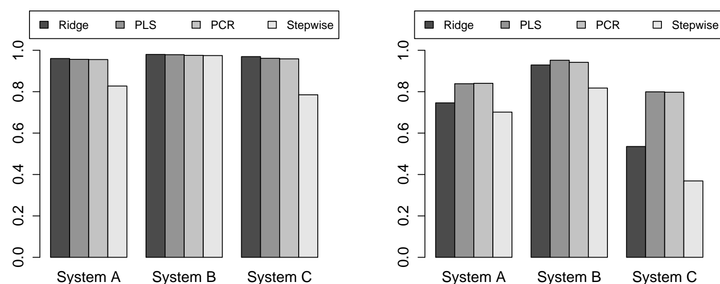


(g) Median r of profiling

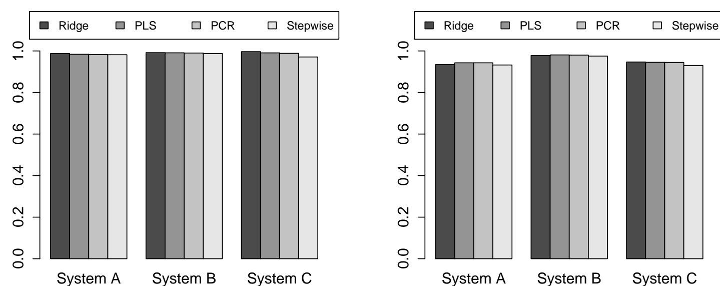


(h) Median r of testing

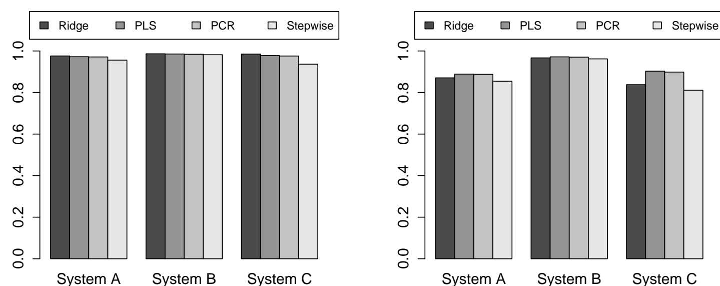
Figure 5.4: Pearson's r for model validation, with the profiling (modeling) results to the left and the testing (on another chip) results to the right; from the darkest bar to the lightest are ridge regression, PLS, PCR, and stepwise regression.



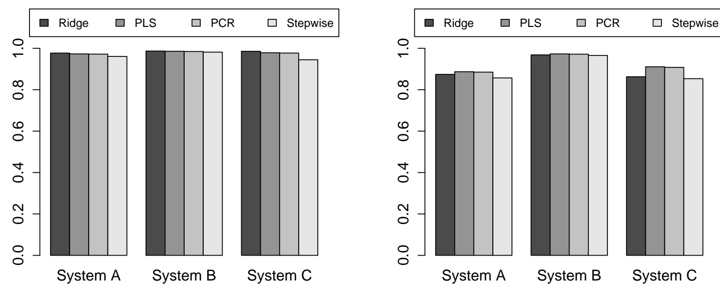
(a) Minimum *rho* of profiling (b) Minimum *rho* of testing



(c) Maximum *rho* of profiling (d) Maximum *rho* of testing



(e) Mean *rho* of profiling (f) Mean *rho* of testing



(g) Median *rho* of profiling (h) Median *rho* of testing

Figure 5.5: Spearman's *rho* for model validation, with the profiling (modeling) results to the left and the testing (on another chip) results to the right; from the darkest bar to the lightest are ridge regression, PLS, PCR, and stepwise regression.

the greater the predictive power. Pearson's r is effective because we observe that slightly moving the probe will only cause the amplitude of EM radiation to change linearly. We still compute Spearman's ρ, which can reveal nonlinear relationships between measurements and models (r and ρ are equivalent when relationships are linear). Pearson's r and Spearman's ρ are shown in Figure 5.4 and 5.5, respectively. r and ρ are almost identical, validating our choice. When profiling and testing using the same FPGA chip, all the metrics, such as rs and ρs, fps and fns (c.f. Section 5.8), are slightly better, as expected. In addition, regressing $\vec{x_t}$ always has the highest correlation coefficients when $\vec{x_t}$ and Y_t are precisely aligned in time, as shown in Figure 5.6. A sharp peak occurs when the model prediction and real measurement have no time offset, showing the soundness and validity of the parameter selection. The parameters of each regression technique are selected to achieve best results for a few pre-selected random modeling/testing combinations and then fixed for all the others: $k = 0.5$ for ridge regression in all cases; number of components is 20 to 23 for PLS; cumulative is 0.99 for PCR; stepwise regression computes the p value of F-test to test the model with or without a term, p_{add} is 0.05 and p_{remove} is 0.1, and the algorithm terminates when no add/remove actions can improve the model to a statistically significant extent. Note that although for a particular combination the best parameter varies, it does not change our conclusions.

The results show that (with few exceptions) linear regression models can predict EM radiation of new programs with satisfactory accuracy, especially for system B. Adding the absolute values of the signals (i.e., Hamming weights) to $\vec{x_t}$ does not improve model performance. PLS and PCR outperform other techniques and are stable in all cases (with no unacceptably low-performing outliers $r < 0.80$). PLS and PCR have been used in various domains such as chemistry and biology, where, similar to our situation, one observation is associated with many variables.[90;104] PLS and PCR have nearly identical performance, which is interesting since unlike PLS, PCR does not take the response Y_t into account when selecting the predictors. The procedure of PCR is deterministic during model building and is not affected by parameter selection. Principal component analysis (PCA) has been used in side-channel analysis as a preprocessing step of pattern matching or classification to reduce dimension and to denoise the sampled traces in *time*.[69;85] We instead use PCA to eliminate

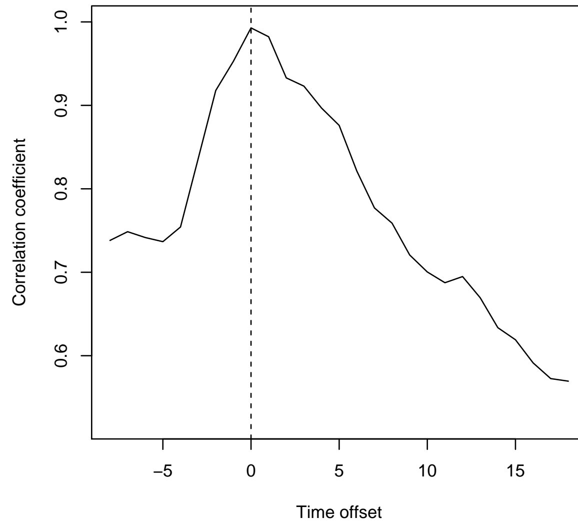
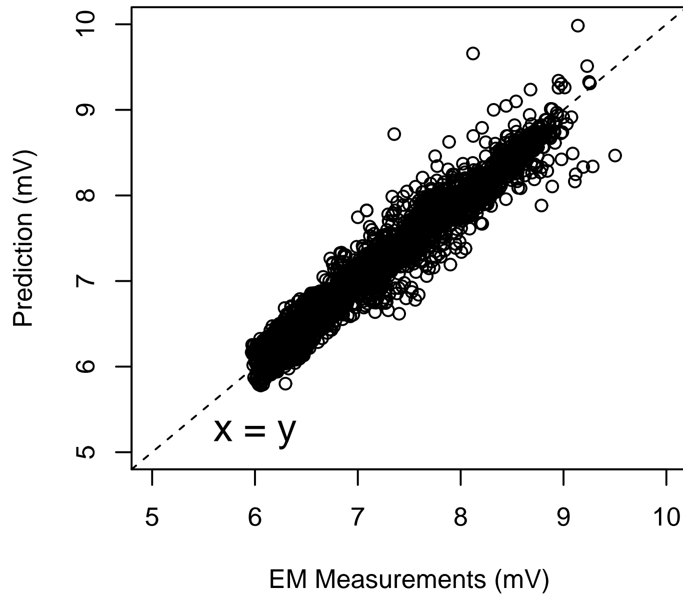


Figure 5.6: Correlation of model prediction and measurement with sliding time window during profiling; the x-axis is the time offset, and the y-axis is Pearson's r computed from the actual measurement and the model prediction which has an offset in time.

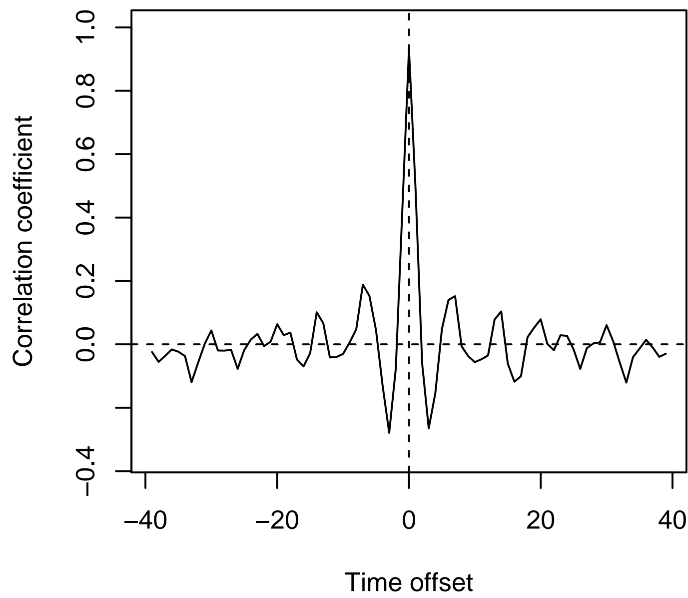
multicollinearity for regression. Note that there is no noise in the predictors $\vec{x_t}$, which are not random variables.

The effectiveness of the regression models is further shown in Figure 5.7. The x-axis of Figure 5.7a is the actual measurement for a testing program, and the y-axis is the model prediction (of the best PLS model of system A). Although some outliers exist when the amplitude of EM measurements is higher, most measurements and predictions fall along the line of $x = y$. Figure 5.7b shows the Pearson's r between the actual measurement of a testing program and the model prediction which has an offset in time from the actual measurement. The x-axis of Figure 5.7b is the time offset, and the y-axis is r. A sharp peak occurs when the model prediction and real measurement have no time offset, showing the soundness and validity of the model.

Second, we examine the resulting models for the reasonableness of their coefficients. The coefficients of the best model in each case are shown in Figure 5.8 through 5.10. It is interesting to observe that PLS, PCR, and stepwise regression result in similar regression coefficients for each system, although the three techniques perform different procedures in



(a) Model prediction (y-axis) versus measurements (x-axis)



(b) Correlation of model prediction and measurement with sliding time window

Figure 5.7: Model prediction and measurements for the best PLS model of system A

regression. Note that the modeling/testing combinations to obtain the best models are different for different techniques. There are both positive and negative coefficients, which we believe is reasonable since unlike the case in power consumption, currents may generate magnetic fields that cancel each other. Several selected signals are clock signals that switch at each clock cycle. These signals do not provide information on internal states, and should only contribute to the constant in the model. We observe that only ridge regression assigns non-zero coefficients to these signals. Indeed, ridge regression performs worst in every case.

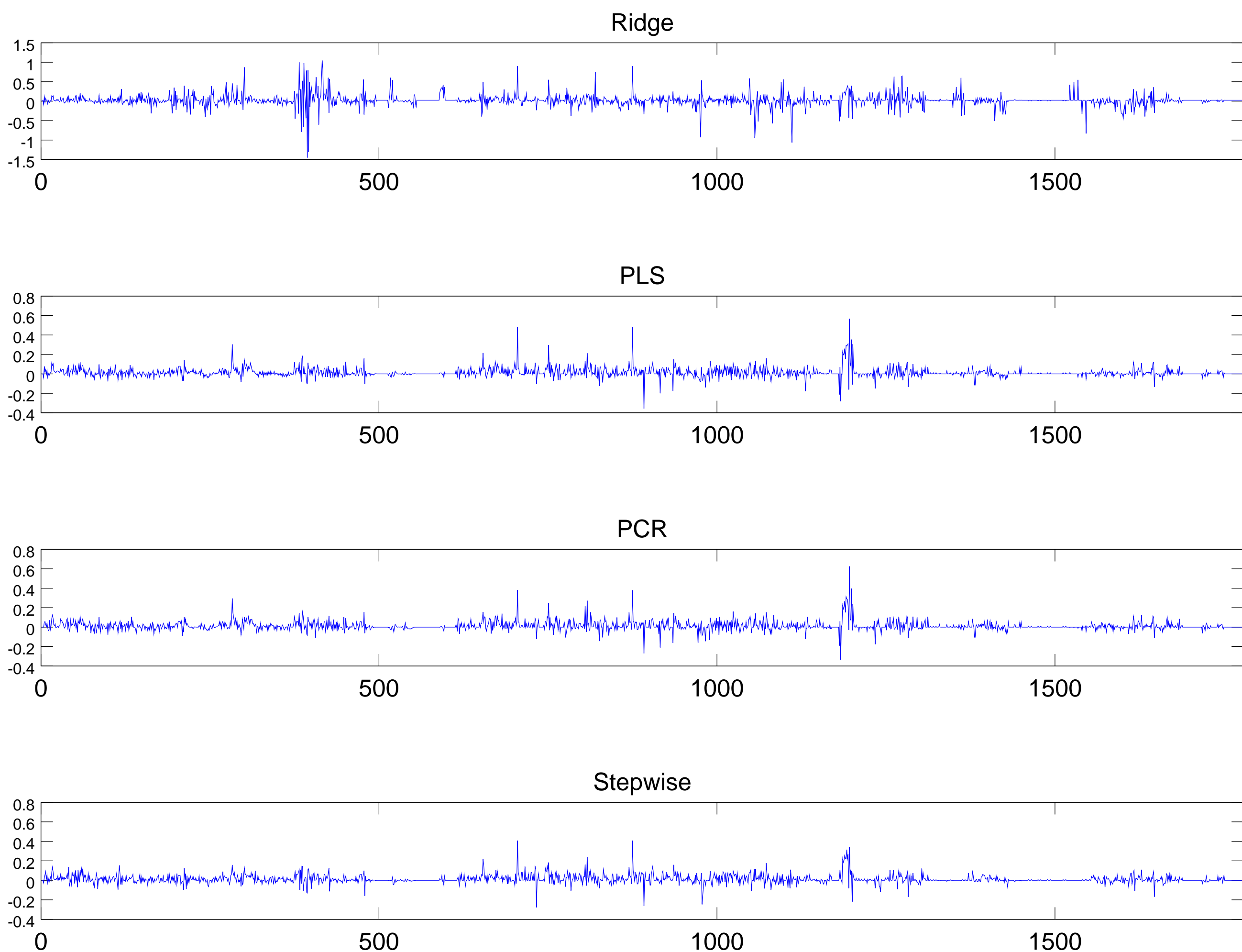


Figure 5.8: The coefficients of the best models for system A, excluding the constant; from the top to bottom are coefficients of ridge, PLS, PCR, and stepwise regression, respectively.

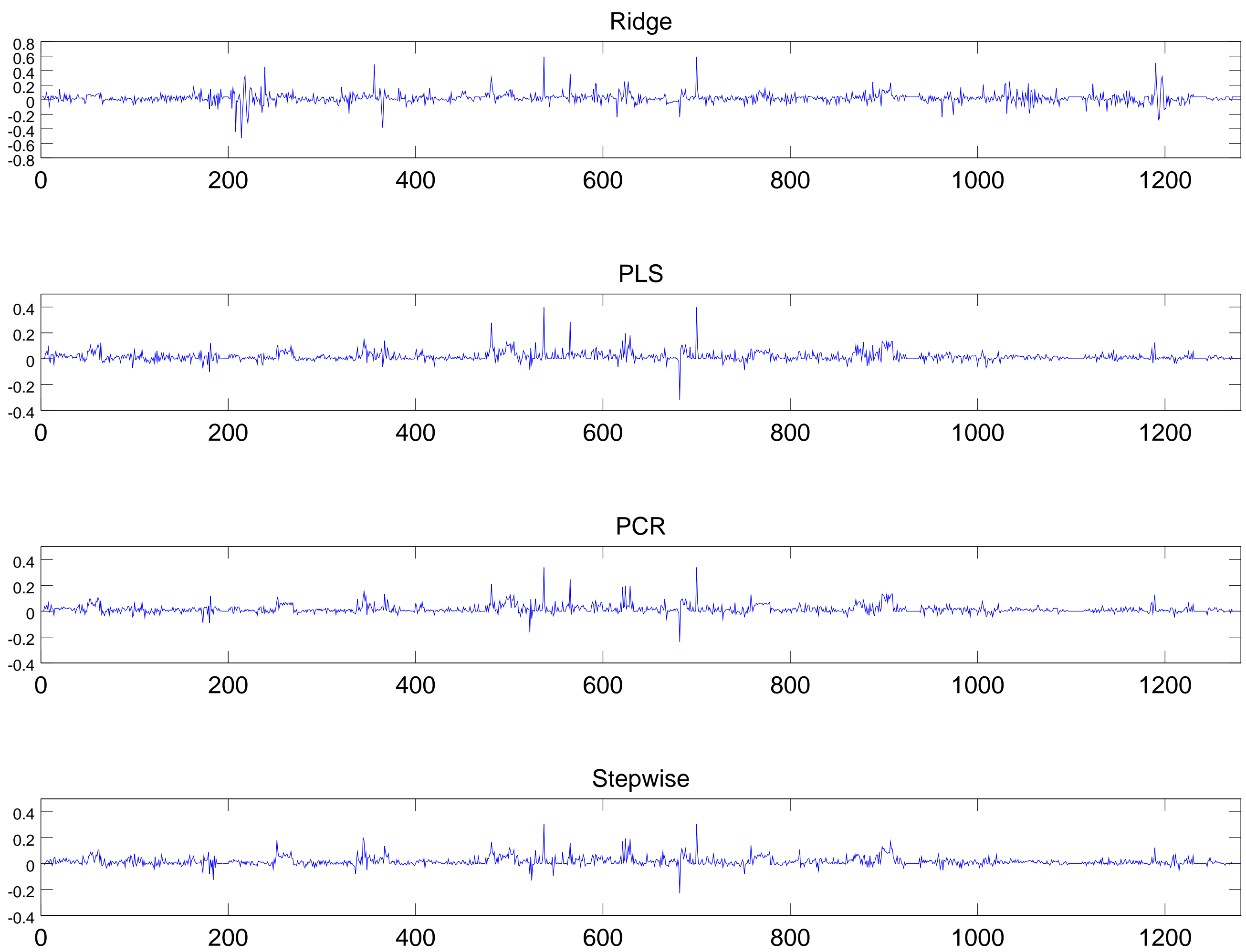


Figure 5.9: The coefficients of the best models for system B, excluding the constant.

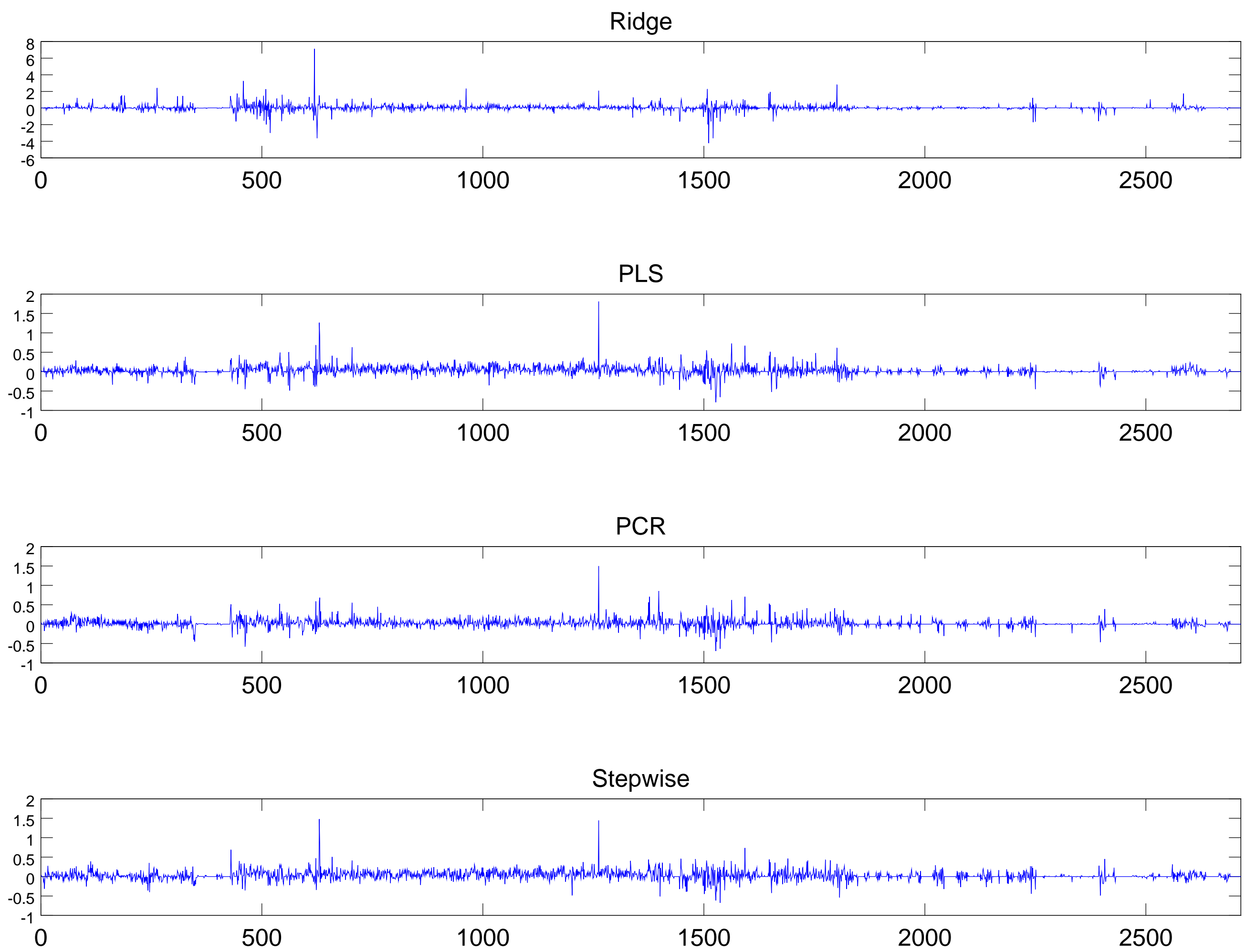


Figure 5.10: The coefficients of the best models for system C, excluding the constant.

PowerPlay reports that the M9K component consumes the majority (∼60%) of the core dynamic power, but only a portion of M9K signals have larger positive regression coefficients in our resulting models. We attempted to regress separately with the M9K signals including memory and registers banks, as well as other signals reported in PowerPlay as consuming more power, yet the resulting models are *n*ot better than original models, especially in cross-validation. The `.vcd` files altered to observe power estimation for individual signals do not seem to work with PowerPlay, producing unreasonable estimates that contradict our observations (such as single signals that have excessive power consumption).

5.8 Applying results to software integrity checking

To enforce code integrity, we must guarantee that tampering with the internal states of the system can be detected from side-channel measurements. Because the EM measurement of the FPGA-based SoCs is a continuous-like variable, we cannot use the same side-channel programming techniques with the PIC microcontroller. Instead, given the regression model, we predict EM radiation of new programs by using the gate-level simulation. We then determine whether tampering with the original code will be reflected in the emission or not by some distance measure. Integrity checking is a hypothetical test of whether a given measurement is from tampered code/data (undesired state) or not. The performance of this integrity mechanism is quantified by (1) the false positive rate (when a normal system state is flagged as tampering), and (2) the false negative rate (when tampering is performed, but is not flagged).

There are typically two use scenarios for software integrity checking. One is to constantly observe the system behavior, here to measure the EM radiation, so that tampering can be detected as soon as tampered code or data start to affect system execution. The drawback is high overhead as the acquisition equipment must be dedicated to the target device. The other less costly scenario is to measure the EM radiation from time to time, at arbitrary intervals, and to determine whether the device is running desirable code – a previous malware execution may have changed internal states of the system. To answer whether we can utilize

our EM acquisition equipment and models for protecting our target system in these scenarios, we compute the statistics of EM radiation from real measurements, the regression models and gate-level simulation.

We first consider a conventional attacker, who is unable to analyze the side-channel information of the target system or unaware of the potential existence of side-channel-based integrity checking mechanisms.

Table 5.1: False positive rates (%) for a(n) (aligned) single-captured EM trace

Threshold	0.90			0.85		
Number of Cycles	7	14	21	7	14	21
System A	14.2	12.9	9.68	8.73	5.92	3.31
System B	13.1	10.3	8.57	8.52	6.14	3.08
System C	16.7	14.7	13.1	9.14	5.53	3.80

Table 5.2: False negative rates (%) for an aligned single-captured EM trace

Threshold	0.90			0.85		
Number of Cycles	7	14	21	7	14	21
System A	20.6	4.65	1.75	30.5	9.83	4.01
System B	5.74	0.99	0.59	10.12	1.98	1.01
System C	0.81	0.18	0.13	1.54	0.22	0.14

Table 5.3: False negative rates (%) for an arbitrary single-captured EM trace

Threshold	0.90			0.85		
Number of Cycles	7	14	21	7	14	21
System A	3.41	0.70	0.24	5.65	1.51	0.56
System B	1.43	0.16	0.09	2.93	0.38	0.16
System C	0.67	0.11	0.10	1.33	0.14	0.10

Recall that the SNR of our experiment is around 15 dB. Taking both environmental noise and inaccuracy in model prediction (regression residual) into account, we obtain from the best PLS models and EM measurements that for system A, 85.8% of the Pearson's r between seven-cycle **single-captured** traces (on the testing chip) with the model prediction (from

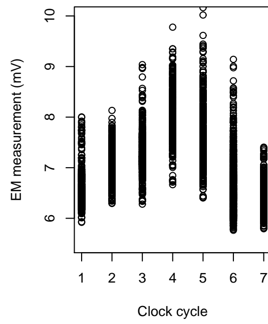
the profiling chip) is greater than 0.90. Note that within seven cycles at most one instruction can be executed. We can design the integrity checking mechanism by fixing the threshold to 0.90, and then compute the false positive rates of any 7-/14-/21-cycle trace for each system from real measurements. Table 5.1 lists the false positive rates when the threshold is fixed to 0.90 and 0.85, respectively. Seven to 21 cycles are chosen because the *actual* number of clock cycles per instruction for NIOS II is from seven to 39. While the actual number of clock cycles per instruction for OpenMSP430 ranges from one to six, we use the same intervals for comparison purpose.

The false negative rate is computed from the percentage of 7-/14-/21-cycle EM traces of different code and/or data on the testing chip, but having r greater than the threshold with the model prediction of the original code with desired data. Table 5.2 lists the false negative rates when the EM traces are aligned with starts of execution, applicable to the case in which tampered code/data executes in the same number of clock cycles with the original code. Table 5.3 lists the false negative rates for arbitrary EM traces that are aligned or misaligned with the original trace.

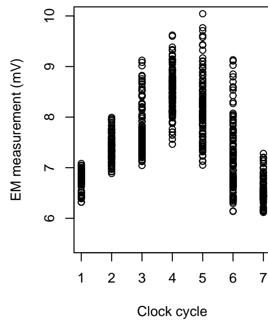
The results show that the probability of random malware evading the integrity check is very low. Even compact malware (of very few instructions) can be detected reliably. Both false positive and false negative rates decrease rapidly as the number of clock cycles increases. When the number of cycles is fixed, there is a tradeoff between false positives and false negatives: a lower threshold will reduce false positives at a cost of higher false negatives. Note that the threshold and number of cycles can be computed to achieve desirable false positive and false negative rates. Overall, the side-channel-based integrity check is effective for a conventional attacker.

Next, we consider a side-channel-aware attacker who actively tries to compute alternative code that has near indistinguishable EM radiation from the original code. To do so, the attacker needs to know the reverse mapping from the EM radiation to runtime states including instructions and data. The effectiveness of our integrity checking scheme relies on the hardness for an attacker to obtain such mapping.

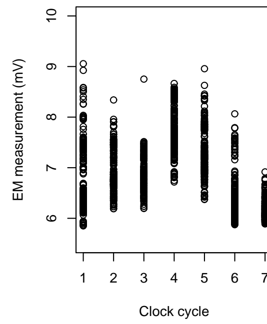
We first analyze the side-channel emissions of instructions classified by operations (e.g.,



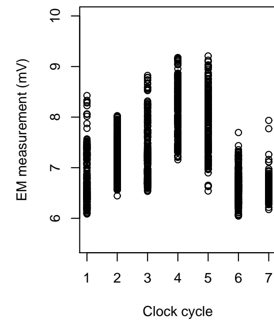
(a) ADD



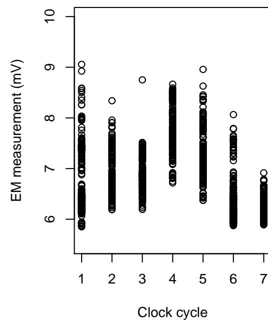
(b) AND



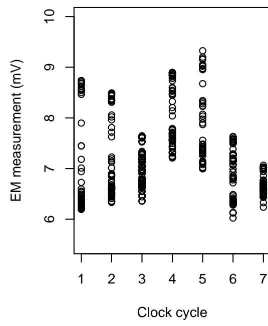
(c) MOV



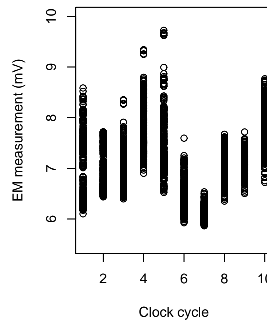
(d) BEQ



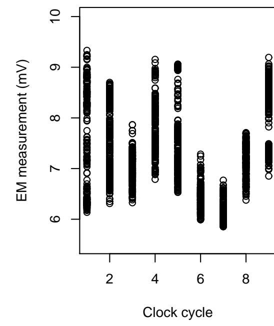
(e) CMPGT



(f) CALL



(g) LDW



(h) STW

Figure 5.11: EM measurements of instructions grouped by operations for system A; x-axis is the number of clock cycles; y-axis is the EM measurement.

`add`, `call`), as done in previous research.[17;69;70;72;105] This is to test whether EM radiation is linked with operations as many researchers have assumed. We find that significant variation exists among instructions of the same operations, and EM measurements of different operations are not discriminatory. Figure 5.11 shows the EM measurements grouped by operations. Figure 5.12a shows an example in which even when executing the same instruction (`add r3,r16,r16`), the EM radiation varies significantly. On the other hand, EM measurements of executing different instructions may have nearly the same value. Figure 5.12b shows an example in which the EM trace of one execution of `add r3,r16,r16` has nearly the same value with those of five different instructions of different operations, obtained by exhaustive search.

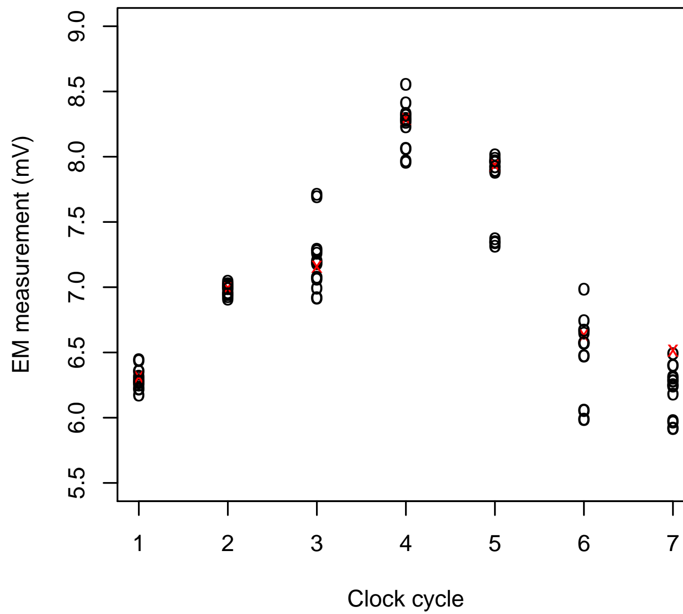
This phenomenon can be quantified by class (i.e., operation) separability by using within-class and between-class scatter matrices (c.f. Section 2.4.1). We compute the statistics of 53 common operations. The resulting J is 1.23, very close to one, which means that EM radiation, when grouped by operations, is not well clustered and is nearly indistinguishable from each other.

For system C, the number of clock cycles per instruction varies from one to six, and depends on the addressing modes of the source and destination operands. Table 5.4 shows class separability for different clock cycles. The resulting J is still very small, meaning that EM radiation of different operations cannot be reliably distinguished from each other.

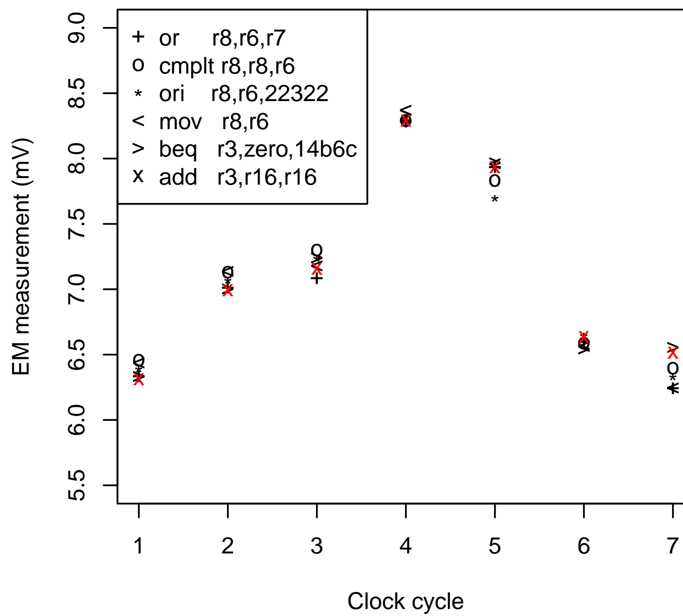
Table 5.4: Class separability J for system C

one cycle	3.81	four cycles	1.71
two cycles	16.63	five cycles	1.54
three cycles	7.51	six cycles	1.37

As shown in Sections 5.6.1 and 5.6.2, the EM model is a function of thousands of selected gate-level signal switches. The operations, bus signals, and M9K signals, which can be easily deduced from code, only contribute to a small portion of EM variance. Even if exactly the same instruction is executed, different runtime state of other signals will cause significantly different EM radiation. The attacker has to rewrite malware based on the many



(a) 16 executions of `add r3,r16,r16`



(b) One execution of `add r3,r16,r16` and five different instructions

Figure 5.12: EM measurements of `add r3,r16,r16` for system A; red cross indicates the same instance of executing `add r3,r16,r16`.

thousands gate-level signals which cannot be manipulated arbitrarily, but rather through the programming model of the processor. Furthermore, the gate-level signals are synthesized and optimized results of the processor core, whose relationship with the assembly code is unknown. Without knowing the design of the processor, as in the case of NIOS II, or without the ability to deal with processor complexity, the attacker will have to exhaustively search for alternative malware code that has similar EM radiation. In addition, each combination of operation and operands will result in a different internal state at each clock cycle. As the length of EM measurements increases linearly, the complexity of searching increases exponentially, effectively making the attack impractical.

Detailed information on experiment setup, data, two ports of test code, and results are available at[106].

Chapter 6

Discussion and Future Work

We have quantified the effectiveness and generality of using low-cost acquisition equipment to verify runtime states of different hardware platforms, including one microcontroller-based SoC and three FPGA-based SoCs, against both conventional and side-channel-aware attackers. Profiling and testing of FPGA-based SoCs use different chips (boards) of the same model.

We show that by using our variable selection procedure and regression techniques, it is possible to model clock-cycle-accurate single-captured side-channel emanations of complex systems, with black-box or grey-box access to the system design. Linear regression has also been used to break cryptographic hardware from side-channel leakage.[82;89;107;108] Side-channel profiling for integrity checking is however very different from that for breaking cryptographic hardware. For cryptographic hardware, what are chiefly concerned are usually special time points when side-channels leak key material information during multiple executions of the same cryptographic routine. It therefore does not give the whole picture of the side-channel emissions of a device. In contrast, for integrity checking to detect one-time execution of malware, we must extract as much information as possible for each cycle in a single measurement, including but not confined to, instruction operations, operands, and register and memory content.

Our integrity checking methods are external. The verifier does not exist within the device

under test, and therefore is invisible to attackers who penetrate the device. The measurement by the verifier is passive and non-invasive, making it a particularly attractive solution for already-deployed systems.

We have shown that for a PIC microcontroller, it is possible to perform "side-channel programming", in which developers utilize side-channel characteristics of the target device to rewrite code in a way that the device is guaranteed to reach from a given initial state to a unique final state when the side-channel constraints are satisfied. Algorithms on how to generate such programs and side-channel constraints are given.

For the FPGA-based SoCs, we have also shown the effectiveness of using passive EM radiation to verify the internal states of SoCs of completely different logic and software configurations. This strongly implies that our approach may be applied to diverse systems of different hardware platforms. Indeed, side-channels have higher potentials than expected to be used for constructive purposes.

Reproducing the work. Some suggestions can be given to interested researchers who would like to reproduce the work on the same devices that we have used and also on different devices. While power consumption can be reliably measured by using the standard shunt resistor approach, there is some uncertainty in the EM measurement since we have used a hand-made loop probe instead of a commercial probe with known electric parameters such as bandwidth and gain. Our intention of making the probe ourselves is to adjust the probe dimension in the preliminary study (and to reduce cost). More rigorous treatment of the design of the sniffer probe for side-channel analysis can be found in[13]. To simply repeat the experiment, one only needs to follow the design of the EMC probe in[95]. The orientation and position of the probe should be adjusted to obtain the maximal SNR. As mentioned in Section 5.4, we have found several positions on the board where the captured EM signals are of similar waveform.

There are also concerns about the environment noise which may reduce the effectiveness of the EM-based integrity checking. Our EM experiment has been conducted in a normal laboratory environment with several working computers nearby, and the device is not EM

shielded. We have not used any anechoic materials. Even the JTAG connector is not unplugged during measurement. This is to show that our approach works in an ordinary noisy environment. The resulting SNR is around 15 dB, as mentioned in Section 5.4. It appears that near-field magnetic measurement is not very sensitive to noise.

Inherent side-channel leakage. Our experiments have measured the inherent side-channel emanations of devices, without any effort to increase the side-channel leakage. This is because we aim to make no modifications on existing hardware, so that our approach is a pure external verifier and can be used for legacy and deployed systems. On the other hand, one may try to increase side-channel leakage so that it may be easier for integrity checking. While it is possible to cause intentional leakage by for example adding an antenna, it requires hardware modification and will inevitably cause interference with the original device. More generally, increasing side-channel leakage conflicts with EMI/EMC requirements.

Future work. Since our side-channel models are based on internal signals that are computed either from the semantic models or the simulation models, it can be inferred that directly applying the approach for integrity checking requires the system to be deterministic. For example, no context switching should happen when measuring a target program. To what extent our approach can be applied for non-deterministic systems is left for future work. In addition, we also plan to try more advanced measurement equipment that may extract side-channel emanations of only a portion of the system and examine how this will help in software integrity checking.

Bibliography

[1] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 1st edition, 2010.

[2] PIC16F631/677/685/687/689/690 data sheet. Microchip Technology Inc., 2008, http://ww1.microchip.com/downloads/en/DeviceDoc/41262E.pdf.

[3] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32. stuxnet dossier version 1.4, 2011.

[4] Saar Drimer. Volatile FPGA design security – a survey. http://www.cl.cam.ac.uk/~sd410/papers/fpga_security.pdf.

[5] Saar Drimer. *Security for volatile FPGAs*. PhD thesis, University of Cambridge, 2009.

[6] Yier Jin and Yiorgos Makris. Hardware trojan detection using path delay fingerprint. In *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust*, HOST, 2008. ISBN 978-1-4244-2401-6. doi: 10.1109/HST.2008.4559049. URL http://dx.doi.org/10.1109/HST.2008.4559049.

[7] Dakshi Agrawal, Selcuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using IC fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE S&P, 2007.

[8] Peilin Song, Franco Stellari, Dirk Pfeiffer, Jim Culp, Al Weger, Alyssa Bonnoit, Bob Wisnieff, and Marc Taubenblatt. MARVEL: Malicious alteration recognition and verification by emission of light. In *IEEE International Symposium on Hardware-Oriented Security and Trust*, HOST, 2011. doi: 10.1109/HST.2011.5955007.

[9] Sergei Skorobogatov and Christopher Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*, CHES, 2012. ISBN 978-3-642-33026-1. doi: 10.1007/978-3-642-33027-8_2. URL http://dx.doi.org/10.1007/978-3-642-33027-8_2.

[10] Oliver Soll, Thomas Korak, Michael Muehlberghuber, and Michael Hutter. EM-based detection of hardware trojans on FPGAs. In *IEEE International Symposium on Hardware-Oriented Security and Trust*, HOST, May 2014. doi: 10.1109/HST.2014.6855574.

[11] Sergei P. Skorobogatov. Semi-invasive attacks – a new approach to hardware security analysis. Technical reports No. 630, the University of Cambridge.

[12] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, volume 5747 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009.

[13] Elke De Mulder. *Electromagnetic Techniques and Probes for Side-Channel Analysis on Cryptographic Devices*. PhD thesis, Katholieke Universiteit Leuven, 2010.

[14] Takeshi Sugawara, Daisuke Suzuki, Minoru Saeki, Mitsuru Shiozaki, and Takeshi Fujino. On measurable side-channel leaks inside ASIC design primitives. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, volume 8086 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[15] Naofumi Homma, Yu ichi Hayashi, Noriyuki Miura, Daisuke Fujimoto, Daichi Tanaka, Makoto Nagata, and Takafumi Aoki. Em attack is non-invasive? design methodology and validity verification of em attack sensor. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, volume 8731 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014.

[16] Franck Courbon, Philippe Loubet-Moundi, Jacques J.A. Fournier, and Assia Tria. Increasing the efficiency of laser fault injections using fast gate level reverse engineering. In *Hardware-Oriented Security and Trust, 2014. HOST '14. IEEE International Workshop on*, HOST, 2014.

[17] Daehyun Strobel, David Oswald, Bastian Richter, Falk Schellenberg, and Christof Paar. Microcontrollers as (in)security devices for pervasive computing applications. *Proceedings of the IEEE*, 102(8), 2014.

[18] Daehyun Strobel, Florian Bache, David Oswald, Falk Schellenberg, and Christof Paar. Scandalee: A side-channel-based disassembler using local electromagnetic emanations. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE, 2015.

[19] Dakshi Agrawal, Selc uk Baktr, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using ic fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE S&P '07, 2007.

[20] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE S&P, 2010.

[21] Zongwei Zhou, J. Newsome V.D. Gligor, and J.M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE S&P, 2012.

[22] Roberto Guanciale Mads Dam and Hamed Nemati. Machine code verification of a tiny ARM hypervisor. In *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED '13, 2013.

[23] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple ARM-based

separation kernel. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, 2013.

[24] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *Proceedings of the USENIX Security Symposium*, USENIX Security, 2012.

[25] grsecurity. http://grsecurity.net/.

[26] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE S&P, 2014.

[27] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the USENIX Security Symposium*, USENIX Security, 2014.

[28] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque control-flow integrity. In *ISOC Network and Distributed System Security Symposium*, NDSS, 2015.

[29] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In *Proceedings of the International Conference on Recent Advances in Intrusion Detection*, RAID, 2011.

[30] Ang Cui, Jatin Kataria, and Salvatore J. Stofo. From prey to hunter: Transforming legacy embedded devices into exploitation sensor grids. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, 2011.

[31] Francisco Rodríguez, Serrano, and JuanJosé. Control flow error checking with ISIS. In *Embedded Software and Systems*, volume 3820 of *LNCS*. 2005.

[32] Frank Stajano and Ross Anderson. The grenade timer: Fortifying the watchdog timer

against malicious mobile code. In *Proceedings of International Workshop on Mobile Multimedia Communications*, MoMuC, 2000.

[33] Karim El Defrawy, Aur elien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *ISOC Network and Distributed System Security Symposium*, NDSS, 2012.

[34] Trusted computing group (TCG). TPM 2.0 library specification, 2014. http://www.trustedcomputinggroup.org/resources/tpm_library_specification.

[35] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. BIOS chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, 2013.

[36] Intel trusted execution technology (Intel TXT) – software development guide. https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[37] AMD secure virtual machine architecture reference manual. https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.

[38] Getting around non-executable stack (and fix). Bugtraq, 1997.

[39] H Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, 2007.

[40] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2011. ISBN 978-1-4503-0564-8. doi: 10.1145/1966913.1966919. URL http://doi.acm.org/10.1145/1966913.1966919.

[41] PaXTeam. PaX: twelve years of securing Linux. In *LATINOWARE*, 2012.

[42] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: SoftWare-based ATTestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE S&P, 2004.

[43] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: Verifying the Integrity of PERipherals' firmware. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, 2011.

[44] Fengwei Zhang, Haining Wang, Kevin Leach, and Angelos Stavrou. A framework to secure peripherals at runtime. In *European Symposium on Research in Computer Security*, ESORICS, 2014.

[45] Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote attestation on program execution. In *Proceedings of the ACM Workshop on Scalable Trusted Computing*, STC, 2008.

[46] Aurélien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, 2014.

[47] N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. SEDA: Scalable embedded device attestation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[48] Kara K. W. Poon, Steven J. E. Wilton, and Andy Yan. A detailed power model for field-programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(2), 2005.

[49] Jason H. Anderson and Farid N. Najm. Power estimation techniques for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10), 2004.

[50] Fei Li, Yan Lin, Lei He, Deming Chen, and Jason Cong. Power modeling and characteristics of field programmable gate arrays. 24(11), 2005.

[51] Chunjie Duan, Victor Cordero, and Sunil P. Khatri. Efficient on-chip crosstalk avoidance CODEC design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2009.

[52] SPICE circuit simulator. https://embedded.eecs.berkeley.edu/pubs/downloads/spice/index.htm.

[53] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A tool to model large caches, 2009.

[54] Jeffrey B. Goeders and Steven J. E. Wilton. VersaPower: Power estimation for diverse FPGA architectures. In *International Conference on Field-Programmable Technology (FPT)*, 2012.

[55] Edin Kadric, David Lakata, and Andr DeHon. Impact of memory architecture on FPGA energy consumption. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, 2015.

[56] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, VLSI Design, 1996.

[57] Peter Zipf, Heiko Hinkelmann, Lei Deng, Manfred Glesner, Holger Blume, and Tobias G. Noll. A power estimation model for an FPGA-based softcore processor. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, FPL, 2007.

[58] Lucile Senn, Eric Senn, and Christian Samoyeau. Modelling the power and energy consumption of NIOS II softcores on FPGA. In *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, 2012.

[59] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO, 1999.

[60] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, 2007.

[61] Kocher, Paul, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[62] Carolyn Whitnall and Elisabeth Oswald. Robust profiling for DPA-style attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *CHES*. 2015.

[63] Tempest: A signal problem. *NSA Cryptologic Spectrum*, 1972.

[64] W van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers and Security*, 4, 1985.

[65] Eavesdropping by visual vibrations. http://newsoffice.mit.edu/2014/algorithm-recovers-speech-from-vibrations-0804.

[66] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE S&P, 2013.

[67] Ryan M. Gerdes, Thomas E. Daniels, Mani Mina, and Steve F. Russell. Device identification via analog signal fingerprinting: A matched filter approach. In *ISOC Network and Distributed System Security Symposium*, NDSS, 2006.

[68] Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware trojan design and implementation. In *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on*, HOST, 2009.

[69] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. Building a side channel based disassembler. In *Transactions on Computational Science X*. 2010.

[70] Jean-Jacques Quisquater and David Samyde. Automatic code recognition for smart cards using a kohonen neural network. In *Smart Card Research and Advanced Application Conference*, 2002.

[71] Dennis Vermoen, Marc Witteman, and Georgi N. Gaydadjiev. Reverse engineering Java card applets using power analysis. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems: First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop, WISTP 2007, Heraklion, Crete, Greece, May 9-11, 2007. Proceedings*, 2007.

[72] Martin Goldack. Side-channel based reverse engineering for microcontrollers. Master's thesis, Ruhr-Universität Bochum, Germany, 2008.

[73] Carlos R. Aguayo Gonzalez. *Power Fingerprinting for Integrity Assessment of Embedded Systems*. PhD thesis, Virginia Polytechnic Institute and State University, 2011.

[74] Mehari Msgna, Konstantinos Markantonakis, David Naccache, Mayes, and Keith. Verifying software integrity in embedded systems: A side channel approach. In *Constructive Side-Channel Analysis and Secure Design*, LNCS. 2014.

[75] Carlos R. Aguayo Gonzalez and Jeffrey H. Reed. Power fingerprinting in SDR & CR integrity assessment. In *IEEE Military Communications Conference (MILCOM)*, 2009.

[76] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Kevin Fu, and Wenyuan Xu. WattsUpDoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *Proceedings of the*

2013 USENIX Conference on Safety, Security, Privacy and Interoperability of Health Information Technologies, HealthTech, 2013.

[77] Carlos Moreno, Sebastian Fischmeister, and M. Anwar Hasan. Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES, 2013.

[78] Yong Yang, Lu Su, Mohammad Khan, Michael Lemay, Tarek Abdelzaher, and Jiawei Han. Power-based diagnosis of node silence in remote high-end sensing systems. *ACM Transactions on Sensor Networks*, 11(2), 2014.

[79] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2013.

[80] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293, Oct 2011. ISSN 2190-8516. doi: 10.1007/s13389-011-0023-x. URL https://doi.org/10.1007/s13389-011-0023-x.

[81] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, CHES. 2003.

[82] Victor Lomne, Emmanuel Prouff, and Thomas Roche. Behind the scene of side channel attacks. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2012.

[83] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition*. Academic Press, 4th edition, 2008.

[84] DPA contest. http://www.dpacontest.org/.

[85] Lejla Batina, Jip Hogenboom, and Jasper G.J. van Woudenberg. Getting more from PCA: First results of using principal component analysis for extensive power analysis. In *Topics in Cryptology – CT-RSA 2012: The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 – March 2, 2012. Proceedings*, 2012.

[86] Hong Liu, Hongmin Li, and Eugene Y. Vasserman. Practicality of using side-channel analysis for software integrity checking of embedded systems. In *Security and Privacy in Communication Networks: 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, SecureComm '15, pages 277–293. Springer, 2015.

[87] ilvinas Nakutis. Embedded systems power consumption measurement methods overview, 2009.

[88] PICmicro mid-range MCU family – reference manual. Microchip Technology Inc., 1997, http://ww1.microchip.com/downloads/en/DeviceDoc/31000a.pdf.

[89] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *CHES*. Springer, 2004.

[90] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. Wiley, 5th edition, 2012.

[91] Hong Liu and Eugene Y. Vasserman. Gray-box software integrity checking via side-channels. In *Security and Privacy in Communication Networks*, SecureComm '17. Springer, 2017.

[92] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, February 2008. ISSN 1551-3939. doi: 10.1561/1000000005. URL http://dx.doi.org/10.1561/1000000005.

[93] Sugawara, Takeshi, Daisuke Suzuki, Minoru Saeki, Mitsuru Shiozaki, and Takeshi Fujino. On measurable side-channel leaks inside ASIC design primitives. *Journal of Cryptographic Engineering*, 4(1):59–73, 2014.

[94] Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proceedings of the USENIX Workshop on Smartcard Technology*, USENIX Smartcard, 1999.

[95] Henry W. Ott. *Electromagnetic Compatibility Engineering*. Wiley, 2009.

[96] NIOS II processor reference handbook. https://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[97] FreeRTOS. http://www.freertos.org/.

[98] The OpenMSP430 project. http://opencores.org/download,openmsp430.

[99] CoreMark. http://www.eembc.org/coremark/.

[100] Dakshi Agrawal, Bruce Archambeault, Josyula Rao, and Pankaj Rohatgi. The EM side-channel(s). In *Cryptographic Hardware and Embedded Systems (CHES)*, CHES. 2003.

[101] Eric Peeters, Franois-Xavier Standaert, and Jean-Jacques Quisquater. Power and electromagnetic analysis: Improved model, consequences and comparisons. *Integration, the VLSI Journal*, 40(1):52–60, 2007.

[102] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech, 2nd edition, 2000.

[103] PowerPlay early power estimator user guide. https://www.altera.com/literature/ug/ug_epe.pdf.

[104] LLdiko E. Frank and Jerome H. Friedman. A statistical view of some chemometrics regression tools. *Technometrics*, 35(2), 1993.

[105] L. Bohy, M. Neve, D. Samyde, and J.-J. Quisquater. Principal and independent component analysis for crypto-systems with hardware unmasked units. In *Proceedings of e-Smart 2003*, 2003.

[106] Experiment setup and data. http://people.cs.ksu.edu/~hongl/fpga/.

[107] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *Cryptographic Hardware and Embedded Systems (CHES)*, CHES. 2005.

[108] Michael Kasper, Werner Schindler, and Marc Stttinger. A stochastic method for security evaluation of cryptographic FPGA implementations. In *International Conference on Field-Programmable Technology*, FPT, 2010.