Platform Verification and Secure Program Execution in Embedded Devices

Mehari Gebrehaweriya Msgna

Thesis submitted to the University of London for the degree of Doctor of Philosophy

Information Security Group
Department of Mathematics
Royal Holloway, University of London

2015

Declaration

This doctoral study was conducted under the supervision of Dr. Konstantinos Markantonakis.

The work presented in this thesis is the result of original research carried out by myself whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Signature .... Mehari Gebrehaweriya Msgna
Date:

Publications

Mehari Msgna, Raja Naeem Akram, Konstantinos Markantonakis and Keith Mayes, "Subscriber Centric Conditional Access System for Pay-TV Systems", in IEEE 10th International Conference on e-Business Engineering (ICEBE), *pages* 450-455. Coventry, United Kingdom: IEEE Computer Society, September 11-13, 2013.

Mehari Msgna and Colin Walter, "An Overview of PIC Microcontrollers and Their Suitability for Cryptographic Algorithms", *chapter* in Secure Smart Embedded Devices, Platforms and Applications. Konstantinos Markantonakis and Keith Mayes (*Eds*). Springer, 2013.

Mehari Msgna, Konstantinos Markantonakis and Keith Mayes, "The B-Side of Side Channel Leakage: Control Flow Security in Embedded Systems", in The 9th International Conference on Security and Privacy in Communication Networks (SecureComm), *series* LNICST. Tanveer Zia, Albert Zomaya, Vijay Varadharajan and Morley Mao (*Eds.*), *volume* 127. Sydney, Australia: Springer, September 2013.

Mehari Msgna, Konstantinos Markantonakis, David Naccache and Keith Mayes, "Verifying Software Integrity in Embedded Systems: A Side Channel Approach", in The 5th International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE), *series* LNCS. Emmanuel Prouff (*Eds.*), *volume* 8622. Paris, France: Springer, April 2014.

Mehari Msgna, Konstantinos Markantonakis and Keith Mayes, "Precise Instruction-Level Side Channel Profiling of Embedded Processors", in The 10th International Conference Information Security Practice and Experience (ISPEC), *series* LNCS. Xinyi Huang and Jianying Zhou (*Eds.*), *volume* 8434. Fuzhou, China: Springer, May 2014.

Konstantinos Markantonakis, Raja Akram and Mehari Msgna, "Secure and Trusted Application Execution on Embedded Devices" in The 8th International Conference on Security for Information Technology and Communications (SECITC'15), *series* LNCS. David Naccache and Emil Simion (*Eds.*). Bucharest, Romania: Springer, June 2015.

Acknowledgement

This thesis would not have been possible without the support of a number of people.

My initial and foremost thanks goes to my family. Especially, Samrawit Abegaz for being so patience throughout my study. I would also like to thank my supervisor Dr. Konstantinos Markantonakis and Prof. Keith Mayes for all their support and guidance.

I would like to offer my gratitude to Sheila Cobourne for her wonderful reviews of this thesis. During my stay in Royal Holloway, University of London I have met many people who inspired me in so many ways, I would like to convey my thanks to all of you.

Abstract

Recently, hardware manufacturers are increasingly outsourcing their production process into countries with lower cost structure. Although this reduces the cost of hardware production, it also creates opportunity for attackers to hack into the supply chain and change the original design of the hardware components. Such changes could range from short circuiting a module (for instance random number generators) to inserting parasite circuits and new masks (such as hardware Trojan circuits). These kind of intrusions are difficult to detect through pure functional testing. Furthermore, attacks on runtime program attributes (eg. fault injection attacks) are increasing in number and sophistication.

In this thesis we propose techniques for platform verification and secure program execution that can be used in low-end to medium-end embedded systems. Our design incorporates a pre-deployment device verification and dedicated security module that monitors the program's properties during execution. Both our pre-deployment and runtime verification methods constitute compile and execution time computations to reduce the time required for security checks during runtime.

In the core of this thesis, we analyse the current threats to the embedded systems platform and programs. This leads to two major contributions spanning the pre and post integration of embedded systems into the larger electronic equipment. We propose side channel based pre-deployment platform verification techniques. In our techniques we use instruction and basic block level side channel templates to identify anomalies within the target platform. Our approach does not require prior detailed knowledge of the inner workings of the program or the platform under test. Furthermore, we also propose the design of a generic runtime secure program execution architecture. Our proposal protects the target program's runtime data, instructions and control flow jumps during its execution. To achieve this goal without affecting the performance of the main processor we introduce a dedicated hardware module. Finally, we provide the test implementations of our proposals along with their performance measures.

Contents

1 Introduction **17**

1.1 Setting the Scene 18

1.2 The Evolution of Embedded Systems 18

1.3 Motivation and Challenges 22

1.4 Contributions 23

1.5 Thesis Structure 26

2 Embedded Systems Architecture **29**

2.1 Introduction 30

2.2 Embedded Systems Architectures 32

2.2.1 Microprocessor Unit 32

2.2.2 Memory 34

2.2.3 Input/Output 35

2.2.4 Bus 35

2.2.5 Supporting Devices 36

2.3 Characteristics of Embedded Processors 36

2.3.1 Design Parameters 36

2.3.2 Classification 37

2.4 Application Development Tools 40

2.4.1 Source Code Editor 40

2.4.2 Compiler 41

2.4.3 Debugger 41

2.4.4 Simulator 42

2.4.5 Device Programmer 42

2.5 Example Applications 43

2.6 Summary 44

3 Attacks on Embedded Systems **45**
3.1 Introduction . . . 46
3.2 Invasive Attacks . . . 47
3.2.1 Delayering . . . 47
3.2.2 Block Localisation . . . 48
3.2.3 Microprobing . . . 50
3.2.4 Circuit Extraction . . . 50
3.2.5 Circuit Modification . . . 51
3.3 Semi-Invasive Attacks . . . 52
3.3.1 Local Heating . . . 52
3.3.2 Ultraviolet Attacks . . . 53
3.3.3 Optical Emission Analysis . . . 53
3.4 Non-Invasive Attacks . . . 54
3.4.1 Side Channel Analysis . . . 54
3.4.2 Data Remanence . . . 56
3.4.3 Fault Injection . . . 56
3.5 Summary . . . 57

4 Security In Embedded Systems **58**
4.1 Introduction . . . 59
4.2 Code Hardening . . . 60
4.3 Side Channel Protection . . . 62
4.4 Tamper Resistance . . . 62
4.5 Redundant Execution . . . 64
4.6 Trusted Platform Module (TPM) . . . 65
4.7 ARM's TrustZone . . . 66
4.8 GlobalPlatform . . . 67
4.8.1 Application Management . . . 68
4.8.2 Trusted Execution Environment . . . 68
4.9 Remaining Security Challenges . . . 69
4.9.1 Post-production Pre-deployment Device Verification . . . 70
4.9.2 Runtime Secure Execution . . . 71
4.10 Summary . . . 72

I Post-production Pre-deployment Measures 74

5 Control Flow Verification 75
5.1 Introduction 76
5.2 Device Modelling 78
5.2.1 Model Parameters 79
5.2.2 Principal Components Analysis (PCA) 82
5.2.3 Fisher's Linear Discriminant Analysis (F-LDA) 83
5.3 Control Flow Reconstruction 84
5.4 Control Flow Verification 85
5.5 Implementation and Results 86
5.5.1 Model Parameters 87
5.5.2 Calculating The Most Probable State Sequence 90
5.5.3 Verifying The Reconstructed State Sequence 91
5.6 Summary 91

6 Software Integrity Verification 93
6.1 Introduction 94
6.2 Instruction-Level Template Construction 95
6.3 Dimensionality Reduction 97
6.3.1 Sum of Difference of Means 97
6.3.2 Means-Variance 97
6.3.3 Means-PCA 98
6.4 Instruction Classification 98
6.4.1 Multivariate Gaussian Probability Density Function 99
6.4.2 k-Nearest Neighbors Algorithm 99
6.5 RSA Signature Screening Algorithm 100
6.6 Basic Block Integrity Verification 101
6.7 Implementation and Results 104
6.7.1 Instruction-Level Template Construction 105
6.7.2 Dimensionality Reduction 107
6.7.3 Instruction Classification 109
6.7.4 Basic Block Integrity Verification 113
6.8 Summary 115

II Runtime Secure Execution **117**

7 Program Data Security **118**
7.1 Introduction . . . 119
7.2 Related Work . . . 121
7.3 Attack Model . . . 122
7.4 Dual-Stack Architecture . . . 123
7.4.1 TwinStack: Redundant Stack Operations . . . 124
7.4.2 IntegrityStack: Verifying Integrity of Stack Items . . . 125
7.5 Implementation and Analysis . . . 127
7.5.1 Implementation . . . 127
7.5.2 Performance Overhead . . . 130
7.5.3 Latency . . . 131
7.5.4 Attack Detection Capability Evaluation . . . 132
7.6 Summary . . . 132

8 Program Instructions and Control Flow Security **135**
8.1 Introduction . . . 136
8.2 Basic Blocks . . . 138
8.3 Control Flow . . . 139
8.3.1 Inter-Procedural Control Flow . . . 139
8.3.2 Intra-Procedural Control Flow . . . 142
8.4 Instructions Integrity . . . 144
8.5 Implementation . . . 147
8.5.1 Lookup Tables . . . 147
8.5.2 Hash Computation . . . 149
8.6 Summary . . . 150

9 Conclusion and Future Work **152**
9.1 Summary and Conclusions . . . 153
9.2 Recommendation for Future Research . . . 156

Bibliography **156**

A Selected AVR Instructions **181**

List of Figures

1.1 Embedded device hardware and software life cycle 24

2.1 General architecture of embedded processor 32
2.2 Classification criteria of microcontrollers 39
2.3 Example of source code editor software. 41
2.4 PIC simulator IDE, a software designed to simulate a range of PIC microcontrollers. 42

3.1 Different blocks of a decapsulated Infineon SLE66 [1]. 49
3.2 Probing needles on a decapsulated semiconductor chip [2]. 50
3.3 Taking microscopic pictures of the inner circuitry of a semiconductor chip [2]. 51

4.1 A top layer sensor mesh setup and practical implementation [3]. . 63
4.2 A clear and scrambled bus lines [4]. 63
4.3 A simple hardware redundant execution setup. 64
4.4 A simple time redundant execution setup. 65
4.5 Generic architectural view of ARM TrustZone 67

5.1 A Hidden Markov Model representation of a device executing a program with five states (A, B, C, D and E). The power consumption is the observable output that reveals partial information about the executed states. 79
5.2 Integer and boolean branching statements in code segment with 3 states. 81
5.3 Test program's control flow diagram. 87
5.4 High-level description of the test program 87

5.5 Mean of the power traces of the states illustrated in Fig. 5.3. 89
5.6 Original data after PCA. 89
5.7 Original data after F-LDA. 90

6.1 Basic block integrity verification block diagram 102
6.2 Power consumption waveform of selected ATMega163's one clock cycle instructions (`NOP`, `MOV`, `ADD` and `SUB`). 106
6.3 Power consumption waveform of selected ATMega163's two clock cycle instructions (`MUL`, `ST` and `LD`). 107
6.4 Sum of difference of means. 108
6.5 Overall variance of the original data accounted for the first 15 principal components of the instructions `NOP`, `MOV`, `CLR` and `ADD`. . 108
6.6 Classification rate after dimensionality reduction using *Multivariate Gaussian Probability Density Function* for all 39 instructions. 110
6.7 112
6.8 Instruction recognition rate for all 39 instructions using *K-Nearest Neighbors Algorithm* for k=1 after applying the dimensionality reduction techniques. 113
6.9 Recognition rate for all 39 instructions using *K-Nearest Neighbours Algorithm* with different distance functions for k=1 after applying PCA. 114
6.10 Power consumption of the processor when executing PIN checking application with embedded processor parameter update in between the basic blocks. 114

7.1 Fault Injection: Attack example on stack items. 120
7.2 Double stack architecture 124
7.3 Hardware implementation of dual-stack and attack simulator . . . 128
7.4 Normal single stack VHDL implementation 128
7.5 TwinStack VHDL implementation 129

8.1 Assembly code segment and its basic blocks. 138
8.2 Control flow diagram of function A. 140
8.3 Hash generating scheme. 146
8.4 Davies-Meyer hash generation scheme. 147

8.5 Content Addressable Memory (CAM) architecture. 148
8.6 256 byte Content Addressable Memory (CAM). 149
8.7 AES implementation in VHDL. 150

List of Tables

2.1 CISC, RISC and equivalent Pseudo- Code for Multiplying two Memory Contents . . . 38
2.2 A list of embedded system applications. . . . 44

5.1 Transition probability distribution of the program illustrated in Figure 5.3. The columns represent next states and the rows represent current states. . . . 88

6.1 Percentage of true (**bold**) and false positive recognition rate for a selected instructions using MVGPDF. The rows and columns represent executed and recognised instructions respectively. . . . 109
6.2 Percentage of true (**bold**) and false positive recognition rate for a selected instructions using kNN. The rows and columns represent executed and recognised instructions respectively. . . . 111
6.3 Average percentage of true (**bold**) and false positive recognition rate for a selected instructions using kNN in k-fold cross validation. 111

7.1 Number of additional instructions introduced by the countermeasure 131
7.2 Overall increase in executed instructions of software implementation of the countermeasure . . . 131
7.3 Latency measurements of the countermeasures . . . 132
7.4 Comparison of the proposed protection techniques with other related works . . . 133

8.1 Function property lookup table $TABLE_{property}$ . . . 141
8.2 Function property lookup table $TABLE_{property}$ for example function. 141
8.3 Basic block information lookup table ($TABLE_{BasicBlock}$) . . . 143

8.4 Basic block information lookup table for function B. 144

A.1 AVR's instructions seleted for our experiments. 183

List of Abbreviations

ADC	Analogue to Digital Converter
AES	Advanced Encryption Standard
AIK	Attestation Identity Key
ALU	Arithmetic Logic Unit
APDU	Application Protocol Data Unit
ASIC	Application Specific Instruction Set Computer
ATM	Automated Teller Machine
AU	Arithmetic Unit
CA	Certificate Authority
CAM	Content Addressable Memory
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CM	Control Module
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CSE	Common Subexpression Elimination
CT	Computerized Tomography
CU	Control Unit
DAC	Digital to Analogue Converter
DES	Data Encryption Standard
DFA	Differential Fault Analysis
DPA	Differential Power Analysis
DVD	Digital Versatile Disc
ECG	Electrocardiogram
ECU	Electronic Control Unit
EEG	Electroencephalography

EEPROM	Electrically Erasable Programmable Read Only Memory
EFP	Environmental Failure Protection
EFT	Environmental Failure Testing
EK	Endorsement Keys
EMG	Electromyography
FCG	Function Call Graph
FIFO	First In First Out
F-LDA	Fisher-Linear Discriminant Analysis
FIPS	Federal Information Processing Standard
GSM	Global System for Mobile
HMM	Hidden Markov Model
IC	Integrated Circuit
IDE	Integrated Development Environment
IO	Input/Output
IR	Instruction Register
ISS	Instruction Set Simulator
JCVM	Java Card Virtual Machine
kNN	k-Nearest Neighbours
LU	Logic Unit
LM	Lunar Module
MAR	Memory Address Register
MDR	Memory Data Register
MISR	Multiple Input Shift Register
MPU	Microprocessor Unit
MRI	Magnetic Resonance Imaging
MVGPDF	Multivariate Gaussian Probability Distribution Function
NFC	Near Field Communication
OCR	Optical Character Recognition
OS	Operating System
PC	Personal Computer
PCA	Principal Components Analysis
PCR	Platform Configuration Register
PDA	Personal Digital Assistant
PIN	Personal Identification Number

RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RSA	Rivest Shamir Adleman
SEM	Scanning Electron Microscope
SHA	Secure Hash Algorithm
SPA	Simple Power Analysis
SR	Status Register
SRAM	Static Random-Access Memory
SRK	Storage Root Key
TEE	Trusted Execution Environment
TEM	Transmission Electron Microscope
TPA	Template Power Analysis
TPM	Trusted Power Module
TV	Television
UV	Ultraviolet
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

Contents

1.1 **Setting the Scene** 18

1.2 **The Evolution of Embedded Systems** 18

1.3 **Motivation and Challenges** 22

1.4 **Contributions** 23

1.5 **Thesis Structure** 26

In this chapter, we provide a brief discussion on the evolution of embedded systems. We then deliberate on the motivation and challenges of the thesis. This is followed by the contributions of the thesis to the knowledge on security of embedded systems and program execution. Finally, the chapter concludes by outlining the structure of the thesis with a short description of all subsequent chapters.

1.1 Setting the Scene

We start the discussion of this chapter with the history of how embedded systems technology has evolved over the years since its inception. We then explain our motivations for pursuing this research topic and all the challenges that this involves. We also point out our rationale for implementing our security countermeasure techniques. This is followed by a brief discussion of the contributions of this thesis. Finally, we finish the chapter by outlining the structure for the rest of thesis.

1.2 The Evolution of Embedded Systems

An embedded system is a complex computing machine made up of multiple modules, where each module can be considered a separate invention. In 1936 Zuse, a German civil engineer, designed the first relay computer, also known as *Z1 Computer* [5]. The *Z1 Computer* and consecutive computer designs of the 1940's were dedicated for a single task. These computers had instructions made to accomplish a specific task that the computer was made to be used. Each computer had a different binary-code program called a machine language that told it how to operate [6]. Another feature of early computing systems was that they were too large to be considered "embedded".

In 1947 an important invention occurred, "*The Transistor*" [7]. A transistor is not a computer but has greatly influenced the evolution of a computing system. In 1958 Jack Kilby and Robert Noyce presented their work at a conference in Washington, DC about a solid block of electronic equipment without connecting wires. This became to be known as the first Integrated Circuit (IC) or chip [8]. Over time the concept of re-programmable computing systems was developed from a combination of computing technology, solid state device and traditional electromechanical sequence. This led to the emergence of sophisticated but smaller ICs, such as embedded systems.

One of the very first modern embedded systems was the Apollo Guidance Computer designed by Charles Stark Draper [9]. The Apollo Guidance Computer provided computation and electrical interface for guidance, navigation and control of the spacecraft and it was installed on each of the Apollo Control Module (CM) and Lunar Module (LM). It had 16 bit word with 15 bits of data and one parity bit. It also made use of both erasable and permanent memory blocks. The erasable memory block was used to store intermediate results of communication, such as the location of spacecraft, while the program data that did not need changing was stored on the permanent memory.

In 1961 Boeing[1] released the Minuteman I missile. Inside the missile was the Autonetics D-17 guidance computer system [10]. The D-17 was a small serial-binary computer designed for general purposes [11]. When the Minuteman II missile went into mass-production in 1966, the D-17 was replaced with a newer embedded system design, making it the first high-volume production and use of embedded integrated circuit.

After these early and isolated embedded system applications, their price came down, permitting their use in commercial products. In 1969 Nippon Calculating machines approached Intel for the design of 12 custom chips for its new 141-PF printing calculator. However, Intel's engineers suggested a set of four chips called the MCS-4[2]. The MCS-4 contained 4 chips which were the Central Processing Unit (CPU) - the Intel 4004 -, Read-only memory (ROM), random access memory (RAM) and a shift register chip for IO purposes. Intel then launched the Intel 4004 with an advertisement in the November 15, 1971 issue of the Electronic News titled "Announcing the new era in integrated electronics" [12]. That made the Intel 4004 the first general purpose microprocessor that someone can buy from the market and customise it to perform their desired operation.

Since then the size of microprocessors decreased while the number of transistors and computing power increased significantly. In 1965 Gordon Moore in his pa-

[1] Boeing is one of the largest and leading aerospace companies and manufacturer of commercial jet liners, defence, space and security systems.

[2] MCS-4 is a chip-set designed by Intel in the early 1970s. It contained the 4001 ROM, 4002 RAM, 4003 Shift Register and 4004 processor.

per [13], predicted that the number of transistors per integrated circuit will be doubled roughly every year and will continue to do so for at least a decade. The Intel 4004 was the size of a fingernail but delivered the same computing power as the 1940's computer designs [14] which filled an entire room. Intel 4004 had 2,300 transistors and a circuit line width of 10,000 nanometers. By 2010 an Intel-core processor holds 560 million transistors with a circuit line width between 32 and 45 nanometers. This trend of transistor miniaturisation led to integration of additional functionalities into the chip without having to increase the size of the device, leading to the emergence of the first microcontroller MCS-48 in 1976 [15].

The emergence of the first microcontroller, followed by the continuous increase in sophistication and decrease in size of integrated circuits fuelled the next breakthrough in the evolution of embedded systems. This time it came from Germany and France, not from America. The French Postes, Télégraphes et Téléphones (PTT) deployed integrated circuits embedded in a plastic card as telephone cards [16]. The Germans telephone companies soon followed. These deployments provided a testing ground for embedded systems in a new application area, which was later exported to other industries, as chip based cards provided increased reliability and security.

The early versions of these cards were limited storage capacity memory cards based on a simple fixed logic circuit. However, later in the 1990's microprocessor cards start to emerge onto the scene. These cards can store information as well as dynamically process the stored information without relying on a fixed logic circuit. The German post office conducted the first trials of microprocessor cards for their analogue mobile telephone network. Their purpose was to authenticate users when they join the network to avoid the cloning of mobile phones. The success story of these trials resulted in the deployment of such cards in the GSM[3] network. At the time, telecommunication companies all over the world were rapidly adopting microprocessor cards as telephone cards. However, the banks of the time did not embrace the microprocessor cards quickly.

The development of microprocessor cards coincided with another breakthrough

[3] Global System for Mobile Communication (GSM) is a standard for mobile telecommunication industry that is developed and promoted by the GSM Association (GSMA)

in the field of system security and mathematics. The study of cryptography was just emerging from government and military secrecies to the public. The security offered by sophisticated cryptographic algorithms coupled with an improved hardware design paved the way for microprocessor cards to be used as a security tokens. This gave microprocessor cards the edge as a secured token over other technologies. Shortly afterwards the banks followed the footprints of Telecom companies in deploying microprocessor cards as bank cards.

The development of such cards was further facilitated in 1996 when engineers from an IT technology provider (Schlumberger in Austin, Texas, United States of America) designed a microprocessor card that can support a subset of the Java[4] programming language [17]. Such cards are also known as Java Card[5]. This gave birth to the more sophisticated multi-application microprocessor cards, which support the existence and execution of multiple applications from different providers, such as the latest Java Card 3.0 [18] and MULTOS [19]. According to [20], 9 billion Java Cards were deployed since 1998, making it the most widely used embedded system in the world.

The proliferation of such cards into many other applications (for example, the electronic identity cards, passports and driving license) coupled with the advancements in microelectronics helped embedded systems to become one of the most widely deployed computing system. This is even further strengthened by the technological progress in smartphones and the emergence of new technologies (such as the NFC [21] and the Internet of Things [22]). By 2017 two billion NFC enabled devices are predicted to be shipped [23]. Nearly three billion low-end (4, 8 and 16 bit) processors are sold every year [24]. According to [25] a person uses 250 chips or one billion transistors each day. This data was published in 2008 and it is a truism that these numbers will only increase with time.

[4] A general purpose programming language designed to create programs capable of running on any computer.

[5] Java Card is a multiplication microprocessor card platform which supports subset of the Java programming language and is promoted by the JavaCard Forum.

1.3 Motivation and Challenges

A wide range of computing devices are being introduced into our modern life style, performing different tasks, for example, alarm clocks, smartphones, smart watches but also more security critical devices such as bank cards and physical access control systems. With the advancement of the Internet and Internet based services, such devices are increasing in number and sophistication. Furthermore, they are increasingly involved in collecting and processing big amount of data that is paramount for our security. For instance, cyber-physical systems that are used in traffic monitoring and controlling systems.

The security and privacy concerns of users are increasing with the increasing number of different devices through which users access associated services and information. To curve these concerns, several techniques have been proposed to provide hardware-based privacy and security protection. These proposals differ in operation and capability from one system (or device) to another. In other words, a proposed technique's security and privacy architecture becomes specific to the target device. For instance, the difference between the Trusted Platform Module (TPM) [26] and Mobile Trusted Module (MTM) [27] is that they target two different computing systems; a general-purpose computing device and a mobile phone respectively. Other similar solutions are ARMs TrustZone [28] and GlobalPlatform Trusted Execution Environment (TEE) [29].

Having a wide range of security protection options is good but it also means that service providers and system manufacturers, that require security and privacy, have to implement and support a wide range of technologies. In addition, over time several devices have evolved from the original purposes that they were designed for. For instance, initially mobile phones were first designed for voice and later text communications. Similarly, televisions were designed for news and drama but now they are also used to access the Internet. This means in addition to their traditional services they now need to implement internet security techniques as well and the same goes to a number of other devices. The diversity in protection methods creates a complex issue of implementating, integrating, using and managing, not only for the service providers but also the consumers.

A possible solution to the aforementioned challenges is to have a unified hardware-based security and privacy architecture that can be used across different computing platforms (e.g. smart cards, mobile phones, televisions and other embedded devices). We divide the security of embedded systems into two phases; post-production pre-deployment verification and runtime secure execution. The principle is that the underlying device's processor will have the same hardware-based security architecture that can be used to securely execute all applications regardless of their nature. For example, service providers can offer their services on mobile phones, tablets and televisions without a major design change.

1.4 Contributions

In this thesis we set out to analyse the security of embedded systems. We start this by looking into the life cycle of an embedded device. Figure 1.1 shows the different stages of an embedded device during its life cycle. An embedded system contains a hardware and software components. Normally, both components have to go through a number of stages before they can be integrated. The design, synthesis and production stages of the hardware component refer to the pre-manufacturing phases of the integrated circuit. Similarly, the code generation and compilation are the code writing and transformation of the software respectively.

After reviewing the current security threats and countermeasures deployed in embedded devices, we consider modifications to a number of stages within the life cycle of an embedded device. The modified stages are highlighted in red in Figure 1.1. The modifications are spread over different stages of an embedded device's life cycle, from designing the integrated circuit to post-deployment program execution. The questions of how these stages should be modified are the focus of this thesis. Contributions of this thesis extend from chapter 5 to chapter 8.

The main contributions of this thesis are divided into pre-deployment and post-deployment stages. From here on we refer to the pre-deployment and post-deployment changes as platform verification and secure program execution. The pre-deployment platform verification is concerned with identifying undesirable

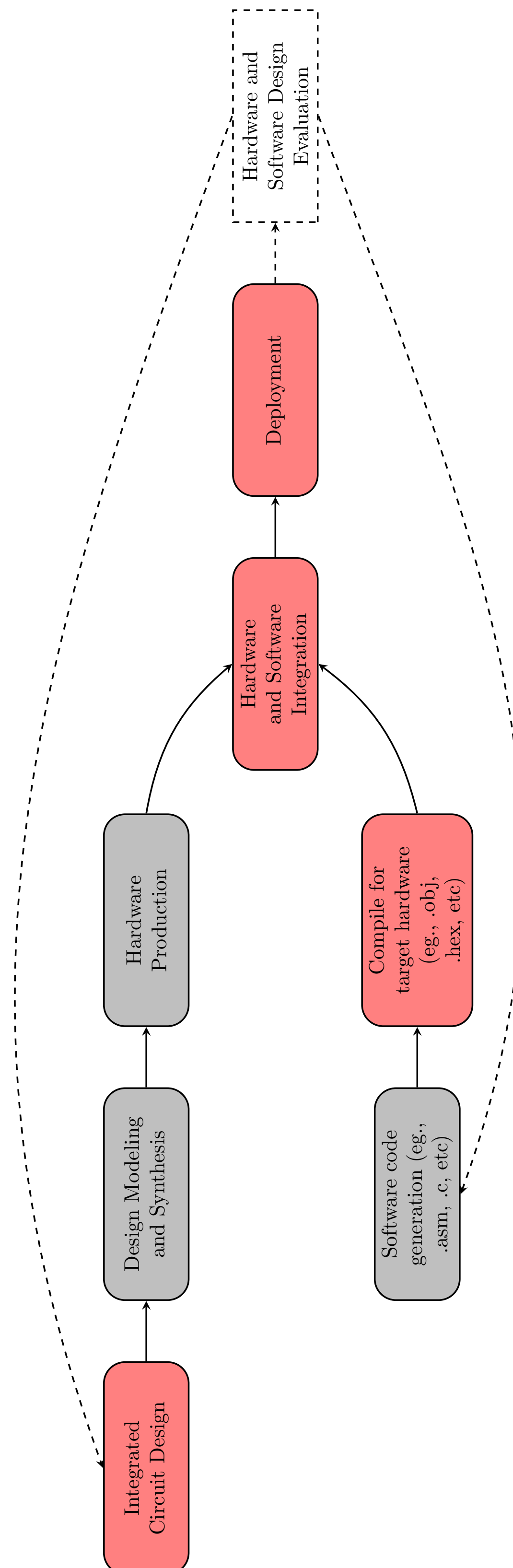


Figure 1.1: Embedded device hardware and software life cycle

platforms or platform components that can not be detected by only *purely functional*[6] tests. Such threats include counterfeits and hardware Trojans. The runtime secure program execution is tasked with ensuring the programs are executed securely during runtime.

Counterfeits and hardware Trojans are designed to be difficult to detect with purely functional tests. Thus we propose side channel based techniques to verify the hardware and software platforms. First we propose a power consumption based pre-deployment program control flow verification. Then we propose basic block instructions integrity verification. This involves the creation of instruction and basic block level template creation. In addition, it also requires precise template recognition methods. The principle behind this technique is, the user creates instruction and basic block level side channel templates using a few legitimate devices. Later, using these templates he/she verifies the integrity of other devices before they are assembled into the final product. The result of these works are published in international conferences and the papers can be accessed in [30, 31]. In addition, we improved the previously known highest instruction based side channel template matching success, 70.1% [32] on PIC microcontrollers [33], to 100%. Details of this work in available at [34].

The above proposal is to verify a device before it is integrated into the final product. The second part of this thesis is protecting the programs against threats that may come once the product is operational. This deals with program control flow, integrity of executed instructions and runtime program data. To that effect we propose a number of changes to the traditional design of embedded devices. We propose an integrated watchdog module as part of the process design. This watchdog takes additional information generated by a compiler and uses it to ensure the security of the control flow jumps and executed instructions at runtime. To protect the runtime data we propose a dual stack processor architecture, where the second stack is used to ensure the integrity of items in the main stack. We also proposed a dedicated hardware subsystem to verify the control flow and instructions integrity. The module takes input from a compiler generated program attribute and the main processor. Details of these proposals are discussed in the

[6] A test designed to verify the conformance between a device and its specification.

second part of this thesis. Details of these proposals and their implementations is presented in Chapters 7 and 8.

1.5 Thesis Structure

The rest of the thesis is organised as follows:

In chapter 2 we begin the discussion with a brief description of embedded systems. With the ever-changing characteristics of embedded systems discussion on the basic architecture of embedded devices is needed. Therefore, in this chapter we define embedded systems. We then explain the underlying architecture and the different components that it comprises. Following that we discuss the design characteristics and application development process of embedded systems. Finally, we present a list of practical examples.

Attacks on embedded systems are increasing in number and sophistication. These attacks are generally divided into three main categories; invasive, semi-invasive and non-invasive attacks. Invasive attacks physically modify the target device irrecoverably. These changes range from removing the insulation layer of an IC, reverse engineering to reconfiguring the underlying circuit. Non-invasive attacks only require exposure to the surface of the target device but do not disrupt the device's normal operations. Semi-invasive attacks are in between invasive and non-invasive attacks. In chapter 3 we discuss these attacks in detail and provide a number of examples for each category.

Due to the recent increasing in number and sophistication of attacks in embedded systems several countermeasures have been proposed. The countermeasures range from modifying the source code, using dedicated external devices to re-designing the processor architecture. In chapter 4 some of the most commonly utilised countermeasures are discussed. Among them are code hardening techniques, Trusted Platform Module (TPM), ARM's TrustZone, redundant execution and GlobalPlatform's application management and trusted execution environment. We provide explanation and examples of such countermeasures before concluding

the chapter by highlighting the remaining security threats to embedded systems.

As with any manufacturing plants, integrated circuit manufacturers are increasingly moving their foundry to countries with a cheaper cost structure. This means electronic equipment makers have to buy their components from all over the world. Therefore, verifying the integrity of such components before they are assembled is paramount for the security of the final product. The first part of our research, ***Part I: Post-production Pre-deployment Measures***, deals with this challenge. We present our findings of this challenge in chapters 5 and 6.

One of the pre-installed program's attributes that a product manufacturer may be interested in verifying is the program's control flow paths. The easiest way to do that would be for the component maker to supply the source code and pre-computed valid paths. However, due to intellectual property issues this does not happen often. Therefore, in our work, chapter 5,we use side channel leakage of the device as an alternative information. We use the side channel information to reconstruct the control flow path followed by the processor in executing the target program. We then verify it by comparing it with the pre-computed valid paths. This way the verifier does not need to have a prior knowledge of the inner workings of the program and the component maker does not need to handover sensitive company information.

Another attribute of a program that needs verifying is the integrity of executed instructions. For the same reason as above we selected the side channel leakage of the embedded device to verify the instructions. Chapter 6 presents our work in this area. Our procedure involves building side channel template of the instructions and classification algorithm to recognise executed instructions from the device's leakage. Besides verifying the integrity of instructions this method can also help identify changes to the device's original design such as hardware Trojan as it will be reflected on the side channel leakage.

The second part of our research, ***Part II: Runtime Secure Execution***, deals with the challenges of securely executing embedded programs. In this part of our work we proposed techniques that will protect runtime data, control flow and

instructions of a program during execution. Our proposals for runtime secure execution are presented in chapters 7 and 8.

Runtime data is an important attribute of a program security during execution. It is often targeted by an adversary to divert the program's control flow, circumvent countermeasures, extract secret information or simply corrupt the execution result. Such attacks usually involve manipulating items that are pushed/popped into/from the stack. Several countermeasure have been proposed to stop or detect manipulating stack items. In chapter 7, we provide a detail discussion of a stack architecture, operations and countermeasures proposed previously. In addition, we propose an alternative countermeasure that can be used to protect runtime program data on embedded systems. We also provide details of our implementation analysis, computational overhead and detection capability of our proposal.

In chapter 8 we discuss runtime threats to embedded systems control flow and instructions. We provide brief explanation of related works before we discuss our proposals. We then discuss about our control flow verification covering inter- and intra-functional execution flow branches. Furthermore, we propose instruction integrity verification for executed instructions. Our proposals depend on having a dedicated hardware subsystem. It also requires a modified compiler module to extract program attributes during compilation. Finally, we discuss and present implementation results.

Finally, in chapter 9 we conclude the thesis by summarising the contributions and suggesting the future direction of this research. In the appendix section we present additional materials and all source codes of our implementations are available at a public repository [35].

Chapter 2

Embedded Systems Architecture

Contents

2.1 Introduction 30

2.2 Embedded Systems Architectures 32

2.3 Characteristics of Embedded Processors 36

2.4 Application Development Tools 40

2.5 Example Applications 43

2.6 Summary 44

In this chapter, we briefly introduce the main architectural design of embedded systems. This is followed by a discussion on their main characteristics. Then we elaborate on the application development process and tools for embedded systems. Finally, we summarise the chapter by listing the main talking points after a short discussion on the some common embedded system applications.

2.1 Introduction

An embedded system is an applied computer system with a dedicated function within a larger mechanical or electrical system often with real-time constraints. That distinguishes them from Personal Computers (PCs) and supercomputers. However, the definition of "embedded system" is fluid and difficult to pin down, as it constantly evolves due to advances in technology and decreases in the cost of implementing various hardware and software components. In recent years, embedded systems have outgrown many of their traditional descriptions. To better understand them it is important to know what these descriptions are and why they may not be accurate today. The most common descriptions of embedded systems are:

- Embedded systems are more limited in hardware and/or software functionality than a personal computer (PC). This holds true for a significant subset of the embedded systems family of computer systems. In terms of hardware limitations, this can mean limitations in processing performance, power consumption, memory, hardware functionality, and so forth. In software, this typically means limitations relative to a PC including fewer applications, scaled-down applications, no operating system (OS) or a limited OS, or less abstraction-level code. However, this definition is only partially true today as circuit boards and software typically found in PCs are being repackaged into more complex embedded system designs.
- An embedded system is designed to perform a dedicated function. Most embedded devices are primarily designed for one specific function. However, we now see devices such as personal data assistant (PDA)/cell phone hybrids, which are embedded systems designed to do a variety of primary functions. Also, the latest digital TVs include interactive applications that perform a wide variety of general functions unrelated to the "TV" function but just as important, such as e-mail, web browsing, and games.
- An embedded system is a computer system with higher quality and reliability requirements than other types of computer systems. Some families

> of embedded devices have a very high threshold of quality and reliability requirements. For example, if a car's engine controller crashes while driving on a busy freeway or a critical medical device malfunctions during surgery, serious problems may occur. However, there are also embedded devices, such as TVs, games, and cell phones, in which a malfunction is an inconvenience but not usually a life-threatening situation.

Some devices that are called embedded systems, such as PDAs or web pads, are not really embedded systems. There is some discussion as to whether or not computer systems that meet some, but not all of the traditional embedded system definitions are actually embedded systems or something else. Some feel that the designation of these more complex designs, such as PDAs, as embedded systems is driven by non-technical marketing and sales professionals, rather than engineers. Whether or not the traditional embedded definitions should continue to evolve, or a new field of computer systems be designated to include these more complex systems will ultimately be determined by the industries and research organisations. For now, since there is no new industry-supported field of computer systems designated for designs that fall in between the traditional embedded system and the general-purpose PC systems, this thesis supports the traditional definition of embedded systems. That is a resource constrained integrated circuit designed to perform a limited functionality (usually for a dedicated task) with limited power supply under a harsh environment.

In Section 2.2 we discuss the generic architecture of an embedded processor and its different components. Following in Section 2.3 we elaborate on embedded system characteristics. We explore the design parameters and their classification. In Section 2.4 we explain the process and tools required to develop a program for an embedded device. We continue in, Section 2.5, with a discussion on some common embedded system application areas. Finally in Section 2.6 we summarise the main points of the chapter.

2.2 Embedded Systems Architectures

At the centre of an embedded system there is always a processor that controls its activities. This processor has multiple modules, including the Microprocessor Unit (MPU), data and program memory, Input/Output module, bus and other supporting modules depending on the application of the embedded system [36]. Fig 2.1 depicts the general architecture of a typical embedded processor.

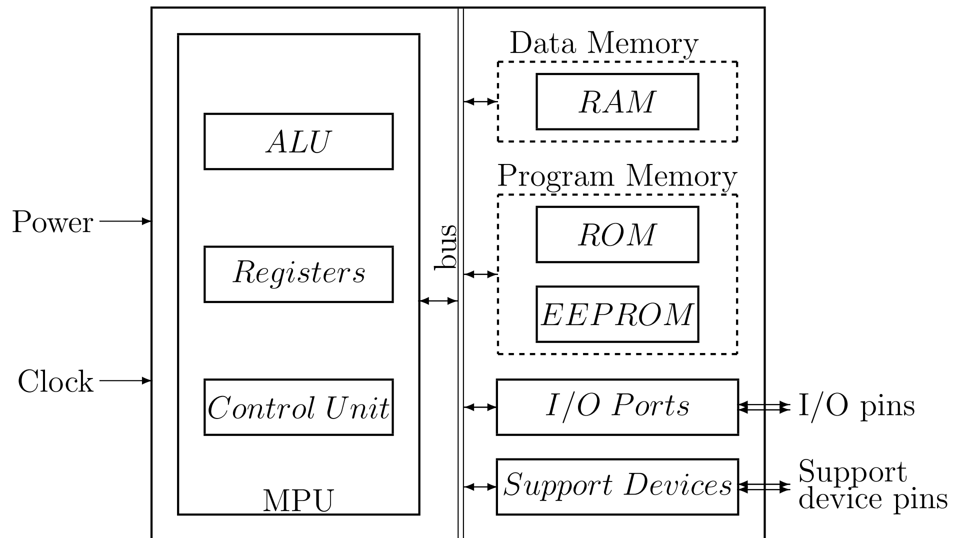


Figure 2.1: General architecture of embedded processor

The processor is powered up by supplying a voltage source through the power pin. The clock is a timing signal which is used to synchronise the processor's various components while they execute arithmetic and logic operations or transfer data between different locations. The clock signal can be generated either inside the processor's integrated circuit or obtained externally via the clock input pin. We now discuss the different components of an embedded processor.

2.2.1 Microprocessor Unit

The *Microprocessor Unit* (MPU) is the main module of an embedded processor. It is responsible for the execution of embedded programs that are installed inside the processor. It performs all logic and arithmetic operations and also controls the

processor's status. To help it accomplish these activities it has three components: the *Arithmetic Logic Unit* (ALU), *Registers* and the *Control Unit* (CU).

The ALU carries the arithmetic and logic operations required on the operands [1] in the program's instructions. In some processors, the ALU is divided into Arithmetic Unit (AU) and Logic Unit (LU). In such configuration the AU performs the arithmetic operations and the LU deals with the logic operations. In addition, some processors contain multiple AUs, for instance one for fixed-point [2] and another for a floating-point [3] operations. Normally, an ALU has a direct input and output access to the control unit, Random-Access Memory (RAM) (described in detail in Section 2.2.2), and input/output modules (described in detail in Section 2.2.3).

Registers are memory cells that are located inside the MPU. They are used for quick reading, storing and manipulating of data. However, there are also designated registers for special purposes. Some of the special registers that are paramount for executing programs are; the Instruction Register (IR), Memory Data Register (MDR), Memory Address Register (MAR), Program Counter (PC), Accumulator and Status Register (SR). The IR hold the instruction that is currently being executed. The MDR (also known as memory buffer register or data buffer register) holds data that is fetched from the memory. The MAR carries the address of a memory cell that needs to be fetched next. The PC holds the address of the next instruction that needs to be executed. Normally, its value is sequentially incremented unless an instruction is executed that changes it (such as function call instructions and conditional or unconditional branching instructions). The accumulator is a register which holds the intermediate result of an arithmetic or logic operations before the final result is moved to the destination memory location or another register. The status register, is used to save or indicate the status of the MPU. It has a number of bits that can be used to flag exceptions, indicate the arithmetic status of the ALU and the reset status of the MPU.

[1] A data on which the operation (specified by the opcode) is to be performed.

[2] A fixed-point is the representation of a number with fixed number of digits after the decimal point.

[3] A floating-point is a number with no fixed number of digits before and after the decimal point. In other words, the decimal point can float.

The final component of the MPU is the Control Unit, which provides timing and control signals for all other components of the embedded processor. Basically the control unit tells the embedded device's memory, ALU, I/O modules and other components how to respond to each instruction executed by the processor.

2.2.2 Memory

In general there are two memory types in embedded processors: *volatile* and *non-volatile*. A volatile memory is a storage where all previous data are generally lost within at most a second or so when the power is turned off. Such storage is also known as a Random Access Memory (RAM), mainly because any location within it can be accessed directly and randomly, rather than sequentially from some starting point, at approximately the same time and its contents can be changed multiple times. On the other hand, non-volatile memory retains its data even when the power source is removed. Examples of such memory are Read-Only Memory (ROM) and Electrically Erasable Programmable Read-Only Memory (EEPROM). Information can only be written into ROM once and cannot be modified again. However, contents of an EEPROM can be erased and reprogrammed repeatedly by applying a higher electrical voltage through the provided program pin of the processor.

In Fig. 2.1, these memory types are classified under two categories; the `Data Memory` and the `Program Memory`. The `Program Memory` is where the application/program is stored. On the other hand, `Data Memory` is where the processor stores data and variables during runtime. Runtime variables are created during and deleted at the end of the program execution and are normally stored in RAM. However, actual program instructions are required to persist even after the power is removed and therefore are usually stored in ROM and EEPROM. Different processors have different configuration of the `Data Memory` and `Program Memory`. For instance the `Data Memory` and `Program Memory` of Harvard processors are organised into two separate blocks: the *program memory* and the *data memory* [37]. They are on physically separate buses so that instructions cannot be used as data or *vice versa*. In von Neumann architecture processors the

MPU uses a single main memory to store both program instructions and data, and are accessed by through a single bus. Details of Harvard and von Neumann architectures are discussed in Section 2.3. Modern high performance microcontrollers incorporate aspects of both Harvard and von Neumann architectures. The on-chip cache memory is divided into an *instruction cache* and a *data cache* which store copies of values that are used frequently by the MPU. The MPU uses Harvard architecture when accessing the cache.

2.2.3 Input/Output

An input/output is a communication between an MPU and the outside world, possibly human beings or another embedded device. The input/output device (module) assists the MPU in achieving the above task. The input/output data can be can be analogue signal or binary information. The input is the signal or data received by MPU from the outside world and the input device can be, for instance, a keyboard, mouse or other sensors. The output is signal or data sent by the MPU to the outside world and the output device can be a printer, display monitor or a server.

2.2.4 Bus

The bus is a communication channel that data between different components inside the processor is transferred through. Generally an embedded processor has three different buses; address bus, data bus and control bus. The address bus connects the MPU with the memory and carries the address of a location in the memory. This address is then used to identify a specific location and before performing an operation on it. A data bus is the pathway through which data is transferred from MPU to memory and vice versa. The control bus carries a control information between the MPU and other components of the embedded processor. This control information carries control signals that indicate the status of the various components.

2.2.5 Supporting Devices

Apart from the various modules discussed above an embedded system may require additional modules to successfully accomplish its tasks. For instance, some of the most commonly integrated modules into embedded processors are Digital-to-Analogue Converter (DAC) [38], Analogue-to-Digital Converter (ADC) [39], serial and parallel ports, timer, etc. The DAC and ADC convert digital signal to analogue and vice versa respectively. This helps the processor to interface with the external world and other electronic equipments more easily. The serial and parallel ports enable the processor to send and receive data either serially or in parallel. A timer allows the processor to measure the precise execution of selected tasks. Sometimes a processor may also have subsidiary MPUs, called `co-processors`, which generally help the main processor with heavy arithmetic and logic operations such as those used in cryptographic operations [40].

2.3 Characteristics of Embedded Processors

The heterogeneity of embedded systems provide engineers with a new set of challenges. As in many areas in engineering, the design of embedded systems is particularly driven by cost/benefit trade-offs. Furthermore, reliability, size and time-to-market provide additional design obstacles to engineers. This can make traditional computer system design methodologies difficult to successfully apply in embedded systems. In the following sections we discuss the main design challenges of embedded systems and their classification criteria.

2.3.1 Design Parameters

The uniqueness of embedded application systems makes generalisation very difficult. Nonetheless, there is a growing interest in the area of embedded systems. The most common design challenges are size, reliability, durability and cost effectiveness [41, 42]. These challenges are discussed briefly below.

Small Size and Light Weight: Like the name implies, embedded systems are physically located inside larger electronic equipment. The size and weight of an embedded system is determined by its application area, power consumption and portability requirement. For example, smart cards are designed to be light weight and fit into a pocket. Therefore, smart card processors are designed to meet those requirements.

Safe and Reliable: Embedded systems have been used in tasks from simple temperature sensing to assisting the functionalities of human organs [43]. Although mission critical embedded systems design raise the obvious reliability concerns, unexpected or premature malfunctionality of embedded systems in applications like game boxes may result in eroding manufacturer's reputation.

Withstand Harsh Environment: Many embedded systems are designed to operate in uncontrolled environment. One of the main problem is excessive heat especially in areas that involve combustion such as missile systems and many transport systems. However, embedded systems are also designed to withstand other harsh environments, like vibration, shock, water, corrosion, fire and other physical abuses.

Cost Sensitivity: Although system designers of all types of electronic devices or integrated circuits talk the importance of cost effectiveness, the sensitivity changes dramatically when it comes to embedded devices. One of the reasons is, embedded devices are deployed in mass and for a specific purpose. For instance, low-end embedded systems used to monitor environmental changes in agricultural farms [44].

2.3.2 Classification

Embedded systems can be categorised into different classes based on their attributes. These attributes include the manufacturer company, architectural de-

sign, processing size, memory layout and instruction set. Fig. 2.2 illustrates the classification of embedded systems based on these attributes.

One category of embedded devices is based on the processing size of the embedded processor. The size refers to the number of bits that the processor can handle at a given time. In modern embedded systems the lowest number of bits that the processor handles is 8 bits and the highest can go up to 32/64 bits. All embedded systems have a memory where they save the program instructions and other data. If the memory module is integrated inside a single chip with the other processor modules it is known as *Embedded Memory* otherwise *External Memory* embedded system. Every embedded system have a certain set of instructions that can be used to program the processor to perform desired operation. Embedded systems with a *Reduced Instruction Set Computer* (RISC) [45] architecture use simple, single clock cycle instructions. However, for specific tasks, the number of instructions per application can be reduced by having multiple operations within a single instruction which lasts for several clock cycles. This gives a *Complex Instruction Set Computer* (CISC) [46] architecture. For example, a CISC processor needs only one instruction to multiply two memory contents whereas a RISC processor needs four to perform the same task. Details of this example is presented in Table 2.1.

CISC	RISC	Pseudo-Code
mul mem-loc1, mem-loc2	**load** a, mem-loc1	a ← mem-loc1;
	load b, mem-loc2	b ← mem-loc2;
	mul a, b	a ← a*b;
	store mem-loc1, a	mem-loc1 ← a;

Table 2.1: CISC, RISC and equivalent Pseudo- Code for Multiplying two Memory Contents

Apart from the CISC and RISC architectures, an embedded system is said to have a *Application Specific Instruction Set Computer* (ASIC) [47] architecture if its processing unit and instruction set are customised to do a specific type of job. Another classification criteria is the general architecture of the embedded processor. The two most common embedded architectures are the *von Neumann* and *Harvard* architectures. Those based on the von Neumann architecture [48, 49] have a single data bus for fetching program instructions and program data. Both

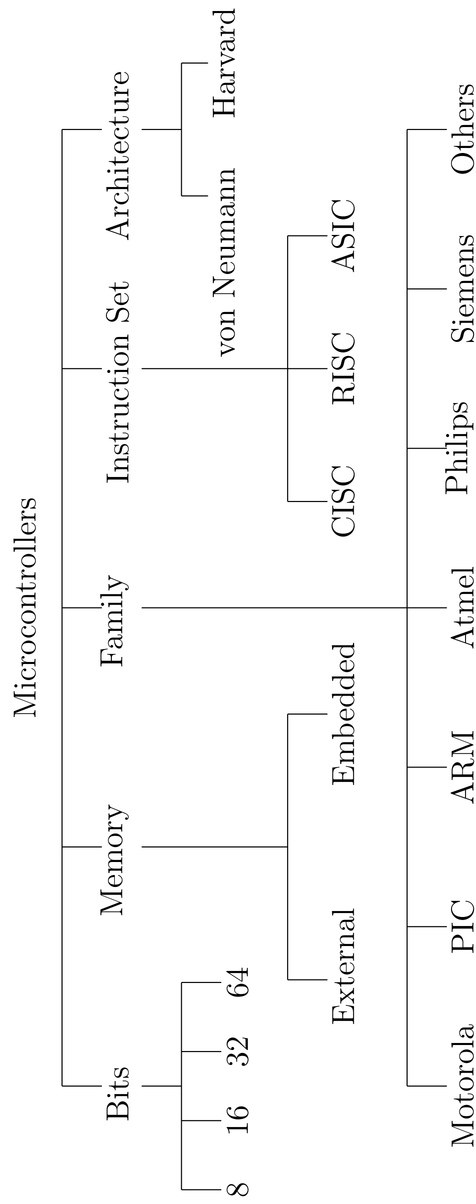


Figure 2.2: Classification criteria of microcontrollers

the program instructions and data are stored in a common main memory. When the processor has to perform a task, it fetches the instruction first and then the data associated with it. Harvard architecture processors use separate buses to access the program's instructions and data. Such a configuration allows parallel memory access to occur. The processor can fetch both the next instruction and its data simultaneously while executing the current instruction. This generally leads to improved performance. The final category criteria is the manufacturer company. In this case the processors are simply known by the name of the company that manufactured them.

2.4 Application Development Tools

Developing applications for embedded systems involve the process of writing the program code, compiling, debugging, simulating and finally loading the program onto the target processor. This requires a set of software and hardware tools. These tools can be standalone software programs or integrated into one development software also known as Integrated Development Environment (IDE).

2.4.1 Source Code Editor

A source code editor is a software program specifically designed to simplify and speed up code writing and editing for computing systems. Some of the main features of a source code editor are indentation, syntax highlighting, key words auto-complete and bracket matching. Such software can be a stand alone, like gedit [50], an integrated module of a bigger system, such as MPLAB IDE [33], or web based editor, like [51]. Code editors also support multiple programming languages. For instance source code editors for embedded systems usually support C, C++ and assembly programming languages. Below, Figure 2.3, is a picture of gedit and MPLAB IDE. The gedit is a general purpose code editor whereas MPLAB IDE is specifically designed for PIC microcontroller families.

(a) Gedit.

(b) MPLAB IDE.

Figure 2.3: Example of source code editor software.

2.4.2 Compiler

A compiler is a software program that translates a source code (usually written in high-level programming language like C, C++, etc) into an object code that is understandable by the underlying hardware of the target device [52]. A compiler performs all or most of the following tasks; code parsing, pre-processing, semantic analysis, code generation and code optimisation. GCC is one of the most accomplished open source compilers that supports multiple programming languages [53].

2.4.3 Debugger

A debugger is a computer program that assists the detection and correction of program errors in other computing programs. A debugger uses Instruction Set Simulator (ISS) to mimic the target processor's behaviour to execute the program under test. To speed up the debugging process, debuggers offer two modes of operation; full or partial simulation. Full and partial simulation refers to simulating all or selected components of the device respectively. Simulation is discussed in detail in Section 2.4.4. During evaluation the debugger stops when it encounters a program bug or invalid data and notifies the programmer about it by showing the location of the error on the original source code.

2.4.4 Simulator

As defined in the Oxford Dictionary [54], "simulator" is a computer program "that enables a computer system to execute a program written for a different device/system". An embedded system simulator is a computer program which imitates the characteristics and functionalities of an embedded device. During application testing the program code is loaded into the simulator instead of the physical device memory, and executed. This process is helpful during prototype verification as it minimises the time needed to load the code into a real device during testing. An example of such a software, PIC simulator IDE, is shown in Fig. 2.4. A PIC simulator IDE simulates the memory, processor, input/output and other functionalities of a range of PIC microcontroller [55].

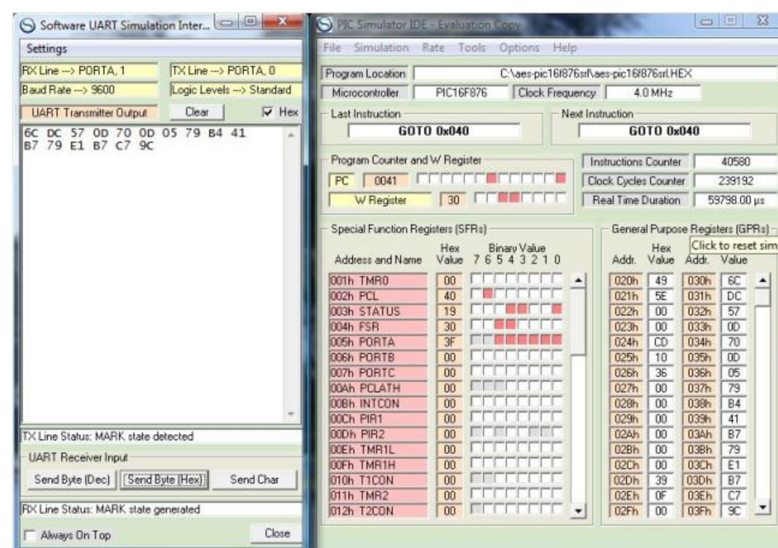


Figure 2.4: PIC simulator IDE, a software designed to simulate a range of PIC microcontrollers.

2.4.5 Device Programmer

A device programmer refers to a device that configures the non-volatile memory of the target processor. It takes the output file of the compiler and writes it to the device's internal memory. Normally, this is the last phase of the embedded program development process. There are four types of device programmers; Gang

programmers for mass production, development programmers for testing, pocket programmers and specialised programmers for certain types of circuits only (such as EEPROM). Example of a device programmer is the universal device programmer [56] used on PIC microcontrollers.

2.5 Example Applications

As discuss earlier, embedded systems have become integral part of our modern life style. In Table 2.2, we list some of the common embedded systems applications that we use on a daily basis.

Home Applications	Dishwasher, Washing Machine, Microwave Oven, Set-Top-Box, DVD player, Answering Machine, Home Security Systems, Lighting system, Remote Controller, Air Conditioner, Sprinklers.
Business Equipment	ATM, Alarm Systems, Card Readers, Fingerprint Detectors, Automatic Toll Systems, Voice Recognisers, Vending Machine, Barcode Reader.
Communication Systems	Router, Hub, Cell Phone, Web Camera, Modem, Network Cards, Tele-conferencing System.
Aerospace	GPS, Automatic Landing System, Space Robotics Flight Control Inertial Guidance System, RADAR.
Industrial applications	Smart Phone, Fax Machines, Photo Copiers, Printers, Scanners, Data Collection System, Voltage, Current and Temperature monitoring, Hazard Detecting System, Industrial Robot.
Automobile	Fuel Controller, Brake System, Cruise Control, Transmission Controller, Active Suspension, Air-bag System, Air-Conditioner.
Game and Entertainment	Video games, Robot, MP3, Mind Storm, Smart Toy.
Education	Smart Board, Smart Room, OCR, Calculator,

	Smart Cord, Stereo Systems, Projector.
Security Systems	Face Recognition System, Finger Recognition, Iris Recognition, Building Security System, Airport Security System, Alarm System, Digital Access Card, Fingerprint based Smart Card.
Consumer Electronic Products	Cell phones, Cordless Phones, Digital Cameras, Video recorders, DVD players, TV set, Calculators, MP3 Players, Stereo Systems, Cable TV tuners, Digital watches, Personal PDA, iPhone.
Medical Technology	CT scanner, ECG, EEG, EMG, MRI, Glucose Monitor, Blood Pressure Monitor, Diagnostic Device, X-ray machines, Digital Pulse Monitor.

Table 2.2: A list of embedded system applications.

2.6 Summary

Embedded systems are becoming mandatory tools to perform our day to day activities. As such providing a brief discussion about them is important to understand how embedded systems work and to establish a background information for the subsequent chapters. Therefore, we started this chapter by defining embedded systems and how this definition has changed over time. We discussed the generic architecture and all the modules that are incorporated for an embedded device to properly function. We then proceeded to discuss the main design parameters of embedded systems and their classification criteria. Finally we provided a list of embedded system applications that we use on a daily basis.

Chapter 3

Attacks on Embedded Systems

Contents

3.1 Introduction 46

3.2 Invasive Attacks 47

3.3 Semi-Invasive Attacks 52

3.4 Non-Invasive Attacks 54

3.5 Summary 57

In this chapter, we review the different attack categories on embedded systems. We then discuss on the three widely used attack categories; invasive, semi-invasive and non-invasive attacks. Under each of these categories we discuss the stages involved and their applicability in detail. Finally, we summarise the chapter by listing the core discussion points of the chapter.

3.1 Introduction

Embedded systems are widely used in applications where conventional workstations or server computers are not suitable due to their functionality, cost, power requirements, size and weight. Such application areas may range from a simple room temperature controller to highly sophisticated systems like bank cards and commercial and military communication equipments. However, the very same characteristics that make embedded systems ideally suited for such applications also lead to a set of potential vulnerabilities.

Some of the main characteristics of embedded systems are limited processing power, limited power consumption and their ability to be deployed outside the immediate control of the owner and sometimes even in hostile areas. Limited processing power means resources of embedded systems can only be dedicated to perform certain tasks. This makes them vulnerable to certain threats that are not applicable or at least preventable on conventional computers. Limited processing power implies that an embedded system can not run applications that are used for defence against attacks in conventional computers, for example virus scanner and intrusion detection system. The majority of embedded systems operate on batteries or limited power supply, therefore, increased power consumption means reduced lifetime. Hence embedded systems can only run low power consuming dedicated tasks with limited attack countermeasures. Furthermore, their deployability outside the immediate control of the owner or sometimes in hostile territory[1] makes them inherently vulnerable to attacks that exploit the physical proximity of the attacker. In this scenario a user can also be the attacker.

In recent years several attacks that exploit the above characteristics and others not mentioned here have been explored in detail. These attacks are discussed in subsequent sections of this chapter. The abuse of such attacks may range from aiming at stealing secret information, draining the power supply, reverse engineering its applications to gain unauthorised access. These security threats to embedded systems are categorised by the intrusiveness of the attack. Some

[1] Outside the control of the manufacturer or issuer, where the device could be subjected to physical and/or logical attacks.

attacks destroy the embedded system's structure irreversibly while others observe its properties through available output channels. In general they all fall into one of the following categories; invasive, semi-invasive and non-invasive attacks.

Section 3.2 discusses invasive attacks and the different stages and techniques that they involve. Section 3.3 provides a brief explanation of semi-invasive attacks and some of the most common examples of semi-invasive attacks. Section 3.4 discusses the final class of attacks, non-invasive, and provides a brief explanation of the most common attacks that fall under this class. Finally, we summarise the chapter in Section 3.5 by mentioning the core points.

3.2 Invasive Attacks

Invasive attacks are attacks that require the processor in an embedded system to be exposed and directly attacked through physical means [57]. This category of attacks, at least in theory, can compromise the security of any secure processor chip. However, these attacks require a set of very expensive equipment and large investment in time to produce results. These equipments are discussed in the subsequent subsections. Invasive attacks modifies some of the physical properties of the processor irreversibly [58]. Successful invasive attacks may involve performing a number of different tasks precisely at different stages of the attack [59]. The result of such attacks could be revealing secret information kept inside the processor, modifying the original circuit design and/or reverse engineering the semiconductor chip itself [60]. Some of the common invasive attack methods are discussed below in detail.

3.2.1 Delayering

An IC chip is made of multiple layers of metal and silicon oxide. An IC chip is covered by a global top layer of epoxy resin. The layer below is made of silicon oxide which protects the chip from environmental hazards and ionic contamina-

tions. Delayering is the process of accurately stripping off each layer, one at a time, while keeping the surface flat before gaining access to the underlying chip circuit. This requires detailed removal techniques. The techniques include a combination of dry and wet etching, and polishing. For instance, the epoxy resin, is normally removed using a fuming nitric acid [61]. However, before applying such removal techniques, several scanning methods, such as Scanning Electron Microscopes (SEM) [62] and Transmission Electron Microscopes (TEM) [63], are used to determine the composition and thickness of each layer [64].

3.2.2 Block Localisation

A semi-conductor chip[2] is comprised of multiple components. During *Block Localisation* the attacker uses a decapsulated chip to locate these different components/blocks of the chip. An IC chip block includes the RAM, EEPROM, ROM, Chip logic and BUS. Depending on the nature of chip it may have other components like cryptographic processor and random number generator. Figure 3.1 shows the different components of a decapsulated Infineon processor.

ROM The Read-only Memory (ROM) is a critical part of an IC chip. It is a type of non-volatile memory where the immutable code of the embedded system, such firmware that does not require changes, is stored. The contents of ROM can not be modified.

RAM The Random Access Memory (RAM) is one of the main components of an embedded device. It is a temporary working storage used during application execution. Program data and instructions fetched from permanent memory are stored in RAM.

EEPROM EEPROM which stands for Electrically Erasable Programmable Read-Only Memory is a non-volatile memory used to store data that must be

[2] An integrated circuit on a semiconductor material, such as silicon.

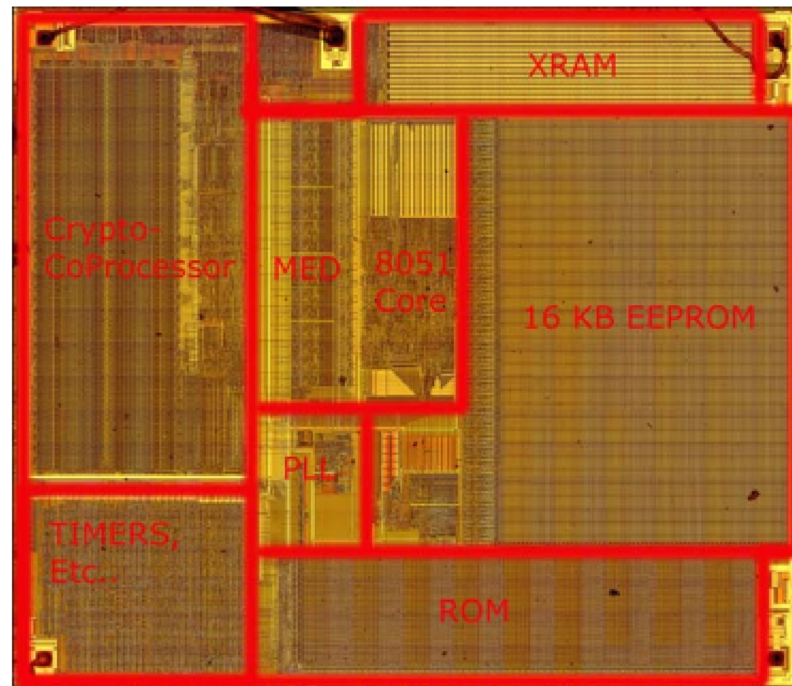


Figure 3.1: Different blocks of a decapsulated Infineon SLE66 [1].

kept after power removal. Unlike ROM individual bytes of EEPROM can be read, erased and re-written.

BUS A BUS is a communication system that transfers data between the different components inside an embedded device. Early bus were parallel electrical wires with multiple connections but the term is now used for any physical arrangement that provide the same logical functionality.

Chip Logic This refers to the core circuit of the embedded processor. This could be either a fixed logic circuit that performs the same operation every time or a programmable general purpose circuit.

Other components Apart from the components listed above an embedded system could also have other modules. Such modules may include cryptographic processor engine, a random number generator or any other component that helps

the embedded system perform its task [65].

3.2.3 Microprobing

As depicted in Figure 2.1, an embedded device has multiple components and these components are linked together through an internal wiring, also known as the BUS. Information flows between these components via the BUS. Probing is the process of attaching microscopic needles onto this internal wiring of a decapsulated chip. Once the probes are attached the attacker can read out information that was not intended to leave the chip. Figure 3.2 shows probing needles attached to the internal wirings of a chip with its layers removed. Microprobing attack require tools like microscope, micropositioners, probing needles and amplifiers [66, 67].

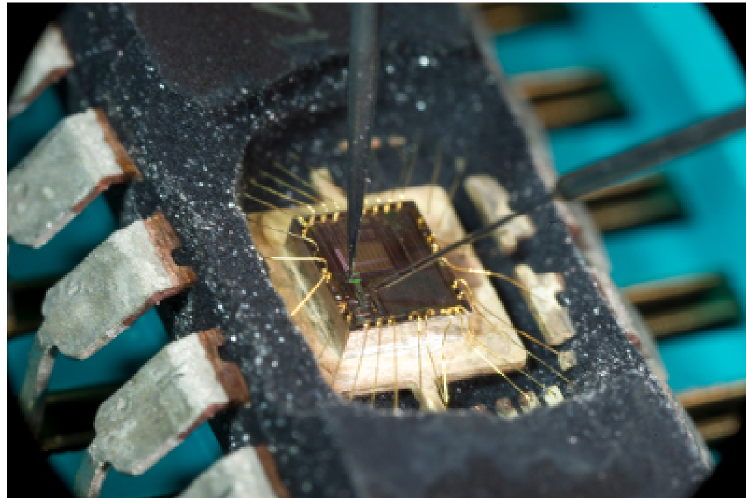


Figure 3.2: Probing needles on a decapsulated semiconductor chip [2].

3.2.4 Circuit Extraction

Following a successful removal of each layer, the attacker analyses the decapsulated chip to understand the structure, functions and circuit of a semiconductor device. This requires photographing the chip and detailed analysis of the pictures

manually or with the help of tools. Figure 3.3 shows a photograph of a decapsulated chip being analysed on a computer, identifying all the transistors, coils, resistors, capacitors, conductors and their interconnection. The result of such analysis may include a circuit diagram and circuit simulation. A standard netlist file may also be created using the information gathered. The resulting netlist[3] will then be used to create an identical device to the original one. A practical example of circuit extraction is described in [68].

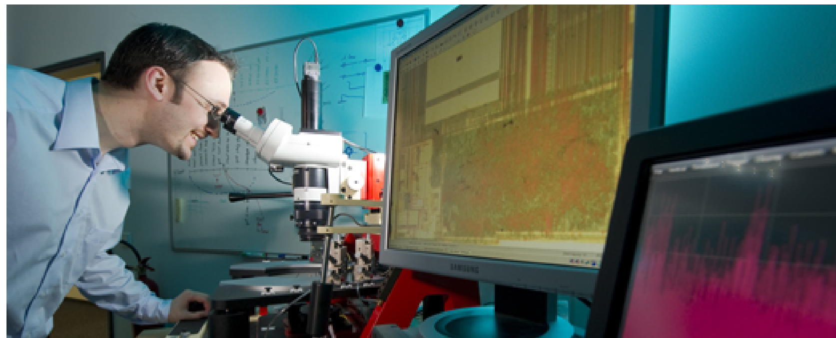


Figure 3.3: Taking microscopic pictures of the inner circuitry of a semiconductor chip [2].

3.2.5 Circuit Modification

At its extreme form invasive attacks can be used to destroy or create new tracks by using focused ion beams [60]. For example, traditional semiconductor chip manufacturers typically use a test circuit to read and write the entire memory space while a fuse was present. When the chip leaves the manufacturing process the fuse is destroyed. However, an attacker may use a focused ion beam to re-introduce the fuse and therefore, have access to the entire memory address of the chip.

Another classic example of chip modification attack is reconfiguring the output of security components. For instance, random number generators are usually part of modern secure embedded devices. On a decapsulated chip an attacker may

[3]A netlist is the description of a circuit connectivity in integrated circuit.

reconfigure the output tracks[4] of random number generators to force an all '0', all '1' or a static value.

3.3 Semi-Invasive Attacks

Semi-Invasive attacks are attacks that require the external surface of the chip to be exposed to the attacker. At the most extreme they require the decapsulation of the target device. As such they lay between invasive and non-invasive attacks in terms of the change that they cause to the device's integral structure. They are less damaging to the target device when compared to invasive attacks but more intrusive compared to non-invasive attacks. Usually these type of attacks use external equipment to induce errors to the contents of the embedded system, then later exploit these errors to deduce secret information stored inside the target device. In the subsequent sections we discuss some of the most common semi-invasive attacks.

3.3.1 Local Heating

Very often, sensitive information such as cryptographic keys and passwords, are stored in non-volatile memory such as EEPROM and flash[5] of secure micro-controllers. Inducing memory errors could enable an attacker to deduce this information. Such attacks do not require expensive equipments and do not cause mechanical damage of the silicon structure. One practical implementation of such attacks is using local heating. The main principle behind local heating is that by focusing a strong enough heat generator, like laser radiation, on a small area inside a semiconductor chip an attacker can modify its contents which eventually will lead to a leakage of sensitive information. In [69] a local heating attack, using laser radiation, on a common microcontroller Microchip PIC16F628 [70] has been

[4] Conductor, normally copper, lines connecting the random number generator output pins with the other components of the device.

[5] A non-volatile memory storage that can be erased and reprogrammed electrically.

demonstrated. In this paper both the EEPROM and Flash memories were found to be sensitive to local heating.

3.3.2 Ultraviolet Attacks

Ultraviolet attacks are among the first attacks used on microcontrollers and they were introduced in the mid seventies [71]. A UV attack involves two stages; locating the area of interest (like a fuse) and resetting its value. In another word, the attacker changes the value of a selected memory location by using a UV light. Initially, UV attacks were considered invasive attacks. However, as they only require decapsulation of the semiconductor chip, at the most of their intrusiveness, they were later categorised as semi-invasive attack.

3.3.3 Optical Emission Analysis

The existence of photon emission regarding the switching of transistors in semiconductor device is a well known. In fact optical emission analysis of semiconductor devices have been widely used in various device malfunction analysis techniques to detect faults in chip circuitry [72]. Apart from device malfunction analysis, optical emission analysis has been used to attack implementation of AES on embedded systems [73]. However, there are two main problems associated with optical emission analysis. These are; *a*) not every transistor switching produces emission of photons. That means the photon emission of the circuit must be collected and added for some time. *b*) the electrical current of the photon emission sensor adds noise in the measurement and increases the time required to achieve a reasonable signal-to-noise ratio of the measurement.

3.4 Non-Invasive Attacks

In a non-invasive attack, the attacker attacks the embedded device only using directly available interfaces without permanently damaging/altering the device. Such attacks can be particularly dangerous for two main reasons. Firstly, the owner of the compromised device might not notice that the device has been attacked. For example it is unlikely that compromised keys will be revoked unless they are abused by the attacker as the user will simply not notice it. Secondly, non-invasive attacks often scale well, as the required attack equipment can usually be reproduced and updated at low cost.

There exist two main types of attacks; passive non-invasive and active non-invasive attacks. Passive non-invasive attacks only observe and exploit the device's properties while it performs certain tasks. This type of attack often referred as side channel attack (discussed in Section 3.4.1 in detail) and the property exploited is often the power consumption and/or electromagnetic emission. Active non-invasive attack temporarily injects a fault into the device's operations without permanently damaging it (discussed in Section 3.4.3 in detail). Below we discuss the three main non-invasive attacks in more detail.

3.4.1 Side Channel Analysis

A side channel is information leaked by an electronic devices while performing certain procedures. Electronic equipment, such as embedded devices, use electric current to turn on or off transistors. The instantaneous electric current that the device consumes depends on how many transistors that the executed instructions and program data turn on and off. This difference in the electric current is reflected in the power consumption and electromagnetic emission of the device. The power consumption and/or electromagnetic emission can then be recorded and analysed to extract secret information from the target device.

Over the years multiple forms of side channel analysis attacks have been proposed. If only a single trace is enough to mount a side channel attack it is known as

Simple Power Analysis (SPA) or single electromagnetic analysis [74]. The SPA is the visual inspection and interpretation of the target device's side channel trace. However, the dependence is quite small or obscured by noise. This can be compensated by collecting multiple traces and subjecting them to a statistical analysis. This attack is known as Differential Power Analysis (DPA) [75] and differential electromagnetic analysis [76]. The DPA uses more advanced statistical tools to analyse a number of power traces, usually 1000s, collected from the target device. An important aspect in these attacks is the traces must be aligned, that is they must be combined in time-domain with corresponding steps coinciding between all traces. Another type of side channel analysis attack is the Template Power Analysis (TPA). In TPA the attacker builds a side channel template model using an identical device and then use the template to attack the target device while in operation [77]. A TPA can be used to reveal secret information or full/partial reverse engineering of embedded program [78].

In the context of cryptology, side channel leakage can be used in retrieving secret keys that were kept inside the embedded devices, such as smart cards and other security tokens. Side channel information such power consumption [79, 77] and electromagnetic emission [80, 81, 82] have been successfully used in attacking implementations of cryptographic algorithms including AES [83], DES [84] and RSA [85]. Besides extracting cryptographic keys, side channel information has also been used to reverse engineer embedded device applications [86, 32, 87]. The attacker constructs a power consumption template of the target device using an identical reference device, then use the templates to recognise executed instructions from the target device's power consumption waveform.

Another well known side channel attack is timing attack. In this class of attack the attacker attempts to compromise a cryptosystem by collecting and analysing the time taken to execute a cryptographic algorithm. Every logical operation in a computer takes time to execute, and the time can differ based on the input. With a precise measurement of timing an attacker can deduce the cryptographic key involved in the operation [88, 89, 90].

3.4.2 Data Remanence

Data remanence is the residual physical representation of data that has been erased or overwritten. In non-volatile programmable device, such as EEPROM and flash, bits are stored as charge in floating gates of a transistor. After each erase operation some of the charge still remains. Residual data after erasure was first found in magnetic media but then appeared to be the case for other memory types [91, 92]. This can lead into extraction of the saved data even after it is erased. Low temperature data remanence is dangerous to tamper resistant security modules which store keys and secret data in a battery backed-up SRAM. Long time data storage causes the data to be "burned-in" and likely to appear after power up. This makes security devices vulnerable to such attacks. Furthermore, lowering the temperature increases the retention time of information after erasure. In an experiment that involved 8 selected SRAMs the data retention was found to be varying from 0.1 to 10 sec at room temperature, 1 to 1000 sec at $-20\,°C$ and 10 sec to 10 hours at $-50\,°C$ [93].

3.4.3 Fault Injection

Fault injection is the deliberate introduction of faults into a system and the subsequent examination of the system for the errors and failures that result [94]. Initially fault injection were used as a hardware and software testing techniques. It was performed on either simulations and models or working prototype. In this manner the weakness of the system can be discovered and fixed before the final design is released.

However, the same technique can also be used in attacking embedded devices either to extract secret information which were kept inside the device or gain unauthorised access. The first use of a fault injection attack to extract a secret key from a cryptographic algorithm was presented in a work by the Bellcore research team in [95]. In their work they showed how a single fault could be used to break Chinese Remainder Theorem (CRT) based RSA implementation. Later, Biham and Shamir introduced the concept of Differential Fault Analysis

(DFA) [96] on the Data Encryption Standard (DES) [84]. Subsequently, the same concept led to different attacks [97, 98, 99, 100, 101] on the Advanced Encryption Standard (AES) [83]. Additional information on fault attacks on cryptographic algorithms is available in [102, 103, 104].

3.5 Summary

Embedded systems are increasingly deployed in a security critical environments. As a result attacks on them are on the rise. Before proceeding to our work a brief discussion on these attacks was needed to understand the security threats of embedded systems. In this chapter, we reviewed the security threats that particularly target embedded systems. We started the chapter by briefly explaining the main criteria of attacks on embedded systems. We then proceed to the different classes of attacks, invasive, semi-invasive and non-invasive attacks, and the characteristics for such classification. We then explored invasive attacks in detail and the different stages that an attacker is required to perform. We talked about IC chip delayering, locating the different components, micro-probing, chip extraction and modification techniques. Following this we elaborated semi-invasive attacks. Under this section we covered some of the most common examples of this class of attacks; local heating, UV attacks and optical emission analysis. This was followed by a discussion on the third and final category of attacks; non-invasive attacks. In this section we deliberated about side channel, data remanence and fault injection attacks.

Chapter 4

Security In Embedded Systems

Contents

4.1 Introduction **59**

4.2 Code Hardening **60**

4.3 Side Channel Protection **62**

4.4 Tamper Resistance **62**

4.5 Redundant Execution **64**

4.6 Trusted Platform Module (TPM) **65**

4.7 ARM's TrustZone **66**

4.8 GlobalPlatform **67**

4.9 Remaining Security Challenges **69**

4.10 Summary **72**

In this chapter, we discuss the common security countermeasures used to secure embedded systems. First, we deliberate on the different security techniques that are used by designers and application developers. Then we explain the remaining security challenges that need to be met, both at pre-deployment and runtime. Finally, we finish the discussion by pointing out the core points.

4.1 Introduction

As discussed in chapter 3, attacks against embedded systems have increased in number and sophistication. In many of the modern embedded system applications security is indispensible. Examples of such applications are transport (avionics, space, automotive, trains), missile control, smartcards and other factory automation systems.

To understand the necessity of security in embedded system let's discuss some of their common applications. For instance, smartcards are issued to individual customers by organisations so customers can access the organisation's services in a secure and reliable manner. Areas where smartcards are deployed include telecommunications, banks and access control. The failure to securely access these services securily may result in monetary, reputation and even physical harm. Another example is the avionics and automative industry. The Electronic Control Unit (ECU) plays a crucial role in ensuring the safety and reliability of cars. These ECUs are embedded devices that control different operations in a car. In a modern car upto 70 ECUs [105], are used and if any of them can be compromised the safety of the car and passangers may be at risk.

To ensure the safety and reliability such systems from attacks, several countermeasures have been proposed and implemented. These countermeasures range from program source code modification, tamper-resistance sensor shields, deploying a dedicated security chip to using application specific processors.

In section 4.2, we discuss a source code hardening technique that introduces certain redundancy to the source code so attacks in progress may be detected. In section 4.3 we explain protections against side channel attacks. Following, in section 4.4 we describe tamper resistant methods against physical attacks. A hardware and time redundant execution against fault injection attacks is discussed in section 4.5. Then we deliberate on Trusted Platform Module (TPM) in section 4.6 and ARM's TrustZone in section 4.7. In sections 4.8.1 and 4.8.2 we explain GlobalPlatform's application management and trusted execution environment respectively. Then in section 4.9 we point out the remaining security

challenges and in section 4.10 we conclude the chapter by summarising the main discussion points.

4.2 Code Hardening

A program code is a group of executable processor instructions designed to achieve the desirable output. During execution of the program each instruction performs a certain operation. Now these instructions can be individually targeted by an attacker in order to force the processor into generating a faulty output. An example of such attack is the fault injection attacks, where the attacker uses equipments such as laser generators and clock manipulators to induce the fault [106]. This type of attack can be prevented by manipulating the code in such a way that either; (a) it is difficult for the attacker to locate and target these instructions or, (b) detect induced faults during execution of the program. This code manipulation process is know as code hardening.

One type of code hardening technique is the code obfuscation. Obfuscation is defined as "to make something obscure, unclear and unintelligible" by Oxford English Dictionary [107]. In software development context obfuscation is the deliberate act of creating a source and/or machine code that is difficult for other programers to understand and manipulate. Program developers may deliberately obfuscate code to conceal its purpose or logic in order to prevent tampering with it.

A common method of the latter code hardening technique is duplicating all or parts of the code. The main principle behind this technique is that induced faults should be detected by executing the duplicate codes and checking if both the results match or not. If both redundant codes generate the same result then the code is considered as secure; otherwise, the execution is terminated. The redundant code may be inserted either into the source or the machine code. In the case of the former, the source code has to pass through a tool, also known as source-to-source rewriters, that essentially inserts the code redundancy by duplicating selected statements. Source-to-source rewriters however, suffer from

major drawbacks. Firstly, modern compilers are equipped with a code optimisation tools. One of such tools is the Common Subexpression Elimination (CSE) [108] which basically removes redundant expressions/statements. During compilation the CSE searches for identical expressions and removes them. One of the great advantages of CSE is compacting the program size by removing duplicated codes. Now this risks the undoing of the security protection that is provided by redundant code execution in the first place. To ensure that sufficient redundancy survives the CSE and still remains in the generated code, the source-to-source rewriter inserts either;

1. un-optimised and un-analysed code by disabling CSE or
2. a code that is complex enough to withstand the compiler optimisation and analysis process.

Secondly, source-to-source rewriters, are very dependant on the language and the compiler being used. Hence, they need to be redeveloped (ported) for every programming language. In other words, neither the protection not the minimal performance overhead can be ported between compilers and languages. Due to the above drawbacks, it still remains a challenge how to guarantee the presence of only the necessary redundancy with acceptable performance overhead. It is very difficult to have a redundant source code statements that;

1. would survive the compiler optimisation,
2. do not limit the compiler's existing analysis and optimisation scope.

To avoid the above source-to-source rewriter drawbacks in certain cases redundancy is inserted onto the binary code of the program. Such tools are also known as Link-Time rewriters. These rewriters do not suffer from the same drawbacks as the source-to-source code rewriters. However, they suffer from a lack of high-level semantic information such as symbol and type information. This lack of information limits the precision and scope of protection provided by the binary code rewriters. The best example of binary rewriter is Diablo [109].

4.3 Side Channel Protection

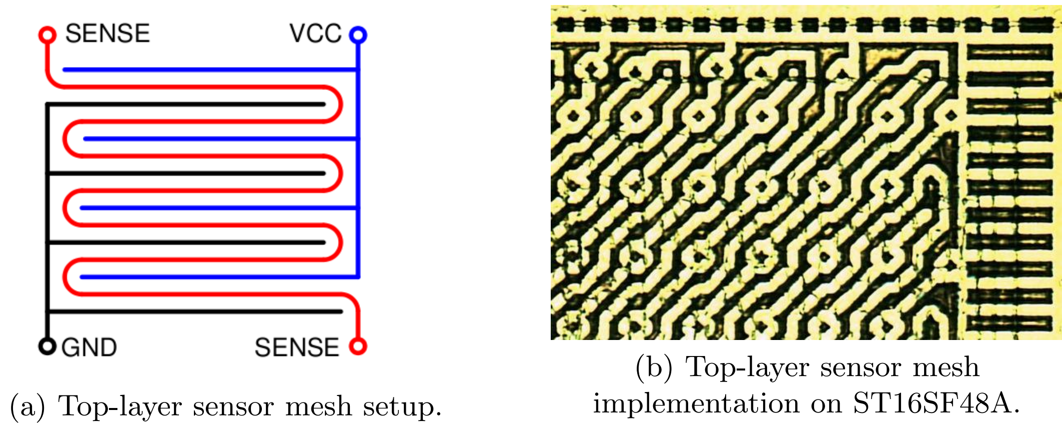
As discussed in Section 3.4.1, side channel attacks exploit the dependancy of leakages such as power consumption and electromagnetic emission. Therefore, protection against side channel attacks try to disguise the dependency of side channel leakage on the data processed. This is usually achieved by adding noise to the side channel leakage. The design challenge is putting enough countermeasure to make the attack too expensive to be interesting [110]. Several software and hardware implementations of side channel protection are proposed. Among them is Masking [111], which offers protection against DPA un-correlating the intermediate results to the actual (unmasked) intermediate values.

4.4 Tamper Resistance

Several advanced packaging and attack response techniques have been recommended by the Federal Information Processing Standard (FIPS 140-2) [112]. For example, the standard recommends four increasing levels of physical (and other) security requirements that can be satisfied by a secure system. Security Level 1 requires minimum physical protection, Level 2 requires the addition of tamper-evident mechanisms such as a seal or enclosure, while Level 3 specifies stronger detection and response mechanisms. Finally, Level 4 mandates environmental failure protection and testing (EFP and EFT).

Physical attacks are one of the most effective attacks on embedded systems. Thus, modern embedded systems implement sensor mesh to detect any physical attacks on progress [66]. Simple configuration of top-layer sensor mesh is depicted in Fig 4.1. During operation the sensor line is checked for any interruptions or short circuits which trigger countermeasure alarms. The countermeasure alarm could be processor halt or flash erase.

Another common countermeasure for physical attacks, for instance micro-probing, is changing the order of data bus to make it difficult to observe bus signals, known

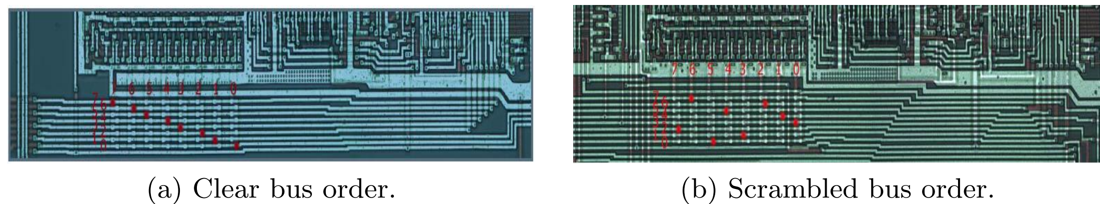


(a) Top-layer sensor mesh setup.

(b) Top-layer sensor mesh implementation on ST16SF48A.

Figure 4.1: A top layer sensor mesh setup and practical implementation [3].

as bus scrambling [16]. Figure 4.2 depicts a clear and scrambled versions of a data bus lines. Bus scrambling can be static, chip-specific or session-specific. Static scrambling uses the same scheme in every chip. This makes it difficult for an attacker to observe data bus order but not for a very long time for dedicated attacker.



(a) Clear bus order.

(b) Scrambled bus order.

Figure 4.2: A clear and scrambled bus lines [4].

The security provided by static scrambling can be improved by deploying chip-specific scheme. This can be further strengthened by using session-specific scrambling circuits where the order of the bus is changed for every session. In addition to scrambling encrypting information sent on global buses, such as data and address bus, is a common practice on modern secure processors [113, 114]. To achieve this the CPU incorporates a dedicated unit that performs the cryptographic operations. Furthermore, external memory contains only encrypted data that will be decrypted when fetched into the CPU cache.

4.5 Redundant Execution

A straight forward way of checking whether a program is executed correctly is to run it more than once and compare the results. When the results do not match an alert signal is transmitted to a decision block. At this stage the decision block responds by either reseting the system or activating a dedicated countermeasure. Redundant execution can be implemented in two ways; hardware redundancy or time redundancy. In hardware redundancy, selected or all hardware blocks of the embedded system are implemented more than once. During execution the program is fed to all blocks and their results are compared for match. Figure 4.3, shows a simple configuration of hardware redundant execution, where the program is executed by two identical but separate hardware blocks.

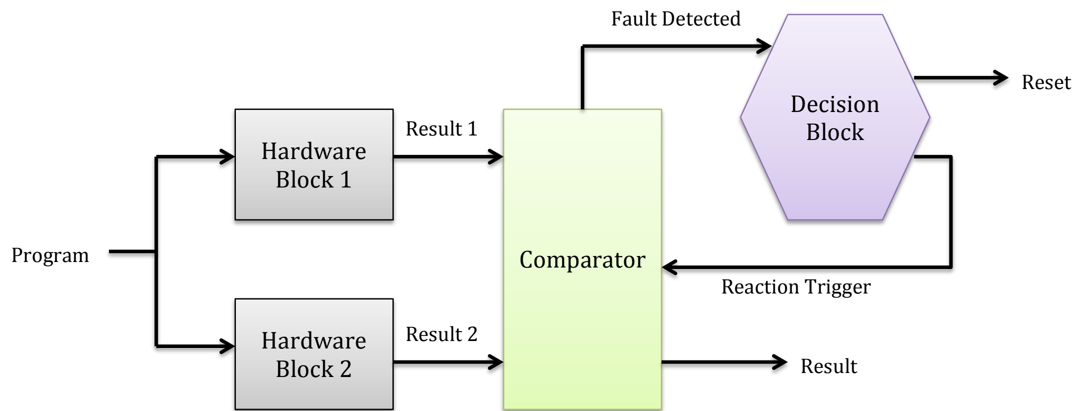
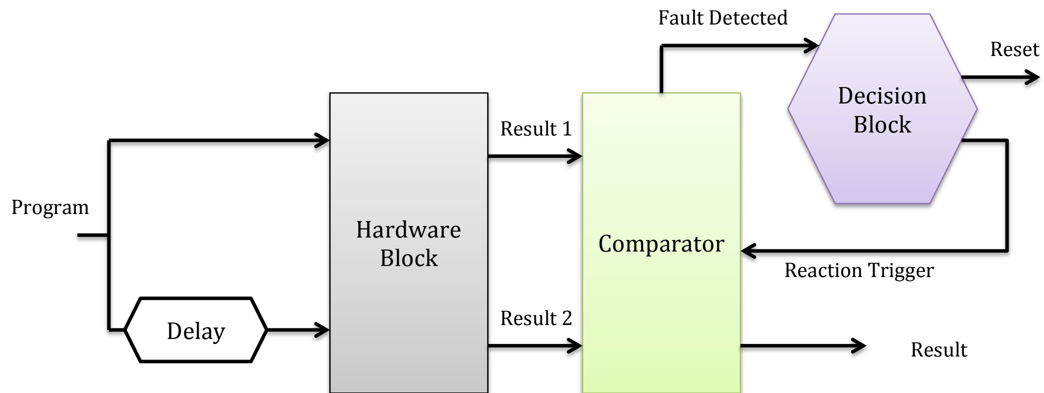


Figure 4.3: A simple hardware redundant execution setup.

On the other hand, in time redundancy the program is executed by the same hardware block but multiple times. In this configuration a delay circuit is used to feed the program to the processor multiple times. Figure 4.4, depicts a simple time redundant execution setup, where the program is executed twice by the same hardware. As in hardware redundancy, the results are checked for match.



(b) Time redundancy

Figure 4.4: A simple time redundant execution setup.

4.6 Trusted Platform Module (TPM)

The TPM chip, whose specification is defined by the Trusted Computing Group [26], is known as a hardware root-of-trust into the trusted computing ecosystem. Currently it is deployed to laptops, PCs, and mobiles and is produced by manufacturers including Infineon [115], Atmel [116] and Broadcom [117]. At present, the TPM is available as a tamper-resistant security chip that is physically bound to the computer's motherboard and controlled by software running on the system using well-defined commands. The TPM MOBILE with Trusted Execution Environment has recently emerged; its origin lies in the TPM v1.2 a with some enhancements for mobile devices [26] . The TPM provides:

1. The **Roots of trust** includes hardware/software components that are intrinsically trusted to establish a chain of trust that ensures only trusted software and hardware can be used.

2. The **Platform Configuration Register "PCR"** in the most modern TPM includes 24 registers. It is used to store the state of system measurements. These measurements are represented normally by a cryptographic hash computed from the hash values (SHA-1) of components (applications) running on the platform. PCRs cannot be written directly; data can only

be stored by a process called extending the PCR.

3. The **RSA keys**: There are three types of RSA keys that TPM generates and which are considered as *root keys* (they never leave the TPM):

   (a) **Endorsement Key (EK)**: This key is used in its role as a *Root of Trust for Reporting.* During the installation of an owner in the TPM, this key is generated by the manufacturer with a public/private key pair built into the hardware. The public component of the EK is certified by an appropriate CA, which assigns the EK to a particular TPM. Thus, each individual TPM has a unique platform EK. For the private component of the EK, the TPM can sign assertions about the trusted computer's state. A remote computer can verify that those assertions have been signed by a trusted TPM.

   (b) **Storage Root Key (SRK)**: This key is used to protect other keys and data via encryption.

   (c) **Attestation Identity Keys (AIKs)**: The AIK is used to identify the platform in transactions such as platform authentication and platform attestation. Because of the uniqueness of the EK, the AIK is used in remote attestation by a particular application. The private key is non-migratable and protected by the TPM and the public key is encrypted by a storage root key (or other key) outside the TPM with the possibility to be loaded into the TPM. The security of the public key is bootstrapped from the TPM's EK. The AIK is generally used for several roles: signing/reporting user data; storage (encrypting data and other keys); and binding (decrypting data, used also for remote parties).

4.7 ARM's TrustZone

The ARM's TrustZone provides the architecture for a secure trusted platform for a wide range of devices including handsets, tablets, wearable devices and other

enterprise systems. The underlying concept is the provision of two virtual processors with hardware-level segregation and access control [118, 28]. This enables the ARM's TrustZone to define two segregated execution environments described as Secure world and Normal world. A generic architectural view of ARM's TrustZone is depicted in Fig. 4.5. The Secure world executes the security and privacy-sensitive components of applications and normal execution takes place in the Normal world. The switch between the secure and normal world is managed by a dedicated module described as Monitor Mode. The ARM's TrustZone is implemented as a security extension to the ARM processors (e.g. ARM1176JZ(F)-S, Cortes-A8, and Cortex-A9 MPCore) [28], which a developer can opt to utilise if required.

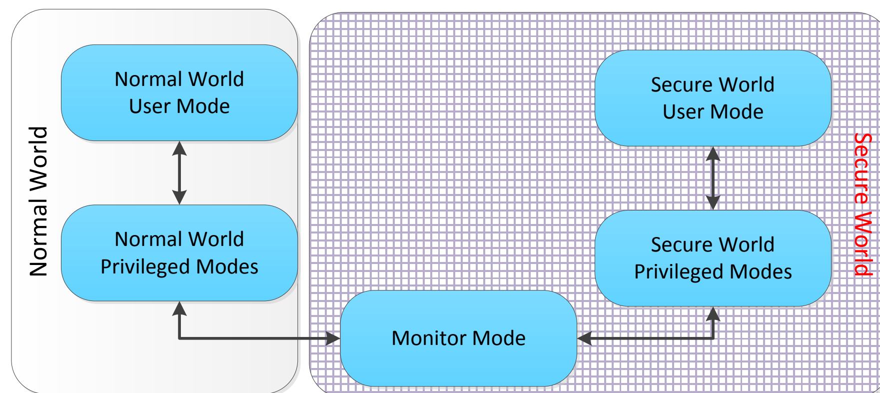


Figure 4.5: Generic architectural view of ARM TrustZone

4.8 GlobalPlatform

GlobalPlatform is a inter-industry organisation that publishes and promotes security and interoperability among applications on secure chips [119]. In this section we discuss two GlobalPlatform standards.

4.8.1 Application Management

GlobalPlatform is an application scheme that is used in smart cards and secure elements. It provides a platform for different entities to perform various tasks during the life cycle of an application, from developing, verifying, loading to deleting it. The GlobalPlatform card security requirement specification [120], specifies nine entities that perform various in the overall card application management architecture. The card issuer in this platform is responsible for acquiring the card, set policies and issue it to individual customers. The card administrator then manages the cards once they are issued to individual users. If application providers want to issue their application, they have to get it verified by the verification authority. The verification authority performs an off-card code verification to ascertain whether the given code conforms to the security policy set by the card issuer. Once the verification is performed, the application provider requests the controlling authority to give permission to load the application. The controlling authority checks the verification authority's verification and issues the permission to load the application. Finally, the application provider sends its application to the application loader, which will install it onto individual customer smartcards.

4.8.2 Trusted Execution Environment

The GlobalPlatform Trusted Execution Environment (TEE) is GlobalPlatform's initiative [121, 122, 29] for mobile phones, set-top boxes, utility meters, and payphones. GlobalPlatform defines a specification for interoperable secure hardware, which is based on GlobalPlatform's experience in the smart card industry. It does not define any particular hardware, which can be based on either a typical secure element or any of the previously discussed tamper-resistant devices. The rationale for discussing the TEE as one of the candidate devices is to provide a complete picture. The underlying ownership of the TEE device still predominantly resides with the issuing authority, which is similar to GlobalPlatform's specification for the smart card industry [123].

4.9 Remaining Security Challenges

Security, in one form or another, is a requirement for an increasing number of embedded systems applications, ranging from low-end systems such as Personal Digital Assistants (PDAs), wireless handsets, networked sensors and smart cards to high-end systems such as routers, gateways, firewalls, storage servers and web servers [124]. The security needs in embedded systems can be divided into security needs for data transfer and security needs within the embedded device [125]. The former deals with transforming outgoing data into unintelligible form so that unauthorised entities will not be able to understand. The security challenges regarding this need are solved by using encryption, digital signature and hashing algorithms. However, the latter deals with securing the embedded processor from within. This may involve adding physical attack protections; such as mesh sensors, secure ROM and secure bootloader.

Another security risk in the supply chain of embedded systems is the growing practice of outsourcing the device production to countries with cheaper infrastructure cost. While this reduces the cost of production significantly, it also make it easy for a hacker to compromise the supply chain and introduce a hidden backdoor into the design. Such a backdoor can be hard to detect with purely functional testing. This threat to the embedded systems supply chain is already a cause for alarm in some countries [126, 127]. For this reason, some governments have been subsidizing few high-cost local foundries for producing components used in military applications [128].

The security countermeasures discussed in Sections 4.2, 4.3, 4.4, 4.4, 4.6, 4.7 and 4.8 provide protection against attacks discussed in Chapter 3. However, they have certain limitations. For example, it is difficult to ensure sufficient redundancy in code hardening. In TPM, only marked sections of the program securely and the same goes for GlobalPlatform TEE. Therefore, generic security architecture that can be used in low-end to medium-end processors is necessary.

The reasons discussed above brings the need to verify embedded devices before they are deployed in operation to the foreground. Furthermore, monitoring their

operation for any unusual activities while in operation is paramount. The complexity of embedded processor cores design, including multi-processor cores, along with constrained resources make implementation of security features a challenge in embedded systems [129]. This is because, as discussed in Chapter 2, embedded systems are a lot more resource constrained in terms of their area restriction, storage capacity, processing capabilities and energy consumption. In this thesis we divide the security needs into post-production pre-deployment and runtime security needs.

4.9.1 Post-production Pre-deployment Device Verification

Outsourcing the production of chips, to cheaper cost structure countries, has a number of economic advantages for designers. Normally, a designer sends the ICs design along with the ROM contents, such as firmwares and other native functionalities, to be manufactured. In this scenario, an attacker may hack the supply chain and subvert the original design. According to report by U.S. Department of Commerce [130], defective components incidents have increased from 3,868 in 2005 to 9,356 in 2008[1]. Defective electronic components have at least the following ramifications; (a) original component providers incur an irrecoverable loss due to the sale of often cheaper defective components, (b) low performance of defective chips (that are often of lower quality and/or cheaper older generations of a chip family) affects the overall efficiency of the integrated systems that unintentionally uses them; this could in turn harm the reputation of authentic providers, (c) unreliability of defective devices could render the integrated systems that uses them; this potentially affects the performance of weapons, airplanes, cars or other crucial devices [131], and (d) untrusted components may have intentional malware or backdoor for spying information, remotely controlling critical objects and leaking secret information.

These ramifications and their growing presence in the market begs the question; how can these devices be detected before they are integrated into the final product? Individual chips can be reverse engineered and checked for fidelity of the

[1] We could not find a more recent publication on those figures.

original design. However, this technique is destructive and does not guarantee other chips that are not subjected to the test are not compromised. Another option is camouflaging and obfuscation of critical chips [128], where critical ICs are requested to be manufactured with other non-critical chips and hiding critical functionalities in a confusing logic. Unfortunately, this will not deter a committed attacker willing to spend some effort and time to subvert the original design.

4.9.2 Runtime Secure Execution

In this chapter we have discussed several security techniques that are deployed in embedded devices. However, embedded systems still remain vulnerable to a range of attacks that target runtime attributes of a program. This is partly true because security was not a priority during the design of early processor cores and it has not changed much since.

The common theme of currently deployed security techniques in embedded devices is either to obfuscate programs or the device logic, check for secure boot or divide programs in to critical and non-critical parts and executing the critical parts securely. Unfortunately, there are still threats in each countermeasure that could be exploited by an attacker. For example, obfuscating only makes the attacks difficult but does not deter committed attackers. Secure boot protects embedded devices from booting into untrusted state. This does not provide prevention against attacks that can be applied during runtime. Furthermore, an attacker could target non-critical parts of a program to divert the execution flow in devices that only execute critical parts of a program securely.

For the rest of this thesis we focus on solving these two security challenges. First, we discuss on how defective platforms can be detected before they put in operation. We selected side channel information of the target device to achieve this. Our rational for our selection is (1) side channel leakage reflects the inside state of embedded systems, (2) verifier will have full access to the device's side channel leakage and, (3) hardware and software changes can be detected from such leakage. Second, we propose a change to traditionally accepted design concept

of embedded system. In this work we propose modifications to how the stack is used by the core processor during program execution. In addition, we propose and discuss a compiler assisted component that verifies the control flow and instructions integrity of embedded programs.

These two proposed countermeasures, seemingly independent, complement each other in ways that enhance the security of embedded systems before and after deployment. Under each category we propose different techniques. For instance for the pre-deployment we propose methods to verify the integrity of executed instructions and control flow jumps. Separately both techniques protect different attributes of a program (i.e either the instructions or the control flow jumps). However, when combined together they protect the device from attacks that seek to exploit both the program's instructions and control flow jumps. The same goes for the runtime execution countermeasures. When the countermeasures proposed in Chapters 7 and 8 are combined they protect the target program's runtime data, instructions and control flow jumps during the program's execution.

At this point it may be worth mentioning some of the limitations of our proposals. Some of the techniques proposed utilise dynamic code analysis tools to compute a program's basic blocks and their valid control flow jumps between them. The result of such analysis is a list of valid execution paths from the beginning to the end of the program execution. Normally, such available techniques suffer from coverage issues. This issue may lead to some valid execution paths being excluded from the final list. This limitation of dynamic code analysis tools also affect the efficiency and accuracy of our proposed countermeasures. Details of the proposed countermeasures is discussed in subsequent chapters of this thesis.

4.10 Summary

In this chapter we have looked into the current attack protection techniques deployed in embedded systems. The countermeasures range from a simple source code manipulation (to add redundant markers within the object code), side channel leakage masking, tamper resistant shield to a dedicated hardware module.

The dedicated hardware provides either program authentication during booting or curtained execution environment. The program authentication is done through a cryptographic signature over selected attributes of the application. The core theme of the dedicated curtained execution environments is that the programmer labels sections of the application as secure during compilation. Then the processor treats them differently from the rest of the application during execution. Finally, we concluded the chapter by discussing the limitations of such countermeasures.

Part I

Post-production Pre-deployment Measures

Chapter 5

Control Flow Verification

Contents

5.1 Introduction **76**

5.2 Device Modelling **78**

5.3 Control Flow Reconstruction **84**

5.4 Control Flow Verification **85**

5.5 Implementation and Results **86**

5.6 Summary **91**

In this chapter, we discuss about verification of program control flow on embedded systems before deployment. First, we explain our device modelling and the model parameters. Then we elaborate on the re-construction of program's control flow fom the device's side channel, followed by the verification of the re-constructed control flow. Furthermore, we present our implementations and results of the proposed verification system. Finally, we summarise the chapter by discussing the core points.

5.1 Introduction

In recent years, embedded systems have proliferated into a wide range of modern life applications. One of the main application vectors of embedded systems is communication [132, 133, 134]. A typical embedded system application contains hardware and software components. The hardware component includes storage areas, execution engines and other peripherals required to successfully execute instructions. The software component is a written procedures or rules stored in a memory pertaining to the operation of a computer system or part of the system itself.

The execution of a software program always involves incrementing the program counter (a special register which stores the address of the next instruction). Normally the program counter is incremented by "1"; however, certain instructions change its value by more than one in both directions. This kind of change is known as *Control Flow Change* and can be caused by both conditional and unconditional branching instructions. According to [135], program control flow is the most attacked target in software and such attacks are called *Control Flow Attacks*. *Control Flow Attack* is one of the main threats for embedded systems [136, 137, 138]. *Control Flow Attacks* can be performed on embedded systems using two approaches. First approach, the attacker installs his code segment on the target device. Then later when the device executes a genuine program, the attacker targets saved function return addresses to divert the control flow into his previously installed code. Second approach, the attacker does not install any code but instead when the program is executed the attacker changes the saved return addresses just in order to skip the execution of certain part of the program.

In the literature, several countermeasures have been proposed to counteract these kinds of intrusions. To explain some of them; in [139], the authors discuss a technique that employs a dedicated hardware module to detect and prevent unintended program behaviors. In this method the program's properties are extracted through a static code analysis and the hardware module uses them to enforce a permissible program behavior at runtime. Another countermeasure, described in [140] introduces *Control-Flow Integrity (CFI)* enforcement. The CFI

dictates that software execution must follow the path of a *Control-Flow Graph (CFG)* determined ahead of time. The work of Michael Frantzen and Michael Shuey [141], presents a buffer overflow prevention method. This is acheived via a kernel modification that performs transparent, automatic and atomic operations on the function return addresses before they are written into the stack and before the program transfers execution back to the saved return addresses. In [142], Aurélien et al. discussed a control flow enforcement technique based on Instruction Based Memory Access Control (IBMAC). This is done by using a simple hardware modification to divide the stack into a data and a control flow stack (or return stack). Moreover, access to the control flow stack is restricted only to return and call instructions, which prevents control flow manipulation. More countermeasures can be found in [143, 144, 145]. Most of the proposed countermeasures are demanding in terms of computational capability, memory usage and often rely on a hardware module that is not present on simple devices.

In this paper we present a novel approach to verify a program's control flow by using the device's side channel leakage. In our proposal we modelled the device as a *Markov Process* [146] with hidden states, each state belonging to a part of the program. Then a verifying device extracts the control flow transition that the device had followed when executing the program from its side channel leakage (power consumption). This extracted control flow (state sequence) is then verified against a list of valid state transitions of the application which was calculated ahead of time.

The rest of the chapter is structured as follows. In Section 5.2 we discuss device model building techniques. We provide detailed explanation of how the model parameters are constructed. In addition, we discuss techniques to speed up the process of model parameters construction. Section 5.3 discusses how the device parameters can be used to reconstruct the control flow transfers that goes inside the processor. In Section 5.4 we discuss how the reconstructed control flow can be verified of its validity. The implementation and results of all techniques discussed above are presented in Section 5.5. Finally, we concluded the chapter by discussing the main concepts of our proposal in Section 5.6.

5.2 Device Modelling

An embedded program is a combination of basic blocks. A basic block is a linear sequence of executable instructions with only one entry point (the first instruction of the basic block) and one exit point (the last instruction of the basic block) [147]. After executing one basic block the processor jumps into another basic block determined by the branching instruction executed at the end of the current basic block. This branching instruction can be conditional or unconditional. A basic block may have many predecessors and many successors. It might also be its own successor. Program entry basic blocks might not have predecessors that are within the program and program ending basic blocks never have successors within the program itself.

Based on the above definition of embedded programs, it can be modelled as a state machine with each basic block corresponding to a state and the branching statements to a state transition. To verify the validity of control flow transfers of a pre-installed program without prior knowledge of the program itself, we need an additional information. At runtime we can not directly observe which states of the program are being executed. However, we can observe the side-channel information of the processor. This side-channel information can be power consumption [75, 148] or electro-magnetic emission [81, 80, 82]. Side channel information collected from a processor running a program is directly dependent on and reveals partial information about all executed program states.

Taking the program definition and the side channel information into consideration, an embedded processor running a program can be modelled as a *Hidden Markov Model (HMM)* [146, 149]. A *Hidden Markov Model* is a memoryless system with a finite number of hidden states. It is called memoryless because the next state depends only on the current state. Figure 5.1, illustrates a *Hidden Markov Model* representation of a processor executing a program with five hidden states (i.e. A to E). The observable output of the HMM, the power consumption, is measured via a resistor (R_s) connecting the ground pin of the device and ground pin of the voltage source. Building a complete HMM of such processor requires three parameters.

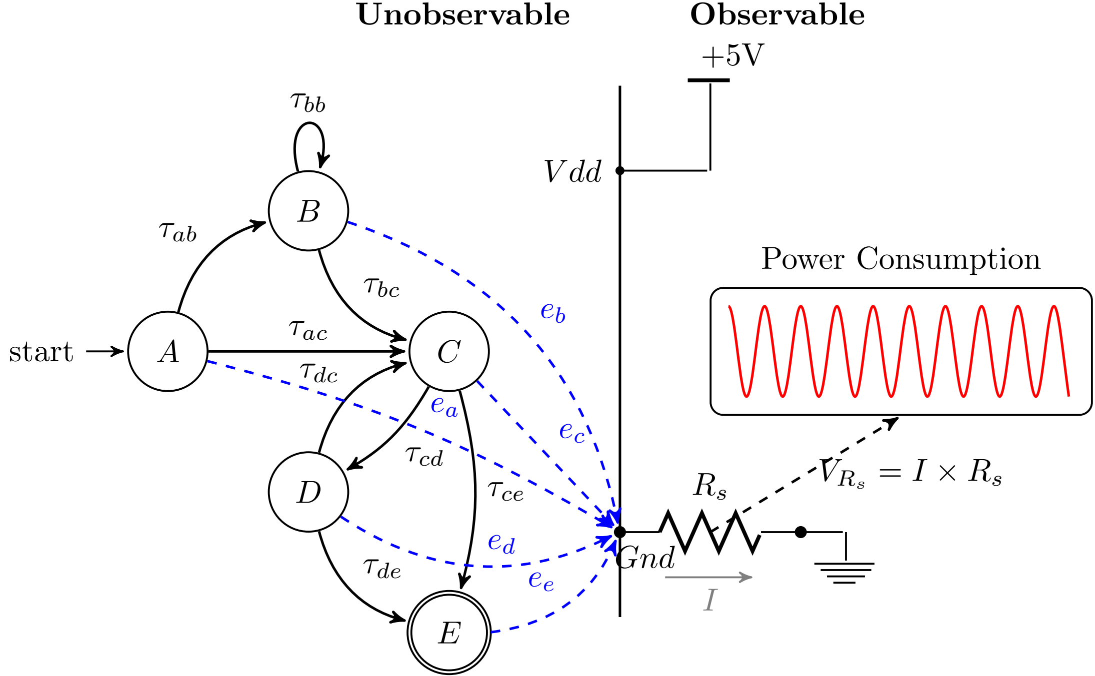


Figure 5.1: A Hidden Markov Model representation of a device executing a program with five states (A, B, C, D and E). The power consumption is the observable output that reveals partial information about the executed states.

5.2.1 Model Parameters

Given a set of finite states $\mathbf{Q} = \{q_i\}$, where $1 \leq i \leq S$ and S is the number of states, building a processor's *Hidden Markov Model* requires a transition probability distribution matrix $\mathbf{T} = \{\tau_{ij}\}$, an emission probability distribution matrix $\mathbf{E} = \{e_i\}$ and an initial state distribution $\vec{\pi}$. Having these probability distribution matrices, the HMM is defined as $\lambda = (\mathbf{T}, \mathbf{E}, \vec{\pi})$.

The transition probability distribution τ_{ij}, is the probability that the next state to be executed is q_j if the current state is q_i, where $1 \leq i, j \leq S$. If we denote s_t as the state that the processor executes at a time t, the $\tau_{ij} = \mathcal{P}(s_{t+1} = q_j \mid s_t = q_i)$ is the probability of state transitioning from state q_i to state q_j. Given an observable emission, in this case the power consumption, $\mathcal{O}_t$ at time t, the emission probability distribution $e_i(\mathcal{O}_t) = \mathcal{P}(\mathcal{O}_t \mid s_t = q_i)$ is the probability that $\mathcal{O}_t$ belongs to the processor executing state q_i. To compute $e_i(\mathcal{O}_t)$ first we need

to build a power consumption template for each state.

The template of a state is generated by computing the mean, μ_{q_i}, and the covariance, σ_{q_i} of the state's power consumption traces. Let us consider N L-dimensional power consumption traces $\{x_n\}$ collected from the target device while executing the state q_i repeatedly. The mean, μ_{q_i}, and covariance, σ_{q_i}, are calculated using the formulas in equations (5.1) and (5.2) respectively.

$$\mu_{q_i} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{5.1}$$

$$\sigma_{q_i} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu_{q_i})(x_n - \mu_{q_i})^T \tag{5.2}$$

where N is the number of recorded power traces for state q_i and $(x_n - \mu_{q_i})^T$ is the transpose of $(x_n - \mu_{q_i})$. These templates can be built beforehand using a target program in an identical reference device.

Given the mean and covariance, and assuming the power traces are derived from a *Multivariate Gaussian Normal Distribution Model* [150], the emission probability distribution $e_i(\mathcal{O}_t)$ is computed as shown in equation (5.3).

$$e_i(\mathcal{O}_t) = \frac{1}{(2\pi)^{L/2}\sqrt{\sigma_{q_i}}}\exp(-\frac{1}{2}(\mathcal{O}_t - \mu_{q_i})\sigma_{q_i}^{-1}(\mathcal{O}_t - \mu_{q_i})^T) \tag{5.3}$$

Now, if we take a number of time domain observations $\mathcal{O} = \{\mathcal{O}_t, \mathcal{O}_{t+1}, \mathcal{O}_{t+2}, \cdots, \mathcal{O}_{t+n}\}$, the emission probability distribution matrix $\mathbf{E}$ becomes:

$$\mathbf{E} = \begin{bmatrix} e_1(\mathcal{O}_t) & e_1(\mathcal{O}_{t+1}) & e_1(\mathcal{O}_{t+2}) & \cdots & e_1(\mathcal{O}_{t+n}) \\ e_2(\mathcal{O}_t) & e_2(\mathcal{O}_{t+1}) & e_2(\mathcal{O}_{t+2}) & \cdots & e_2(\mathcal{O}_{t+n}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e_S(\mathcal{O}_t) & e_S(\mathcal{O}_{t+1}) & e_S(\mathcal{O}_{t+2}) & \cdots & e_S(\mathcal{O}_{t+n}) \end{bmatrix} \tag{5.4}$$

Transition probability distribution is a simple statistical representation of the transition between states. For example, in Figure 5.2, we present two code segments with three basic blocks (states).

```
int X = RG() % 100;          s1
   ⋮
if(x ≥ 10){
   ⋮                          s2
} else {
   ⋮                          s3
}
```

(a)

```
boolean b;                   s1
   ⋮
if(b){
   ⋮                          s2
} else {
   ⋮                          s3
}
```

(b)

Figure 5.2: Integer and boolean branching statements in code segment with 3 states.

The significant difference between the two code segments is the state transition instruction i.e. the *if ... else* statement. Code segment (a) compares a random integer X that can have a value between 0 to 99. After completing state s_1, if $0 \leq X \leq 9$ execution control transfers to s_3 but if $10 \leq X \leq 99$ s_2 is executed next. Now given that X is randomly generated using a uniformly distributed random generator, the probability of s_3 being executed after s_1 is 0.1 and s_2 after s_1 is 0.9. However, for code segment (b) the branching statement simple checks if b is '0' or '1'. The probability of going to s_2 or s_3 is the same and that is 0.5. Given these information, the transition probability distribution matrix for code segments (a) and (b) will look like as shown below (equation 5.5) respectively.

$$\mathbf{T} = \begin{bmatrix} \tau_{11} & \tau_{21} & \tau_{31} \\ \tau_{12} & \tau_{22} & \tau_{32} \\ \tau_{13} & \tau_{23} & \tau_{33} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0.9 & 0 & 0 \\ 0.1 & 0 & 0 \end{bmatrix} \textit{ and } \begin{bmatrix} 0 & 0 & 0 \\ 0.5 & 0 & 0 \\ 0.5 & 0 & 0 \end{bmatrix} \tag{5.5}$$

Normally, a program's execution always starts at the program's entry basic block, i.e. the basic block which starts with *main()* or simple at the processor's reset point. Therefore, the initial state distribution for the entry basic block is always

'1' and '0' for the other basic blocks. For instance, for the program state machine depicted in Fig. 5.1 the program execution always starts at state "A". So, the initial state distribution of the whole program will look like as shown below in equation (5.6).

$$\pi = \{\pi_A, \pi_B, \pi_C, \pi_D, \pi_E\} = \{1, 0, 0, 0, 0\} \tag{5.6}$$

So far we have discussed how a device's HMM parameters are computed from the program's basic blocks and the processor's side channel information. However, there are still a few challenges that are worth discussing at this point. Firstly, to successfully compute $\mathbf{E}$ using equation (5.3), all observations $\{\mathcal{O}_t, \cdots, \mathcal{O}_{t+n}\}$ must have equal dimensionality. In other words, the power consumption traces generated by all states must have the same number of sample points. However, in reality this may not always be true. Secondly, the dimension of the emissions (power traces) may be too large for a robust and fast classification. Fortunately, both challenges can be addressed using the same technique. A common way to attempt to resolve these challenges is to use a dimensionality reduction technique. However, we have to maintain as much information about the original emission (power consumption) as possible while reducing the dimentions of the traces. Two of the most popular techniques that can be used for such purpose are: the *Principal Components Analysis (PCA)* and *Fisher's Linear Discriminant Analysis (F-LDA)*.

5.2.2 Principal Components Analysis (PCA)

PCA is a technique used to reduce the dimension of an observation while keeping as much of its variance as possible [151]. This is achieved by orthogonally projecting the observation onto a lower dimensional subspace vector.

Let us consider an N L-dimensional observations of emissions $\{x_n\}$, where $n = 1, ..., N$ and their covariance matrix σ. A lower dimensional subspace in this Euclidean space can be defined by a D-dimensional unit vector $\overrightarrow{u_1}$, where $D < L$.

The projection of each observation, x_n, onto that subspace is given by $\overrightarrow{u_1}^T x_n$. Now if we stack up all the emissions into a matrix of $N \times L$ matrix, where L is the number of samples of each observation, the projection of each row of the matrix is represented as $U^T X$, where U is a matrix of *eigenvectors* of the covariance matrix σ. The projection of the observations onto a D-dimensional subspace that maximizes the projected variance is given by D *eigenvectors* [152] $\overrightarrow{u_1}, \ldots, \overrightarrow{u_d}$ with the D largest *eigenvalues* $\lambda_1, \ldots, \lambda_d$.

5.2.3 Fisher's Linear Discriminant Analysis (F-LDA)

F-LDA is a method used in statistics, pattern recognition and machine learning to find a linear combination of features which characterises two or more class observations [153, 154, 155]. The resulting combination may be used as a linear classifier for dimensionality reduction before classification. However, instead of maximising the variance of the original data like PCA, information regarding the covariance of different classes is taken into consideration. These are the "between-class" and "within-class" covariance matrices.

Now, let us consider again the N L-dimensional observations for each class. Then the "within-class" covariance σ_W is computed as,

$$\sigma_W = \sum_{i=1}^{S} \sum_{w \in x_i} (w - \mu_{q_i})(w - \mu_{q_i})^T = \sum_{i=1}^{S} N_{q_i} \sigma_{q_i} \tag{5.7}$$

In the above equation, N_{q_i}, σ_{q_i} and w are the number of observations, the covariance and the power traces of class q_i. The "between-class" covariance σ_B is computed as

$$\sigma_B = \sum_{i=1}^{S} (\mu_{q_i} - \mu)(\mu_{q_i} - \mu)^T \tag{5.8}$$

where μ_{q_i} is the individual class's mean as defined in equation (5.1) and μ is the

mean of the entire observation which is computed as shown in equation (5.9).

$$\mu = \frac{1}{N}\sum_{\forall x} x = \frac{1}{N}\sum_{i=1}^{S} N_{q_i}\mu_{q_i} \tag{5.9}$$

Now, let us consider a D-dimensional unit vector $\overrightarrow{u_1}$ onto which the data is projected. This time the objective is to maximise both the projected "between-class" and the projected "within-class" covariance:

$$\mathcal{J}(\overrightarrow{u_1}) = \frac{\overrightarrow{u_1}^T \sigma_B \overrightarrow{u_1}}{\overrightarrow{u_1}^T \sigma_W \overrightarrow{u_1}} \tag{5.10}$$

The projected $\mathcal{J}$ is maximised if $\overrightarrow{u_1}$ is the *eigenvector* of $\sigma_W^{-1}\sigma_B$. The D-dimensional subspace is created by the first D orthogonal directions that maximise the projected $\mathcal{J}$. These are given by the D *eigenvectors* $\overrightarrow{u_1}, \cdots, \overrightarrow{u_D}$ of $\sigma_W^{-1}\sigma_B$ with the largest *eigenvalues* $\lambda_1, \cdots, \lambda_D$.

5.3 Control Flow Reconstruction

Having the HMM model parameters we need to reconstruct the control flow transfers that goes inside the device before trying to verify if it was valid or not.

The probability distribution matrices $\mathbf{E}$, $\mathbf{T}$ and $\vec{\pi}$ can be created prior to the control flow reconstruction using an identical reference device and the target program. This phase can also be done by a third party and the verifier does not need to have prior information on the inner working of the program.

Now let us consider observations (power consumption traces) $\mathcal{O}' = \{\mathcal{O}'_t, \mathcal{O}'_{t+1}, \mathcal{O}'_{t+2}, \cdots, \mathcal{O}'_{t+n}\}$, where n is the number of executed states. These emissions sre recorded while the device was executing the target program. The most likely sequence of states that produces the observations $\mathcal{O}'$ can be calculated using the *Viterbi Algorithm* [156] as shown in equations (5.11) and (5.12). The calculated

state sequence is then regarded as the control flow that the device has followed during the execution of the program.

$$\mathcal{V}_{1,j} = \mathcal{P}(\mathcal{O}_1 \mid s_1 = q_j) \cdot \pi_j \tag{5.11}$$

$$\mathcal{V}_{t,j} = \mathcal{P}(\mathcal{O}_t \mid s_t = q_j) \cdot max_{i \in S}(\tau_{ij} \cdot \mathcal{V}_{t-1,j}) \tag{5.12}$$

In equation (5.12), S is the state space of the *Markov Process*, π_j is the probability of state q_j being the initial state and τ_{ij} is the probability of transitioning from state q_i to state q_j. The $\mathcal{V}_{t,j}$ is the probability of the most probable state sequence responsible for the first t emissions that has q_j as its final state. The state sequence that resulted in highest probability, according to equation (5.12), from all possible state sequences of the same length as the emission is regarded as the most probable state sequence that generated the emissions.

5.4 Control Flow Verification

As described in Section 5.2, a program is a combination of basic blocks. Before loading the program into the target device, a list of valid transitions between the states (basic blocks) is extracted using a code analysis tool. This list of valid transitions is known as the *Control Flow Graph (CFG)*. A CFG, $G = (I, P)$, is represented by the program's states identity, I, and control flow path, P. For instance, for the program illustrated in Fig. 5.1, the CFG is given as $G = (I, P)$, where $I = \{A, B, C, D, E\}$ and P=$\{$*(A,B),(A,C),(B,B),(B,C),(C,D),(C,E),(D,C),(D,E)*$\}$.

Now the task is verifying if the reconstructed state sequence is among the valid transitions in the CFG. However, the reconstruction of the state sequence (explained in Section 5.3) from the power consumption is a probabilistic process. In other words, the reconstructed sequence is an execution path with a highest probability of generating the given power consumption. Therefore, it is logical to

confirm the reconstructed sequence is the execution path followed by the target device while executing the program. This can be done by assigning each state with unique identity and then later verify their hash value.

Let's assume that each state (basic block) is assigned with a unique identiy during compilation. At runtime the processor (target device) concatenates the identiy of executed states, compute hash value on it (H^*) at the end of execution and send it out to the verifier. In the mean time another hash value ($H^{'}$) is computed over the identity of states reconstructed from the power consumption. Finally, these hash values are verified using equation (5.13).

$$f(H^*, H^{'}) = \begin{cases} 1, & \text{if } H^* = H^{'} \\ 0, & \text{otherwise} \end{cases} \tag{5.13}$$

If it is a match, the reconstructed sequence is what the processor went through when executing the program. Otherwise, the reconstructed sequence is not the path that was followed by the device. Equation (5.13) can only verify that the execute state sequence and the extracted state sequence are the same. Unfortunately, this does not verify if the executed state sequence (control flow) is valid. Therefore, the validity of the control flow is verified by comparing it against the pre-calculated paths, P, in CFG. If the reconstructed state sequence is not among the valid paths in CFG, the device/program is regarded as compromised.

5.5 Implementation and Results

For our experiment we implemented a test application with five basic blocks (states). This test application is implemented in ATMega163 based smart card [157]. Each state accomplishes certain task within the program. The processor follows different control flow paths to execute the application depending on a value "V_{reader}" sent from a terminal. The state machine diagram of the test application and its pseudo code are presented in Figure 5.3 and Figure 5.4.

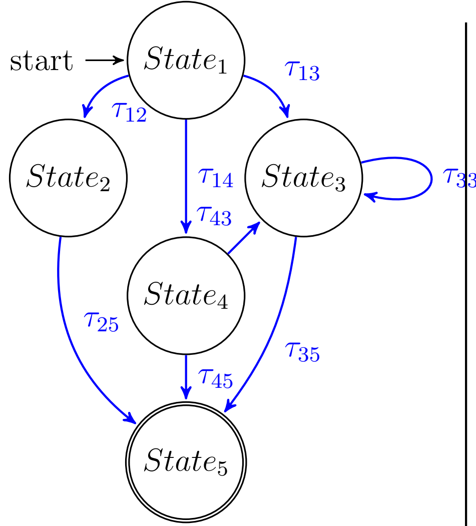


Figure 5.3: Test program's control flow diagram.

```
State1: Par = receive()
        Vreader = receive()
        Vnvm = read(nvm)
        if(Vreader == Vnvm)
State2:     par = (par)^2
            goto State5
        end
        else if(Vreader > Vnvm)
State4:     par = par + 216
            par = par/5
            Vreader = Vreader - 2
            if(Vreader < Vnvm)
                goto State3
            end
            else
                goto State5
            end
        end
        else if(Vreader < Vnvm)
State3:     par = par * 2
            par = par - 129
            Vreader = Vreader + 1
            if (Vreader < Vnvm)
                goto State3
            end
            else
                goto State5
            end
        end
State5: clear_registers
        clear_memory
```

Figure 5.4: High-level description of the test program

Invoking the test program requires passing two arguments: "V_{reader}" ($0 \leq V_{reader} \leq 9$) and "Par" ($0 \leq Par \leq 255$). The "V_{reader}" is compared with a reference value "V_{nvm}" ($0 \leq V_{nvm} \leq 9$) (stored in the non-volatile memory of the smart card) before changing a state. For our experiment the V_{nvm} is initialised to "4" and the arguments Par and V_{reader} are randomly generated and passed to the program through the smart card reader.

5.5.1 Model Parameters

As illustrated in Figure 5.3, the execution of the test program always starts at $State_1$. Therefore, the probability of $State_1$ being the initial state is "1", and "0" for all other states. If π_i is the probability of $State_i$ being the initial state in the execution of the program, the initial probability distribution vector of our

Table 5.1: Transition probability distribution of the program illustrated in Figure 5.3. The columns represent next states and the rows represent current states.

Transition from	**Transition to [%]**				
	$State_1$	$State_2$	$State_3$	$State_4$	$State_5$
$State_1$	τ_{11}=0	τ_{12}=0.1	τ_{13}=0.4	τ_{14}=0.5	τ_{15}=0
$State_2$	τ_{21}=0	τ_{22}=0	τ_{23}=0	τ_{24}=0	τ_{25}=1
$State_3$	τ_{31}=0	τ_{32}=0	τ_{33}=0.55	τ_{34}=0	τ_{35}=0.45
$State_4$	τ_{41}=0	τ_{42}=0	τ_{43}=0.2	τ_{44}=0	τ_{45}=0.8
$State_5$	τ_{51}=0	τ_{52}=0	τ_{53}=0	τ_{54}=0	τ_{55}=0

test program is given as:

$$\vec{\pi} = \{ \ \pi_1 = 1, \quad \pi_2 = 0, \quad \pi_3 = 0, \quad \pi_4 = 0, \quad \pi_5 = 0 \ \} \tag{5.14}$$

To compute the transition probability distribution matrix, $\mathbf{T}$, we invoked the program with a randomly generated "*Par*" and all possible values (i.e. 0 to 9) of "V_{reader}" and record the control-flow transition of the program. Note that for each different value of "V_{nvm}" the matrix $\mathbf{T}$ is different.

To compute the emission probability distribution matrix $\mathbf{E}$, we collected 1000 traces for each state. Using these traces we computed the mean μ_{q_i}, and covariance, σ_{q_i}, for each state as a template. Figure 5.5 shows the mean value of the traces that we collected.

Principal Components Analysis (PCA): is used to find a subspace whose basis vectors corresponding to the maximum variance directions in the original data. In other words PCA searches for those vectors in the underlying data that best describes the data. When applying *PCA* the dimensionality of the projected data has to be selected carefully. On the one hand, if it is too small, too much of variance of the original data may get lost and with it important information about the state emissions. On the other hand, if it is too large, the state classification becomes less reliable again. This might be because of the bad conditioning of large covariance matrix. Another reason can be, as the dimension increases the class emission cross-correlation increases. Therefore, when choosing

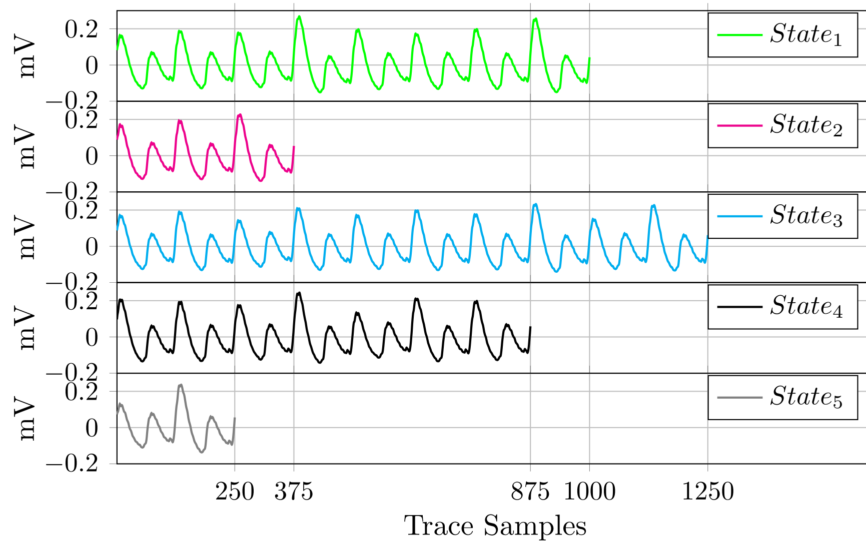


Figure 5.5: Mean of the power traces of the states illustrated in Fig. 5.3.

the dimensionality for the projected data we have to decide how much of variance of the original data that we can afford to lose.

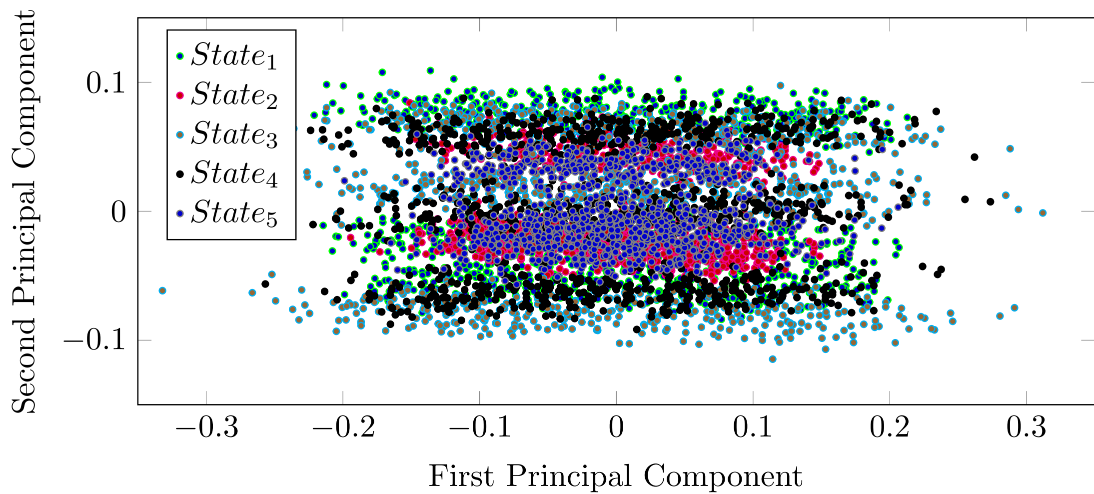


Figure 5.6: Original data after PCA.

Fisher's Linear Discriminant Analysis (F-LDA): is a technique used to classify between classes by finding discriminant features of the class data and projecting them onto these discriminant vectors. In other words, F-LDA searches for those vectors in the underlying data that best separates among the classes.

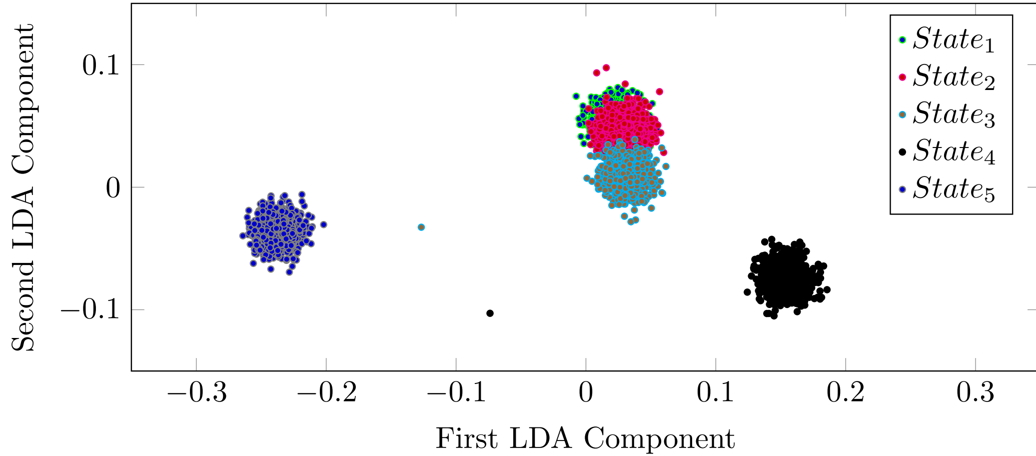


Figure 5.7: Original data after F-LDA.

In Fig. 5.7 we present the first two components of the state emissions after F-LDA. As discussed earlier PCA searches for vectors that best describes the original data. However, it does not take the other classes into consideration. For this reason PCA may not produce a satisfactory result when classifying different classes. We can see that in Fig. 5.6 the principal components of classes emissions overlap. However, as shown in Fig. 5.7 the classes are better separated after F-LDA.

5.5.2 Calculating The Most Probable State Sequence

To calculate the most probable state sequence, first we have to implement the *Viterbi Algorithm* discussed in Section 5.3. To do this we have two options: use the MATLAB [158] Statistics Toolbox implementation *hmmviterbi*[159] or create our own implementation of the equations (5.11) and (5.12). Although, the MATLAB Statistics Toolbox implementation of *Viterbi Algorithm* might be useful for some statistical calculations we could not use it in our experiment. This was because firstly it does not utilise the initial probability distribution ($\vec{\pi}$) and secondly the output is not in the format that we want it to be. Therefore, we created our own MATLAB implementation and the source code is available at GitHub repository [35]. As you can see it from the source code, our implementation takes all three matrices ($\vec{\pi}$, $\mathbf{E}$ and $\mathbf{T}$) and gives us the most likely state sequence as a vector.

Our test program has six valid control-flow paths from the initial state, $state_1$, to the final state, $state_5$. Our implementation of the *Viterbi algorithm* calculates a sequence of states with the highest probability of generating the emission $\mathcal{O}$. We ran the test program for all possible valid paths by varying the argument "V_{reader}" and calculated the most probable state sequence from the smart cards power consumption trace. We ran the test program 1000 times by varying "V_{Reader}", recorded the power trace and calculated the most likely sequence of states for each run.

5.5.3 Verifying The Reconstructed State Sequence

For all the state sequences that we calculated, we verified them using the 2-step verification system discussed in Section 5.4. Before comparing the reconstructed state sequence against the CFG, we have to make sure that the reconstructed sequence is the actual path that the device went through. For that purpose we verified the hash values calculated by the device against the hash values calculated over the reconstructed state sequence. Then we compared the reconstructed state sequence against the valid paths in CFG. In our experiment we successfully verified the control flow for all (1000) runs of the test program that we made. In our experiment we calculated the CFG manually; however, for large programs calculating it manually might be difficult and complicated. In such a case the CFG may be extracted using source code analysis tools, such as MALPAS [160].

5.6 Summary

In this chapter we proposed a novel approach into checking a program's control flow integrity by using the side channel leakage of the target device before it is integrated into the larger electronic equipment. In our approach the device is not required to perform extra computation. However, it requires another device to check for its program's control flow integrity as it executes the program. Our approach can be used to verify the integrity of mass produced embedded systems

by using a few legitimate devices. The legitimate devices are used to build a side channel fingerprint of the target device. These templates are then used to verify the mass produced devices.

Chapter 6

Software Integrity Verification

Contents

6.1 Introduction 94

6.2 Instruction-Level Template Construction 95

6.3 Dimensionality Reduction 97

6.4 Instruction Classification 98

6.5 RSA Signature Screening Algorithm 100

6.6 Basic Block Integrity Verification 101

6.7 Implementation and Results 104

6.8 Summary 115

In this chapter, we discuss verifying the basic block integrity of embedded programs. To achieve that first we explain how to create a precise power consumption template of the device's instructions. Then explain how these templates can be used to extract executed instructions. Furthermore, on how these extracted instructions can be used to verify the integrity of the program's basic blocks. Finally, we summarise the chapter by pointing out the main concepts.

6.1 Introduction

An embedded system integrates hardware and software components. The hardware components mainly comprise a processor, volatile and non-volatile memory, and IO module. The software component controls what the hardware does by using the underlying processor's executable instructions. Therefore, maintaining the integrity of the software is vital for the security of the entire system.

So far several methods have been proposed to verify the integrity of desktop software [161, 162, 163, 164]. These methods involve generation and verification of cryptographic authentication message codes on the software binary. However, these methods often fail to work in embedded environment, the main reason being the *memory read protection*[1] implemented by most of today's microcontrollers. However, in the real world these processors leak information about their internal state unintentionally. As an example, we can consider a game of poker, where everyone plays by the same functional rules and keeps their cards well concealed. If a novice player looks worried or excited when he receives his cards, then he leaks information about his hand to the other players. An experienced player may manipulate his reaction (block his emotions/expressions or fake them) to fool the other players. However, if other physiological reactions (such as heart beat, blood pressure, respiratory rate and electro-dermal activity) of the players are measured then even an expert player's deception can be detected. Of course, measuring such physiological reactions need more sophisticated instruments, like *Polygraph* [165], than reading someone's facial reaction.

Embedded systems do not have physiological reactions or emotions but as any electronic devices they have varying electric current flowing through them. This varying current gives away information about the internal state of the device in the form of variations in the power consumption or the electromagnetic emission which can be recorded and analysed. The power consumption has previously been used for the purpose of extracting secret cryptographic keys from embedded devices [75, 77, 79]. In [86, 32, 87] power consumption has been demostrated for reverse engineering embedded programs. In [166], the authors discuss, theoreti-

[1] Techniques that prevent unauthorised users from accessing memory contents.

cally, how side channel leakage can be used to fingerprint a smart card platform and then use it later to detect cloned cards. However, this paper provides a high-level and does not discuss in detail how the platform fingerprint is constructed and how a cloned card is detected. George et. al. [167], demonstrated the Hamming weight of executed instructions can create a unique power consumption fingerprint which may be enough to verify the originality of a software program.

In this chapter we present a technique for verifying integrity of the executed instructions of an embedded program. In our technique both the embedded device and the verifying device initialise and update their own parameters using pre-computed signatures and hash values of executed instructions respectively. This will be discussed in detail in Section 6.6. At the end of the execution the verifying device verifies the integrity of the embedded software using both the parameters and the RSA signature screening. This process also involves instruction-level templates and instruction classification from the device's power consumption waveform.

The rest of this chapter is structured as follows. In Section 6.2, we discuss our instruction-level side channel construction techniques. In sSection 6.3, we explain three more dimensionality reduction techniques in addition to those we explored in Chapter 5. In Section 6.4, we explain instruction classification algorithms that can be used to extract executed instruction from side channel leakage. Section 6.5 provides a discussion on batch RSA signature verification technique and section 6.6 shows how this can be used to verify the integrity of executed basic blocks. Our implementation and results of all techniques discussed above are presented in Section 6.7. Finally, we summarise the chapter in Section 6.8.

6.2 Instruction-Level Template Construction

The power consumption template of an instruction is constructed by recording and analysing the power intake of identical reference processors while executing the target instructions repeatedly. Here we make the assumption that all genuinely manufactured embedded processors of the same model have similar leakage

characteristics. The template training consumptions are collected while the reference device executes the selected instructions repeatedly. This can be achieved by running simple training programs on the reference devices.

To build the templates let us consider an N L-dimensional observations of the processor's power consumption $\{x_N\}$. Each of these N L-dimensional observations belong to one of the K selected instructions (classes) I_k, where $1 \leq k \leq K$, running under different conditions (states). The different conditions refers to data processed, registers and memory cells used by the instruction. Each of the observations have L number of sample points. The mean of the N L-dimensional observations of each instruction $\{x_N\}_{I_k}$, μ_{I_k}, is calculated as shown in equation(6.1).

$$\mu_{I_k} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{6.1}$$

Given the mean, μ_{I_k}, and the power consumption observations, $\{x_N\}_{I_k}$, of the instruction I_k the covariance matrix σ_{I_k} is calculated as follows:

$$\sigma_{I_k} = \frac{1}{N}\sum_{n=1,I_k}^{N} (x_n - \mu_{I_k})(x_n - \mu_{I_k})^T \tag{6.2}$$

Now, the template of instruction I_k is represented by the triplet of $\tau_{I_k} = (\langle\{x_N\}_{I_k}, I_k\rangle, \mu_{I_k}, \sigma_{I_k})$. However, in practice the dimentionality of the observations can be too large and have too many cross correlated feature points. In such a case the template construction may become too time consuming and result in an unreliable templates. Therefore, to solve this problem we have to employ dimensionality reduction techniques.

6.3 Dimensionality Reduction

Dimensionality reduction techniques are feature selection algorithms used to compress data while preserving as much variance of the original data as possible. In the literature, several dimensionality reduction methods have been proposed [168, 169]. Apart from the dimentionality reduction techniques PCA and F-LDA (discussed in Sections 5.2.2 and 5.2.3 respectively) we conside three more techniques in this chapter. These are the Sum of Difference of Means, Means-Variance and Means-PCA.

6.3.1 Sum of Difference of Means

Differential power has been used for correlating information leakage with the power consumption of a device [75]. In [170] the same technique has been utilised to reduce the dimensionality of traces obtained from Rivest Cipher 4 (RC4) [171]. In our work we are going to use this method to reduce dimensionality of our instruction-level traces. In order to compute the first D dimensions from the original L dimensional observations, where $D \ll L$, we performed the following computations;

- Compute the difference of each pair of mean vectors (the mean is computed as part of the instruction template),
- Compute the summation of these differences,
- Select the first D points among the highest peaks.

6.3.2 Means-Variance

The most important criterion when reducing the dimensionality of a data is to retain as much variance of the original data as possible. So, it may be reasonable to take the feature points accounting for the maximum variance of the original

data across the different classes. To identify these points we need the mean of each class μ_{I_k}, where $1 \leq k \leq K$. Now if we put these mean vectors into a matrix (with the kth row being the mean vector of the kth class), we will have a K by L matrix where L is the number of feature points of the original data. Computing the variance of each column gives us the inter-class variance of each feature point. Finally, we reduce the dimension by taking the first D features with the highest variance (where $D \ll L$).

6.3.3 Means-PCA

As discussed in Section 5.2.2, PCA maximises the overall variance of class observations but does not consider other classes. Since our aim is achieving a higher classification rate it may be reasonable to maximise the overall variance of the class means. In other words we maximise the variance of inter-class observations. The reason for this is moving the class mean vectors apart may result a higher recognition rate. To do this consider the class means as instances of the classes and compute the projection coefficients using the techniques discussed in Section 5.2.2. Later on, these projection coefficients will be used to transform the observations. Simply speaking the *Means-PCA* computes global projection coefficients that maximises inter-class variance while *PCA* computes projection coefficients that maximise intra-class variance.

6.4 Instruction Classification

The instruction classification is the process of recognising extracted instructions from the device's power consumption waveform. In this section we discuss two classification algorithms; *Multivariate Gaussian Probability Density Function* and *k-Nearest Neighbors Algorithm.*

6.4.1 Multivariate Gaussian Probability Density Function

Given the template of each instruction, $(\langle\{x_N\}_{I_k}, I_k\rangle, \mu_{I_k}, \sigma_{I_k})$, the *Multivariate Gaussian Probability Density Function* based instruction recognition is performed as follows. Let W be the power consumption waveform captured at runtime and assume that its samples are drawn from a *Multivariate Gaussian Normal Distribution* model [150]. The noise introduced into the power waveform, W, is extracted by subtracting the mean value (μ_{I_k}, which is part of the template) from the waveform as shown in equation (6.3). For the instruction I_k the noise n_{I_k} is computed as:

$$n_{I_k} = \{(W[1] - \mu_{I_k}[1]), (W[2] - \mu_{I_k}[2]), ..., (W[p] - \mu_{I_k}[p])\} \tag{6.3}$$

where p is the selected feature points of the original dimensionality if reduced. The probability of observing the noise n_{I_k} in the device's power consumption trace is then computed as shown in the equation (6.4).

$$\mathcal{N}(n_{I_k}, \mu_{I_k}, \sigma_{I_k}) = \frac{1}{(2\pi)^{D/2}\sqrt{\sigma_{I_k}}} \exp(-\frac{1}{2}(n_{I_k})\sigma_{I_k}^{-1}(n_{I_k})^T) \tag{6.4}$$

After computing the probability function, $\mathcal{N}(n_{I_k}, \mu_{I_k}, \sigma_{I_k})$, for all instructions I_k, where $1 \leq k \leq K$, the template that generates the highest probability of observing the n_{I_k} is chosen as the instruction executed by the processor.

6.4.2 k-Nearest Neighbors Algorithm

The k-Nearest Neighbors Algorithm (kNN) [172] is a non-parametric lazy supervised learning algorithm. The "non-parametric" means the learning algorithm does not make assumptions about the data and "lazy" means data generalization (training) is not needed. In a supervised learning the training data is an ordered pair $\langle x, i\rangle$, where x is an instance and i is its class label (instruction). The goal

of the algorithm is to predict a class given a new power consumption instance. Let us assume, instance x is member of $\{x_N\}$ and class label i is member of $\{I_k\}$. Then the classifier is any function $f : \{x_N\} \rightarrow \{I_k\}$

In kNN, the training phase simply stores the training data along with their class (instruction) labels. During classification, the classifier computes the distance between the unlabelled power consumption waveform W and all training traces $x \in \{x_N\}$. Then it keeps the k' closest training traces, where $k' \geq 1$. The class that is most common among these traces is assigned to the unlabelled trace W. In kNN there are two major design choices to be made; (a) the value of k', for instance, if only two classes exist $k' = 3$ is used to avoid ties, and (b) the distance function to use. The most common distance function used in kNN is the *Euclidean distance function* [173, 174]. Given a training trace x and unlabelled trace W the Euclidean distance, d_e is computed as shown in equation (6.5).

$$d_e(x, W) = \|x - W\| = \sqrt{(x_1 - W_1)^2 + \cdots + (x_p - W_p)^2} = \sqrt{\sum_{i=1}^{p} (x_i - W_i)^2} \tag{6.5}$$

where both x and W have p sample points and x_i and W_i are the ith point for $i \leq p$. Apart from the Euclidean distance function, some of the other distance functions that can be used in kNN are the *Correlation* [175] and the *Cosine* learning distance functions.

6.5 RSA Signature Screening Algorithm

Digital signature algorithms are used to verify the authenticity and integrity of a block of message. RSA is one of the popular digital signature algorithms [85]. The verification of n RSA signatures involve the verification of n signatures sequentially. In a hash-and-sign scheme the process of verifying n signatures involves the generation of n hash values and n public key encryptions with respect

to the issuer's public key. Now this could be very time consuming. A known method of improving the performance of such a system is to verify a batch of signatures at the same time. In [176], the authors discuss a method that verifies if a batch of messages were signed by the correct authority without verifying the individual signatures. This process is called the *RSA signature screening*. The *RSA signature screening* works as follows: given a batch of message and signature pairs

$$\{\{M_1, S_1\}.....\{M_n, S_n\}\},$$

where S_i (computed as $S_i = M_i^d \ mod \ N$) is the signature of a message M_i with respect to some private key (N, d). We assume that the signatures were generated using the hash-and-sign scheme, then this batch of signatures is verified using the computation in the equation (6.6) with respect to the corresponding public key (N, e).

$$(\ \Pi_{i=1}^{n} S_i\ )^e \ = \ \Pi_{i=1}^{n}\ H\left(M_i\right)\ mod\ N \tag{6.6}$$

However, as discussed in [177], *RSA signature screening* can be bypassed if a message M_i appears more than $e - 1$ times even though it was never signed before. This can be an issue if the value of e is significantly small. This problem can be easily solved by choosing a large value of the public key component e.

6.6 Basic Block Integrity Verification

Our software integrity verification method uses the notion of basic blocks. A basic block is a group of instructions executed sequentially by the processor. A basic block has only one entry point (the first instruction executed) and one exit point (the last instruction executed) [147]. A basic block may have many predecessors and many successors. It might also be its own successor. Program entry basic blocks might not have predecessors that are within the program and

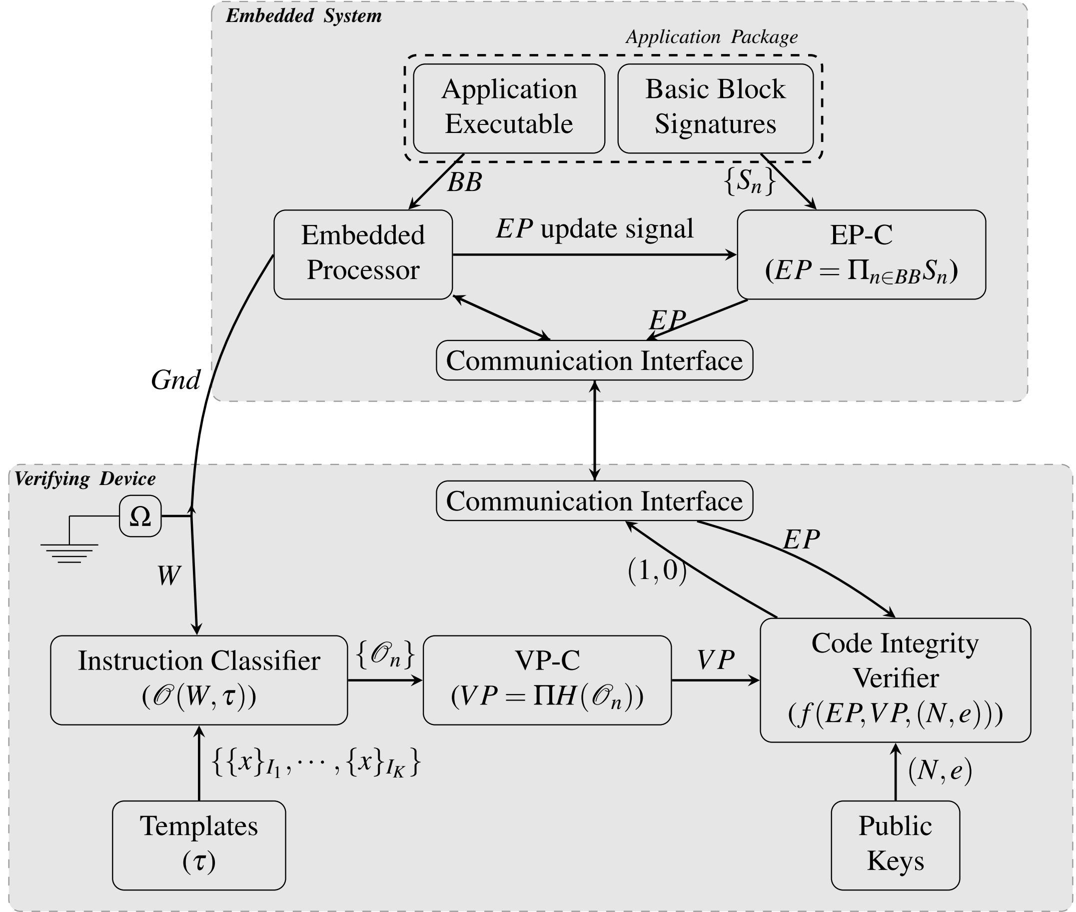


Figure 6.1: Basic block integrity verification block diagram

program ending basic blocks never have successors within the program itself. After executing one basic block the processor jumps into another basic block based on the branching instruction executed at the end of the current basic block. This branching instruction can be conditional or unconditional.

During development, the application is divided into basic blocks and that each basic block is signed with the developers RSA private key, then these basic blocks together with their signatures are installed in the processors non-volatile memory. These instruction-level templates together with the basic blocks signatures are used to verify the integrity of the software using the RSA signature screening algorithm. Figure 6.1, elaborates the block diagram of our proposed software integrity verification method.

As shown in the diagram (Figure 6.1), the embedded system has the embedded parameter calculator (EP-C), embedded processor and the application package which includes the application executable and the basic block signatures. The EP-C is a special module that calculates the product of two large numbers. This module can be implemented in hardware or software; although, hardware would be preferable for performance reasons. The embedded processor is the core CPU that executes the software component of the embedded system.

The verifying device has the instruction-level templates, the instruction classifier, the verifier parameter calculator (VP-C) and the software integrity verifier. The templates are constructed ahead of time using identical processors and then installed into the verifying devices non-volatile memory. The instruction classifier uses these templates to extract the executed instructions from the processors power consumption waveform (W). The power consumption waveform is measured as a voltage drop across a shunt resistor connecting the embedded systems ground and the verifying devices ground voltage. The VP-C uses the output of the classifier to compute the verifying devices parameter (VP). Finally, the software integrity verifier uses the output of the EP-C and VP-C to verify the integrity of the software using RSA signature screening algorithm.

When the software execution starts, both the EP-C and VP-C initialise their parameters to the value "1". As the execution commences both modules update their parameters after the execution of each basic block. Given $S = \{S_1, S_2, \cdots, S_n\}$, a collection of basic block signatures with S_n being the signature of the nth basic block. The EP-C updates its parameter (EP) by multiplying it with the basic blocks signature. At the end of the execution EP looks like equation 6.7.

$$EP = \Pi_{i \in BB} S_i \tag{6.7}$$

where BB is list of executed basic blocks. At the same time the verifying device records the power consumption wave form and extract executed instructions using the function $\mathcal{O}(\mathcal{W}, \tau)$. This function takes the power consumption, W, and the instruction-level templates, τ, and generates a list of executed instructions as

shown in equation 6.8.

$$\mathcal{O}(W, \tau) = \{Ins_1, Ins_2, \cdots, Ins_n\} \tag{6.8}$$

Let $\mathcal{O}_n$ be the output of $\mathcal{O}(\mathcal{W}, \tau)$ for the nth basic block. The VP-C updates its parameter, VP, by multiplying it with the hash of $\mathcal{O}_n$. At the end of the execution VP will look like equation 6.9

$$VP = \Pi_{n \in BB} H(\mathcal{O}_n) \tag{6.9}$$

Once the execution of the program finishes the *Code Integrity Verifier* verifies the integrity of the executed part of the software using both parameters EP and VP as follows.

$$f(EP,\ VP,\ (N, e)) = \begin{cases} 1, & \text{if } (EP)^e = VP\ mod\ N \\ 0, & \text{otherwise} \end{cases} \tag{6.10}$$

If the result of f, equation (6.10), is “1”, the integrity of the executed part of the software is executed is still intact. Otherwise, it is regarded as compromised (modified by an unauthorised entity).

6.7 Implementation and Results

To implement the techniques discussed above we have selected an *ATMega163 + 24C256* based smart card. The ATMega163 is an 8-bit AVR[2] microcontroller, and it has 130 instructions. To simplify our experiment we chose 39 instructions. The selected instructions are explained in detail in Appendix A. During the instruction selection process we considered the following criteria; redundancy and usage of

[2] AVR microcontrollers are Atmel family ICs. `www.atmel.com`

instructions. The redundancy refers to more than one instruction performing the same operation; for example in ATMega163 the instructions `LD` R_d`, Z` and `LDD` R_d`, Z+q` perform indirect load operation. So, in our experiment we only use `LD`. Besides the redundancy, we also tried to choose the most commonly used instructions by analyzing several source codes. We created a source code base by using publicly available source codes from various web sites [178, 179]. These websites host open source projects like AES, DES, analogue synthesizer, general purpose libraries, etc. We have also included our own implementation of cryptographic algorithms and general purpose applications in the analysis. The selected instructions are listed in Appendix A.

The power traces are captured via a voltage drop across a shunt resistor connecting the ground pin of the smart card and the ground pin of the voltage source. The smart card is running at a clock frequency of 4MHz and is powered by a +5V supply from the reader. The measurements are recorded using a *LeCroy WaveRunner 6100A* [180] oscilloscope capable of measuring traces at a rate of 5 billion samples per second (5GS/s). The samples have 8-bit accuracy within a pre-selected range. The shunt resistor is connected with the oscilloscope using a special cable, a *probe*, which was a *Pomona 6069A* [181], a 1.2m co-axial cable with a 250MHz bandwidth, 10MΩ input resistance and 10pf input capacitance. All measurements are sampled at a rate of 500 MS/s. The same measurement setup is used throughout the experiment.

6.7.1 Instruction-Level Template Construction

To generate the number of traces we needed for the templates construction we created several training code snippets. To construct the templates we attempted to remove all other factors that influence the power consumption apart from the instructions themselves. Such factors can be the initial values of source and destination registers/memory cells, data processed by the instruction and, intrinsic or ambient noise introduced by the measurement setup. To remove the influence of the source and destination registers/memory cells we selected a random source and destination before we executed the selected instructions

and we initialised them with random values sent from the terminal over the Application Protocol Data Unit (APDU) channel. For the data processed, we have generated random data for each execution of the target instruction. To minimise the influence of the ambient noise introduced in the measurement, all equipment is properly warmed up beforehand so that it is all running at a uniform temperature throughout the power trace collection phase. This requires running a few test measurements to be discarded before the actual power trace collection begins.

To minimise the effect of measurement noise introduced by the reference card on the power traces we used 5 of the same model reference cards throughout the experiment. To reduce the influence of other random noise from our measurement we collected 3000 traces for each of the selected instructions (i.e. 600 traces from each of the reference cards). Out of these 3000 traces, we used 2500 of them to construct the templates. As part of the templates we took the average of recorded traces and this reduces the standard deviation of the random noise by a factor of $\sqrt{n}$, given that n is the number of traces used for calculating the averaged value.

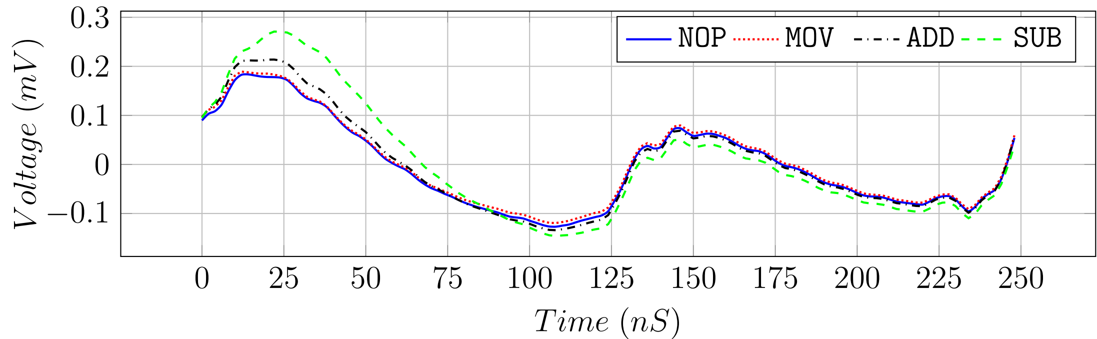


Figure 6.2: Power consumption waveform of selected ATMega163's one clock cycle instructions (`NOP`, `MOV`, `ADD` and `SUB`).

For multiple clock cycle instructions, the clock cycles are treated as consecutive instructions. Hence, more than one template is created for them. For the conditional branching instructions, templates are created for both conditions. When the condition is false the branching instructions only need one clock cycle; however, when it is true they need two clock cycles. Therefore, for each conditional branching instruction we created three templates. Including the mul-

tiple templates for the multi-clock cycle instructions and conditional branching instructions we generated a total of 76 templates. In Figure 6.2 and Figure 6.3 we plot the average of the power consumption waveforms generated by one and two clock cycle instructions respectively for selected instructions.

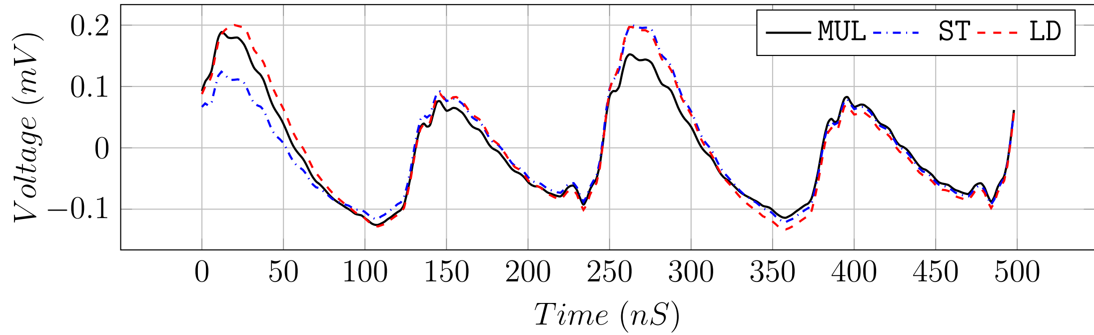


Figure 6.3: Power consumption waveform of selected ATMega163's two clock cycle instructions (`MUL`, `ST` and `LD`).

As shown in both the plots, some instructions (for instance `NOP` and `SUB`) generate sufficiently different waveforms to recognise them successfully. However, others (for instance `NOP` and `MOV`) generate similar waveforms which makes it more difficult to recognise them from their power waveform. So, in order to recognise each instruction from a given waveform we have to create a well-conditioned template and for that we need several training traces.

6.7.2 Dimensionality Reduction

When using the *Sum of Difference of Means* to reduce the dimensionality we computed 2850 vector subtractions and additions. Fig. 6.4 illustrates the summation of these differences. The Means-Variance is a straight forward method and involves the computation of variance for 125 column vectors.

When using PCA, the new dimensionality D has to be chosen carefully. On the one hand, if D is too small, too much of variance of the original data may get lost and with it important information about the observations. On the other hand, if D is too large, the templates cross-correlation increases and the classification be-

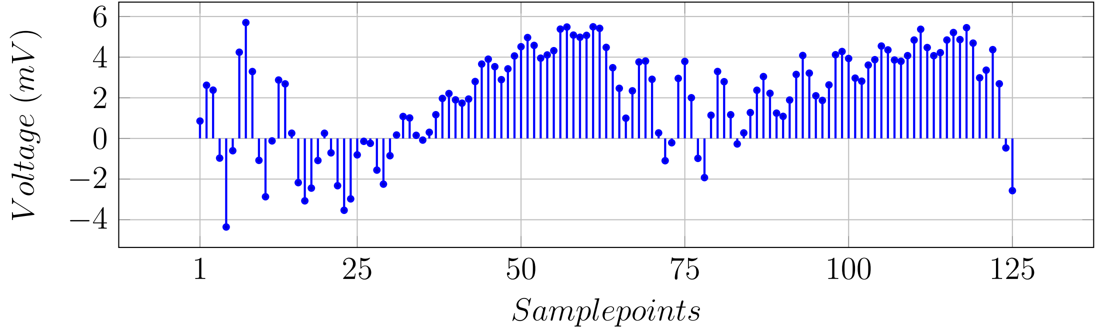


Figure 6.4: Sum of difference of means.

comes less reliable. In Fig. 6.5, we have plotted the amount of variance accounted for each principal component of instructions NOP, MOV, CLR and ADD.

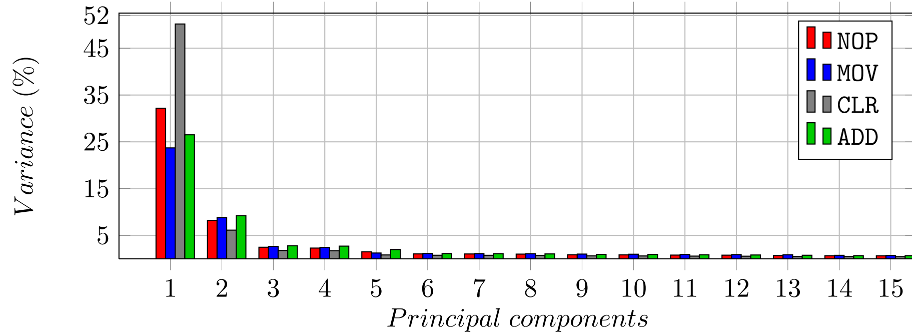


Figure 6.5: Overall variance of the original data accounted for the first 15 principal components of the instructions NOP, MOV, CLR and ADD.

As shown in Figure 6.5, for the instruction MOV, the first 4 components accounted for 37.598%, the first 10 for 44.163% and the first 15 for 48.3387% of the overall variance of the original data. For the instruction CLR 59.796%, 64.089% and 66.648% of the original variance is accounted for the first 4, 10 and 15 components respectively. So, when choosing the dimensionality, D, we have to decide how much variance of the original data that we are willing to lose. In addition to PCA, we also performed Means-PCA on the class-means instead of on the class observations. Like the other techniques we also reduced the dimensionality of the original data into 50 using LDA. In the next section, we discuss how our classification algorithms perform on the unseen 500 traces for all dimensions $1 \leq D \leq 50$.

6.7.3 Instruction Classification

So far, we have selected 39 instructions out of the possible 130 and collected 3000 power consumption traces for each of the instructions. Out of these 3000 traces we used 2500 of them to train the templates. Now we discuss the classification result for the remaining 500 traces.

Multivariate Gaussian Probability Distribution Function (MVGPDF): We have tested the MVGPDF classification both before and after the dimensionality reduction. Before reduction, we utilised the full space of the original data, the overall recognition rate was 64.97%. In Table 6.1, we present the recognition rate of 11 selected instructions using the full data space.

Table 6.1: Percentage of true (**bold**) and false positive recognition rate for a selected instructions using MVGPDF. The rows and columns represent executed and recognised instructions respectively.

Instruction	**Recognised as [%]**										
	NOP	MOV	ADD	ADC	MUL_1	MUL_2	CLR	CP	INC	SUB	SBC
NOP	**28.7**	0	2.8	5.2	0.8	14.3	0.2	10	1.2	0	0.6
MOV	0	**49.2**	5.2	0	0.4	0	3.2	0	10.2	0	0.6
ADD	9	4.6	**17.5**	0	0.4	0.6	0	0.2	7.2	0.6	0.2
ADC	0.6	0	0	**91.6**	0	1.6	0	5.2	0	0	0.2
MUL_1	2.6	0.4	1.2	0	**68.7**	0	9.2	0.8	0	0	0.6
MUL_2	20.7	0.8	1	1	0	**41.6**	0	0	9.4	0	0.2
CLR	4.8	2	0.6	0	7.2	0.6	**80.1**	0	1	0	1.2
CP	2.2	0	0.6	3.8	0	0	0	**89.8**	0	0	1.2
INC	7.8	4.8	7.8	0.2	0.2	0	0.4	0	**42**	0.2	0.6
SUB	0	0	0	0	0	0	0	0	0	**86.1**	0
SBC	0.6	0	0.8	3.6	1.2	0	1.6	3.6	0.2	0	**86.3**

However, this is costly in terms of computational overhead and is time consuming. So, to find a good subspace for our dimensionality reduction, we tested MVGPDF for the first 50 dimensions of the original data.

In Figure 6.6 we plotted the result of our classification rate after the dimensionality reduction techniques. In the graph, the first number within the bracket is the dimension and the second number is the maximum classification rate. Using MVGPDF, the maximum classification rate we could achieve was 66.78% after

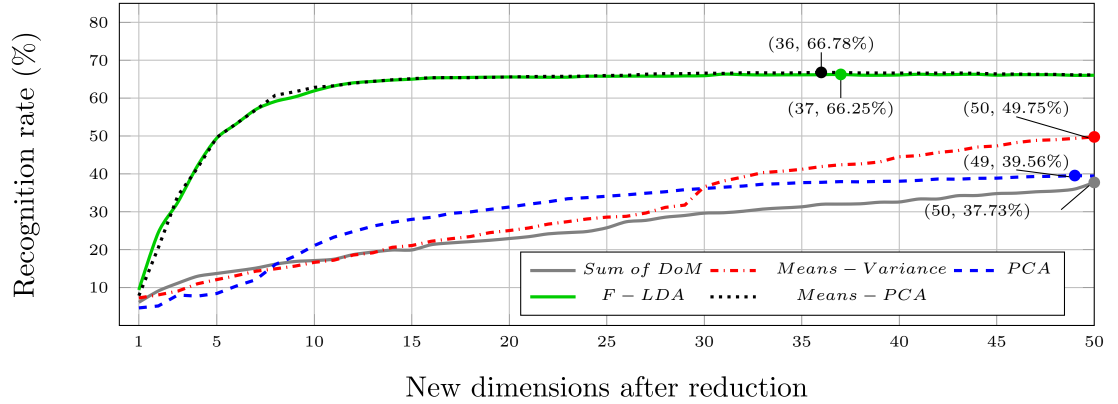


Figure 6.6: Classification rate after dimensionality reduction using *Multivariate Gaussian Probability Density Function* for all 39 instructions.

using Means-PCA for reducing the dimensions.

k-Nearest Neighbors Algorithm (kNN): In kNN there are two major design decisions that need to be made. One is the number of neighbors, k, participating in the decision making. The other is the distance function used to compute the closeness between the template data and the signal that need to be classified. First, we tested our traces with $k = 1$, Euclidean distance function and full dimension of the traces. The average recognition rate for all the templates is 45.31%. The recognition rate for a selected 11 instructions is presented in Table 6.2.

k-Fold Cross Validation k-fold cross validation is a common method used to estimate how a classifier performs over a given data [182]. Given a set of m collected samples; k-fold cross validation proceedes as follows;

1. Divide the data into k-equally sized folds,
2. For i=1...k
   - Train the classifier on all samples that do not belong to i,
   - Test the classifier using all the samples that belong to i

Table 6.2: Percentage of true (**bold**) and false positive recognition rate for a selected instructions using kNN. The rows and columns represent executed and recognised instructions respectively.

Instruction	Recognised as [%]										
	NOP	MOV	ADD	ADC	MUL_1	MUL_2	CLR	CP	INC	SUB	SBC
NOP	**25.9**	0.2	2.8	4.6	2.0	13.9	1.8	6.0	2.2	0	1.0
MOV	1.6	**31.1**	4.4	0	0.8	0.2	7.6	0	5.6	0	0.4
ADD	6.6	3.8	**10.2**	0.4	1.0	1.4	1.6	0.6	7.6	0.2	0.4
ADC	7.6	0	0.2	**43.4**	0.4	15.3	2.0	18.9	0.6	0	0.8
MUL_1	5.6	1.2	1.6	0.2	**33.5**	1.6	1.6	0.8	0.6	0	0
MUL_2	27.5	1.6	1.6	2.2	0.4	**51.2**	1.2	0	3.4	0	0
CLR	3.6	3.2	12.5	0.2	6.4	1.4	**36.4**	0	9.0	0	1.0
CP	10.6	0	5.0	7.0	1.0	0.6	0.4	**27.7**	0.4	0	3.8
INC	9.0	4.4	8.6	0	0.6	0.4	4.0	0.8	**11.3**	0	1.2
SUB	0	0	1.6	0	0	0	0	0	0	**89.8**	0
SBC	4.2	0.4	9.4	9.8	5.2	2.0	3.2	7.2	2.0	0.2	**23.3**

For the sake of completeness we tested our classifier (the kNN algorithm) using a k-fold cross validation, where k=4. In this test our data is divided into 4 equally sized folds and used for training and tesing the algorithm. This process is depicted as shown in Figure 6.7.

The average performance of our classifier with each fold used for testing is presented in the Tables 6.3.

Table 6.3: Average percentage of true (**bold**) and false positive recognition rate for a selected instructions using kNN in k-fold cross validation.

Instruction	Recognised as [%]									
	MOV	ADD	ADC	MUL_1	MUL_2	CLR	CP	INC	SUB	SBC
MOV	**31.1**	14.5	9.7	0	0	0	10.4	3.1	2.4	5.9
ADD	13.9	**32.4**	6.6	0	0	0.1	9.4	3.2	2.9	6.9
ADC	9.3	7.2	**28.6**	0	0	0.4	7.8	15.9	8.1	8.3
MUL_1	0	0	0	**58.2**	0	0	0	0	0	0
MUL_2	0	0	0	0	**22.4**	0	0	0	0	0
CLR	0	0	0.1	0	0	**53.5**	0	1.4	6.6	3.4
CP	9.8	10.3	6.5	0	0	0	**33.2**	1.5	0	1.6
INC	5.1	4.3	18.8	0	0	2.1	1.0	**35.3**	11.2	12.7
SUB	3	2.8	6.5	0	0	6.9	0	10.9	**33.7**	18.7
SBC	5	5.4	9.2	0	0	2.5	1.5	10.4	21.2	**31.3**

To improve the recognition rate we used dimension reduction techniques. With $k = 1$ and Euclidean distance function we repeated the experiment on a reduced

K=1	Test	Training	Training	Training
K=2	Training	Test	Training	Training
K=3	Training	Training	Test	Training
K=4	Training	Training	Training	Test

Figure 6.7:

dimensions and the recognition rate is presented in Fig. 6.8. The result for LDA, Means-PCA, Sum of Difference of Means and Means-Variance was not very satisfactory. However, for PCA we have achieved a 100% recognition after only using the first 13 dimensions. In order to see the effect of changing k on the recognition rate, we repeated the experiment for $k = \{5, 10, 15, 20\}$ and the result was the same. Now this steep increase in recognition rate could be a combined result of removal of inter-class correlated points using PCA and the fact that the traces are not generalised during the learning process of the algorithm.

To check the effect of the second criterion, the distance function, we tested our traces using three different distance functions. These are the *Euclidean*, *Correlation* and *Cosine* distance function. The classification result after PCA using all three different distance functions is plotted in Fig. 6.9. As shown in the graph, apart from a minor difference for dimensions $1 \leq D \leq 12$, the recognition rates are the same. They all reached a 100% of recognition rate after the first $D \geq 13$ dimensions.

Finally, it may be worth noting that apart from the two classification techniques, we have also experimented with several others. These algorithms include Self-Organizing Maps [183], Support Vector Machines [184], Linear Vector Quanti-

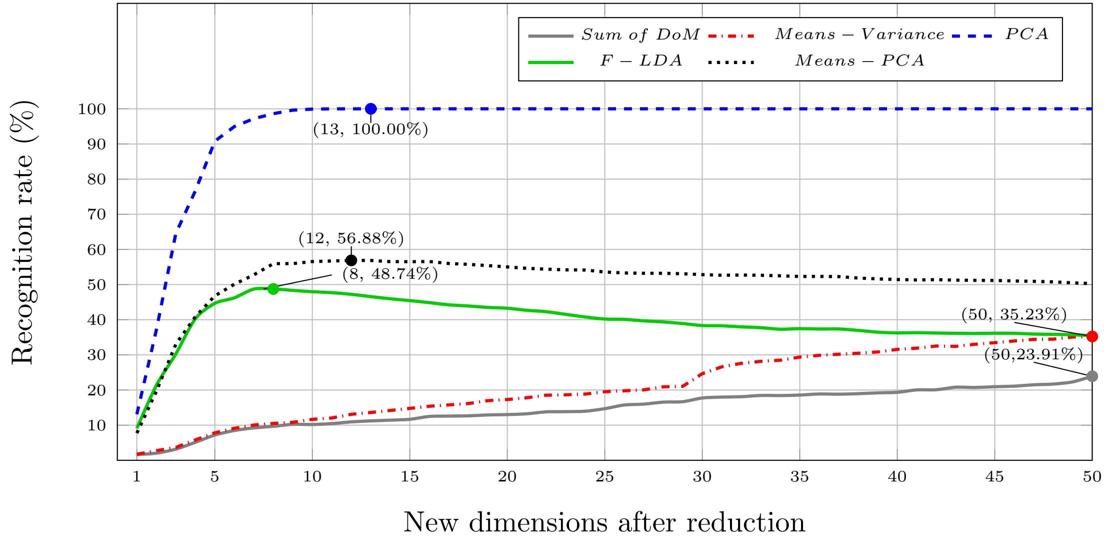


Figure 6.8: Instruction recognition rate for all 39 instructions using *K-Nearest Neighbors Algorithm* for k=1 after applying the dimensionality reduction techniques.

zation [185] and Naive Bayes Classifiers [186]. However, their results were not satisfactory and we stopped pursuing them.

6.7.4 Basic Block Integrity Verification

To test the software integrity verification technique, we generated a pair of RSA keys. Normally it is recommended to use large prime numbers in order to be secure against factorization attacks. However, our aim here is to show that power consumption can be used to verify the integrity of embedded software. Therefore, we generated the key pairs using small prime numbers. We selected the prime numbers to be $q = 23$ and $p = 59$. Using CrypTool [187] we generated the public and private key to be $(N = 1357, e = 3)$ and $(N = 1357, d = 851)$ respectively. For this experiment we also implemented an application that verifies a four digit PIN value. The reference PIN is stored in the non-volatile memory of the processor and the PIN that needs to be verified is sent from a terminal (PC). Once the processor receives the PIN, it compares it with the reference PIN digit by digit. The source code is available at [35]

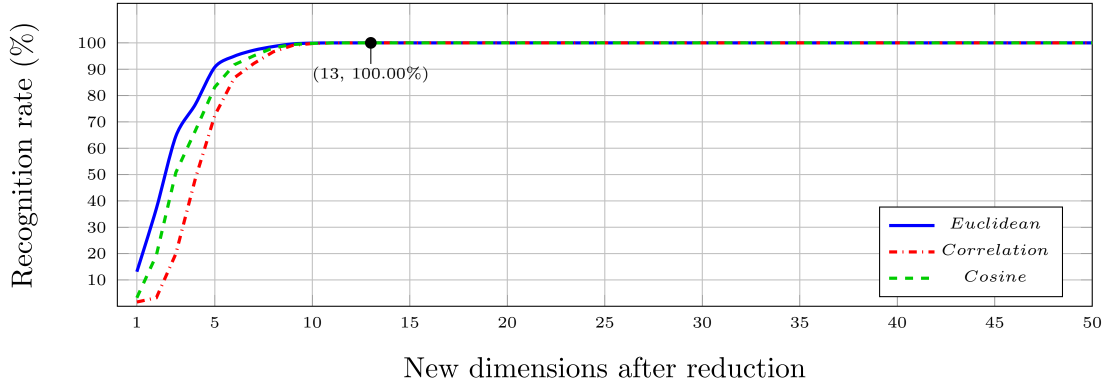


Figure 6.9: Recognition rate for all 39 instructions using *K-Nearest Neighbours Algorithm* with different distance functions for k=1 after applying PCA.

Before signing the basic blocks of our application we generated a hash (fixed) value on the immutable part of the basic block instructions. For example, in *ATMega163* instructions have two parts the *Opcode* and the *Operand*. The *Opcode* is always static and the *Operand* depends on the arguments (parameters). In the RSA hash-and-sign scheme, standard hash algorithms such as SHA-1 [188] and MD5 [189] are used to generate the hash value. However, since our experiment was not about attacking hash algorithms we used a simple XOR of the immutable parts for simplicity.

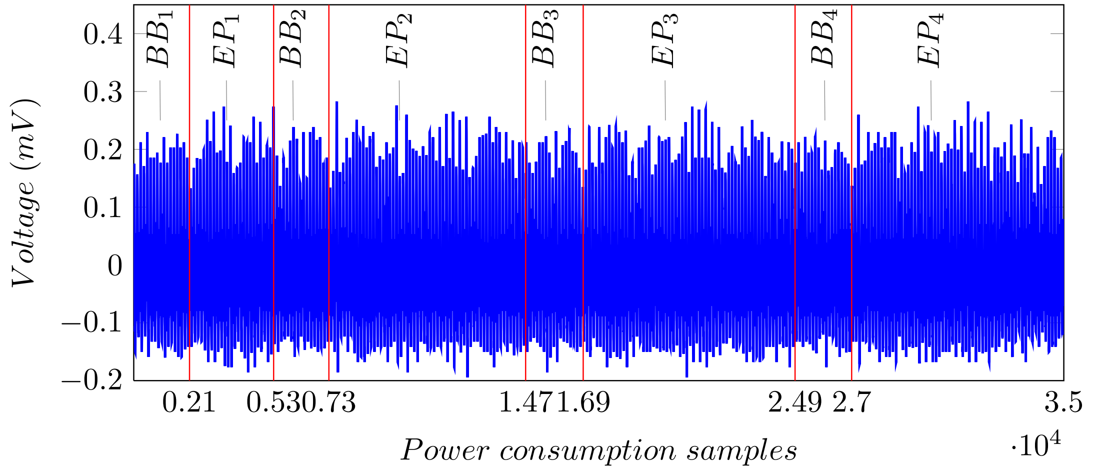


Figure 6.10: Power consumption of the processor when executing PIN checking application with embedded processor parameter update in between the basic blocks.

In Figure 6.10, the plot sections labelled as "BB" are when the processor executes the basic blocks and the sections labelled as "EP" belong to the processor's parameter update operation. To test our technique, we changed some of the instructions of the application after the signatures were generated and ran it again. In our first trial we replaced two consecutive MOV instructions with MOVW. They both accomplish the same task, but generate different waveforms. Secondly we changed the compare instruction CP to CPC in the first two basic blocks. The PIN still gets verified correctly, but the waveform was not quite the same and we detected that using our proposed method. Finally, we replaced the branching statement BRNE with BREQ and ran it. As expected equation (6.10) returned "0", which means the integrity of the application is violated, for all three cases. We have also implemented the same function in PIC16F876 [190] microcontroller and run the verification process using the templates built for ATMega163. Again as we expected it, the instruction classification function did not produce the correct instructions. As a result the verification function returned "0".

6.8 Summary

This chapter has explored the unconventional idea of permitting side channel leakage from an electronic component, before it is deployed in real-time operation, for the purposes of useful analysis and application code integrity verification. This approach can be used in to detect if there is a design anomaly in the hardware platform as it will be reflected in the side channel leakage. We conducted an experiment on an AVR architecture microcontroller, *ATMega163*. In our experiment we achieved a 100% of classification rate using k Nearest Neighbours algorithm for executed instructions. As the verification process is only performed once after acquiring the components (but before deploying them in operation) it will not affect the performance of the chip once used in operation. Furthermore, special equipment is not required to support the components in real-time operation. In this chapter we discussed our proposal in the context of verifying an embedded device before used in operation. However, the technique can

also be used for forensic analysis of electronic components, application integrity verification and counterfeit components detection purposes.

Part II

Runtime Secure Execution

Chapter 7

Program Data Security

Contents

7.1 Introduction **119**

7.2 Related Work **121**

7.3 Attack Model **122**

7.4 Dual-Stack Architecture **123**

7.5 Implementation and Analysis **127**

7.6 Summary **132**

In this chapter, we propose a dual-stack processor architecture to protect runtime data processed by the executable instructions. We start by explaining a simple attack on runtime data, specifically the stack items, and preciously proposed countermeasures. We then define an attacker's capabilities and attack types that an attacker can mount. Then we discuss two variants of a dual-stack processor architecture. In addition, we provide detailed analysis of implementation options, performance overhead and latency of these variants. Furthermore, we provide a comparative analysis with other countermeasures. Finally, we summarise the chapter by mentioning the main points.

7.1 Introduction

During the execution of a program a processor creates a number of temporary runtime data to facilitate the execution. This includes intermediate computational results, function call parameters, return addresses and conditional branching statement parameters. Such data is usually stored in the stack [191]. A stack is a Last In First Out (LIFO) data structure with two principal operations; the `push` and the `pop`. The LIFO serves as a collection of runtime data elements. The `push` adds a new element to the collection and the `pop` removes the last element that was added from the collection. The stack contains valuable information that can be targeted by an adversary to compromise the integrity of a program execution. The result of such an attack can be; (1) extract secret information, (2) divert the execution into a maliciously crafted code segment, (3) skip certain parts of the program from executing or (4) simply corrupt the final computational result.

Processors use a shared stack for both intermediate data and function return addresses. In such a set-up an adversary may write an oversized data into the stack to overrun the stack boundary and overwrite adjacent memory cells in a program that does not check data size. This exploit is also known as stack overflow [192]. One extreme form of stack overflow is the stack smashing [193], where the adversary overflows the stack to change the return addresses. The adversary may experiment on a specific data type and size that will modify the return address in such a way that it will point to a location within the stack itself, which contains executable instructions.

Another, relatively new, attack on embedded system's stack is the *Fault Injection Attack*, where the adversary intentionally generates malfunction by inducing faults, either to exploit the faulty output to extract some secret information or simply injects the fault into the return addresses to divert the program execution. Such faults can be induced by using clock glitches, power source spikes [194], heat or laser generators [195, 196, 197]. The first use of fault injection to extract a secret cryptographic key was presented by Bellcore research team in [95]. In their work they showed how a single fault can be used to efficiently factor

the RSA modulus of Chinese Remainder Theorem (CRT) based implementation with high probability. Later, Biham et al. introduced the concept of Differential Fault Analysis (DFA) [96] on the Data Encryption Standard (DES) [198]. This led to a subsequent publications of similar attacks [97, 98, 99, 100, 101] on the Advanced Encryption Standard [199].

For the sake of completeness we describe a simple fault injection attack example on stored stack items to divert the execution of a target program. This example attack is on Java stack, however, it may be worth mentioning that our proposal is not specific to Java stack items. Figure 7.1 illustrates the Java source code and bytecode of the target program, which is a wallet applet.

```
package Debit;

import java.lang.*;

public class Debit{
  private int balance = 100;
  private String refPIN="1234";

  void Deposit(int value){
    balance = balance + value;
  }
  void Withdraw(int value){
    balance = balance - value;
  }
  boolean pinCheck(String pin){
    if(pin == refPIN){
      return true;
    }
    return false;
  }

  void main(){
    String pinTBC="3456";
    if(pinCheck(pinTBC)){
      Withdraw(20);
    }
  }
}
```

```
1  boolean pinCheck(java.lang.String);
2    Code:
3      0: aload_1
4      1: aload_0
5      2: getfield  #4; //Field refPIN:Ljava/
          lang/String;
6      5: if_acmpne 10
7      8: iconst_1
8      9: ireturn
9      10:  iconst_0
10     11:  ireturn
11
12 void main();
13   Code:
14     0: ldc #5; //String 3456
15     2: astore_1
16     3: aload_0
17     4: aload_1
18     5: invokevirtual #6; //Method pinCheck
          :(Ljava/lang/String;)Z
19     8: ifeq  17
20     11:  aload_0
21     12:  bipush  20
22     14:  invokevirtual #7; //Method
          Withdraw:(I)V
23     17:  return
24
25 }
```

Figure 7.1: Fault Injection: Attack example on stack items.

The `invokevirtual` (at line 5 of the main method) invokes the *pinCheck* function. This diverts the execution to line '0' of the *pinCheck* function (which is the `aload_1`). Since the provided and reference PINs are different the *pinCheck* returns 'false' by executing `return false` instruction. This is done by pushing the value '0' into the stack as shown in line code 10 (`iconst_0`). However, if an adversary can change the top of the stack into anything other than '0' then the

wrong PIN will be accepted as valid.

Recently fault injection attacks are also being used in combination with other attacks to maximise their impact. These attacks are known as *Combined Attacks*. In *Combined Attacks* fault injection on runtime data are used to facilitate other powerful attacks. Such attacks are discussed in detail in [200, 136, 201, 202, 203]. Given the high impact and sophistication of these attacks several countermeasures have been proposed. In this chapter we propose a new countermeasure against attacks that target stack items.

The rest of the chapter is organised as follows. Section 7.2 discusses countermeasures proposed to detect changes to stack items by an adversary. Section 7.3 defines attack types and attacker capabilities with regard to our proposal. Section 7.4 discusses our proposed countermeasure in detail. We explain our techniques using two working variants of our countermeasure. Section 7.5 provides our implementation and analysis. We also provide details of both hardware and software implementation of our technique. In addition, we elaborate in detail about performance overhead, detection latency and fault detection capability of our countermeasure. Finally, section 7.6 summarises the chapter.

7.2 Related Work

One the earliest countermeasures proposed to protect stack items is *stack canaries* [144]. Stack canaries are originally proposed to stop stack overflow attacks. Stack canaries are values that are placed between a buffer and control data in the stack to monitor buffer overflows. When the buffer overflows, the first data that get corrupted will be the canary value. A failed verification of the canary value is therefore an alert of an overflow, which can then be handled, for example, by invalidating the corrupted data. To stop attacks against predictable canary values, they are generated randomly and never revealed to the outside world.

As discussed in Section 7.1, stack smashing is an extreme form of buffer overflow. Stack smashing is the result of a processor's design decisions; (1) to let the proces-

sor store user data and program control data in the same stack (memory block), and (2) the processor being able to execute instructions from the stack area. The countermeasures against stack smashing directly address these two points. The Instruction Based Memory Access Control (IBMAC) proposed in [142], splits the stack into a data stack and function return address stack through a hardware flag. The function return address stack is only used for storing subroutine function call return addresses and can only be accessed via instructions such as `call` and `ret`. A non-executable stack enforces a memory policy on the stack memory region that disallows execution from the stack [204]. Both the IBMAC and the non-executable stack directly address the design decisions discussed above.

Another countermeasure is proposed by Barbu *et al.* [200]. In their work they present countermeasures against fault injection attacks on Java Card operand stack. In their work they explained three methods that can detect faults injected into the operand stack. The first is, *redundant checks*, which checks the value expected to be pushed/popped with the value that was actually pushed or popped. Both the repeated `push` and `pop` operations are performed on the same stack. The second method, *fault propagation*, propagates a potential error into another component of the JCVM such as the `context`. Finally, *stack invariant*, involves the introduction of a variable that holds the XOR of values pushed onto and popped from the stack. These countermeasures only prevent fault injection attacks on the processor's bus not the actual memory content.

7.3 Attack Model

Before we dive into the discussion of the proposed mechanism, we first need to define the attack types and attacker's capabilities in the context of stack item manipulating attacks. These attacks could be physical (e.g. fault injection) or logical (e.g. stack overflow or stack smashing) attacks. The possible types of attacks that an attacker can mount and the capabilities he possesses are described below:

1. *Precise bit fault*: In this attack, the attacker has total control over the timing, location and value of the bit that he/she wants to change.

2. *Precise byte fault*: This scenario is similar to the previous attack; however, the attacker has the ability to change the value of a byte rather than a single bit (i.e. the attacker can change multiple bits within the target byte).

3. *Unknown byte fault*: The attacker has full control over the timing and location of the fault but no control over the value.

4. *Unknown fault*: In this scenario the attacker does not have any control over the timing, location and value of the fault.

Attacker Capability 1: *In this scenario the attacker is capable of successfully injecting a single fault into the stack during execution.*

Attacker Capability 2: *In this scenario a skilled attacker is capable of successfully injecting two separate (at different memory locations) but identical faults into the stack during execution. [205] .*

7.4 Dual-Stack Architecture

The integrity of a stack is critical in regulating how a program behaves during execution. It influences the control flow jumps through its stored function return addresses and conditional branching statement parameters. In addition, it also controls the final output of a function/algorithm via its stored intermediate computation results and function calling parameters. Therefore, ensuring the integrity of stack items is paramount to the security of the entire system. In our work we propose a new countermeasure against attacks that target stack items. Our countermeasure involves having two stacks. In other words it requires two separate RAM blocks dedicated for stack items. The processor will use the second stack to verify items stored in the first stack (*Operational Stack*) during runtime. Figure 7.2 illustrates the `push` and `pop` operations of our dual-stack architecture.

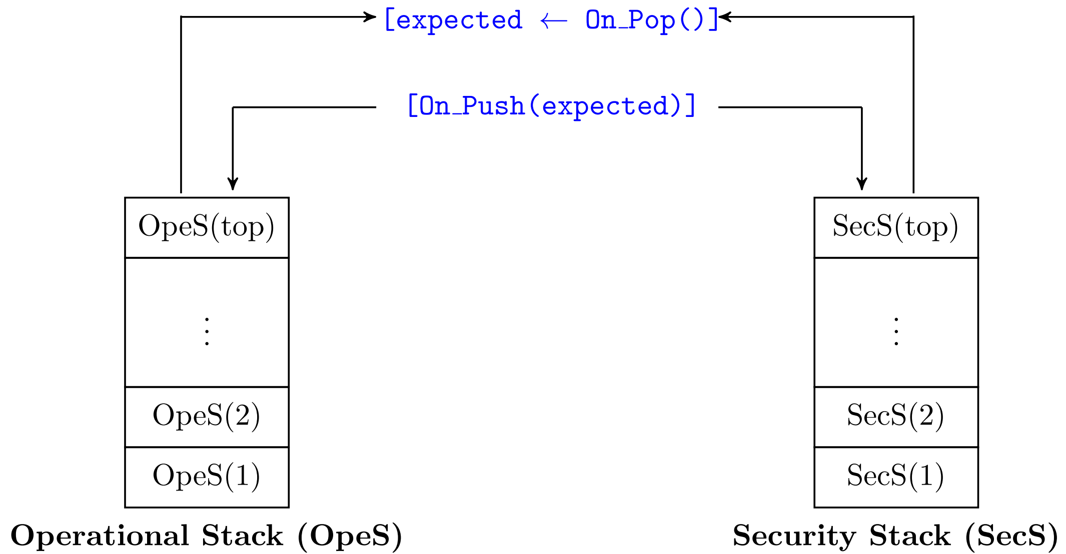


Figure 7.2: Double stack architecture

To ensure the integrity of operational stack items the processor can keep a direct copy of them somewhere else and perform redundant stack operations. This requires the processor pushing and popping the same value into and from both stacks. In a more sophisticated manner the processor may keep integrity values of all items pushed into the *Operational Stack* on the second stack, (*Security Stack*), and use this information to verify them during runtime. In the subsequent sections we will discuss two variants of the dual-stack architecture in detail.

7.4.1 TwinStack: Redundant Stack Operations

The most straightforward approach to check if a stack item is changed is to keep two copies of each stack item and perform redundant stack operations. Hereafter, we refer to this as a *TwinStack*. In *TwinStack* both stacks are identical (the same size and store the same data). The `push` and `pop` operations in *TwinStack* are illustrated in the pseudo code in Listing 7.1.

Listing 7.1: `push` and `pop` operations in TwinStack

```
//Executed when a value is pushed into the stack.
On_Push(expected){
```

```
        push(OpeS, expected);
        push(SecS, expected);
}
//Executed when a value is popped from the stack.
byte On_Pop(){
        expected = pop(OpeS);
        if(expected != pop(SecS)){
                ThrowFaultException();
        }
        return expected;
}
```

When the `On_Push` function is executed it pushes the same value (`expected`) into both the *Operational Stack* and the *Security Stack*. Then when the `On_Pop` function is executed it pops the top item from both stacks and compare them. The value is accepted as valid only if both values are equal. Otherwise, the processor executes the fault handler function (`ThrowFaultException()`).

To circumvent this protection the attacker must replicate the same fault on both stacks and at the same index. This requires a detailed knowledge on the memory organisation of the device and the program implementation. These requirements make a successful attack on *TwinStack* very difficult if not impossible. However, it is a truism that attacks only get better with time and we anticipate significant improvement to fault injection attacks with regard to replicating faults. We attempt to address this by proposing another variant of the dual-stack architecture, the *IntegrityStack*.

7.4.2 IntegrityStack: Verifying Integrity of Stack Items

The other technique that a processor could use to ensure the integrity of the stack items is to compute and keep their integrity matrix and then use it to verify popped items. Hereafter, we refer to it as a *IntegrityStack*. Every time the processor pushes a value it computes its integrity and pushes it into the `Security`

`Stack`. Then later when it pops it back, it uses this information to verifies its integrity. The `push` and `pop` operations of *IntegrityStack* is illustrated in the pseudo code in Listing 7.2.

Listing 7.2: `Push` and `pop` operations in IntegrityStack

```
//Executed when a value is pushed onto the stack.
On_Push(expected){
        push(OpeS, expected);
        push(SecS, (expected XOR SecS[top]));
}
//Executed when a value is popped from the stack.
byte On_Pop(){
        expected = pop(OpeS);
        if(expected != (pop(SecS) XOR SecS[top])){
                ThrowFaultException();
        }
        return expected;
}
```

When the `On_Push` function is executed the processor pushes the value (`expected`) into the *Operational Stack*. At the same time it XORs it with the top of the *Security Stack* and pushes the result into the *Security Stack* as an integrity information of the pushed item. Then when the `On_Pop` function is executed it pops the top item from *Security Stack*, XORs it with the new top and compares it with the item popped from the *Operational Stack*. The value is accepted as valid only if the comparison resulted equal. Otherwise, the processor executes the fault handler function (`ThrowFaultException()`). Unlike the *TwinStack* replicating the same fault will simply not work in *IntegrityStack*.

To circumvent the *IntegrityStack* the attacker needs to inject a fault into the operational stack and accordingly modify the corresponding integrity value on the security stack. To successfully modify the integrity values without being detected the attacker needs to know all the values pushed onto the operational stack before the erroneous item. This is because the integrity value at index i is computed as $SecS(i) = \sum_{ind=1}^{i} OpeS(ind)$. This makes it almost impossible to

launch a successful attack against *IntegrityStack*.

7.5 Implementation and Analysis

In this section we discuss implementation, performance overhead and detection latency analysis of the *TwinStack* and the *IntegrityStack*. Further, we also provide a comparative attack detection analysis of our countermeasure and other countermeasures proposed before.

7.5.1 Implementation

An embedded processor normally maintains one stack during runtime. Therefore, to use our countermeasure we either need to implement a separate second stack or change the way the processor utilizes its existing stack. The former can only be done through a hardware implementation. However, the later can be achieved by modifying the compiler and the processor's runtime environment. Hereafter, we refer to these two implementations as hardware and software implementations. Source code of our implementation is available at [35].

Hardware Implementation

In our hardware version, we implemented three stacks, a traditional single stack data structure along with both dual-stack architectures (*TwinStack* and *IntegrityStack*) that we discussed in this chapter. In our experiment we created a 256 byte SRAM memory block dedicated only for stack operations. Our single stack implementation uses only one memory block to store its items, whereas the dual-stack architectures use two separate 256 byte SRAM memory blocks. Figure 7.3, shows the conceptual block diagram of our implementation.

The two RAM blocks will be used to store stack items. In addition, we also

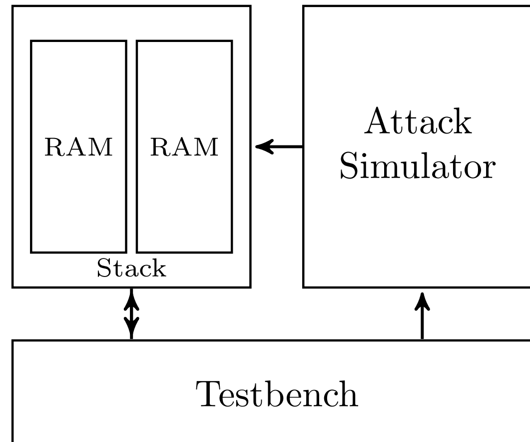


Figure 7.3: Hardware implementation of dual-stack and attack simulator

implemented an attack simulator that can simulate the attack types and capabilities that we described in Section 7.3. The "Testbench" simulates the external environment through which the programmer can push and pop values to/from stack and the attacker can perform his attacks. As per our definition of the attack model, the attacker can either pass the precise attack value and address to the attack simulator (*Precise Bit* and *Precise Byte*) or the simulator will randomly generate them (*Unknown Byte* and *Unknown Fault*). All modules are implemented in VHDL [1], compiled and simulated using GHDL [206]. Figure 7.4 presents the data capture of all input and output pins, and the internal RAM content of our traditional single stack implementation.

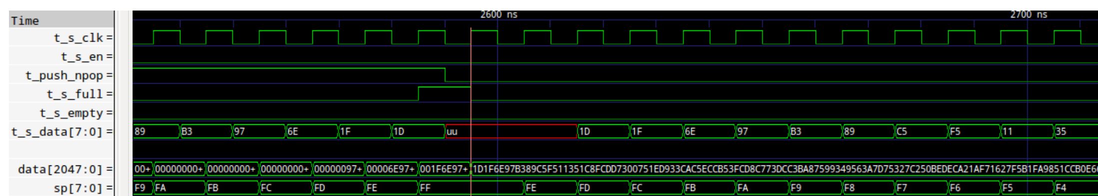


Figure 7.4: Normal single stack VHDL implementation

Our stack entity has six input and output ports. The *t_s_clk* is the clock signal input pin. The active low stack enable signal, *t_s_en*, enables read and write operations on the RAM. The *t_s_full* and *t_s_empty*, when '1', signals whether

[1] A hardware description language used in electronic design to describe combinational and sequential systems such as field-programmable gate arrays and integrated circuits.

the stack is full or empty respectively. The *t_s_data* is an inout [2] data port used to push and pop data into and from the stack. The *data* and *sp* are internal signals representing the SRAM memory content and the stack pointer [3].

In our dual-stack implementation we added one more output port the *s_faulty*. The *s_faulty* changes from '0' to '1' when an attack is detected on the popped item. Then all stack operations are frozen. As shown in Figure 7.5, a value "0x97" is pushed into both *Operational Stack* and *Security Stack*, which then later is changed to "0x33" through our attack simulator, which is shown by the blue highlight. This change is then detected by *TwinStack* (signified by the change of *s_faulty* from '0' to '1') and all stack operations are suspended.

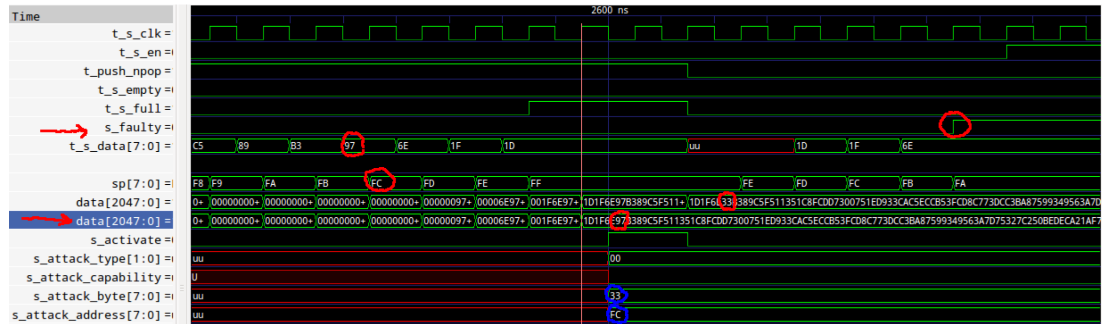


Figure 7.5: TwinStack VHDL implementation

Software Implementation

The software implementation of our proposal only uses the already existing stack. Incorporating our countermeasure will require changing the processor's way of utilizing the stack. For instance, alternating memory locations can be used for each stack that we proposed. Example, consider a stack with 256 byte entries on its allocated memory. The memory locations are addressed as `0x00 - 0xFF`. Even addresses (`0x00, 0x02, ..., 0xFE` can be used as *Operational Stack* and odd addresses `0x01, 0x03, ..., 0xFF` as *Security Stack*. This means for every item the processor has to do two pushes and pops. This requires modification to the compiler and runtime environment.

[2] A bi-directional port. Both writing into and reading from are allowed.
[3] A register that stores the address of the next location in the stack memory.

The stack is accessed in two ways; (1) the programmer pushes and pops values using the stack instructions, and (2) the processor pushes and pops function return addresses to keep track of control flow jumps. For the programmer's data the compiler can be modified in a way that it will rewrite (the push and pop instructions) to perform the necessary actions as discussed in this chapter. A countermeasure that uses similar technique (rewriting source code) is discussed in Section 4.2. However, to achieve a similar goal with the control flow information, the runtime environment needs to be modified. Full implementation of a modified compiler and runtime environment is out of scope of our research. Nevertheless, for the sake of completeness of our analysis, we have created an abstract processor that counts the number of affected instructions by our countermeasure and analyse the execution overhead incurred. Details of this analysis is discussed next.

7.5.2 Performance Overhead

The countermeasures involve performing additional tasks compared to a traditional single stack processor. For instance, in *TwinStack*, `push` stores the same value in both stacks. This is equivalent with 2 `push` instructions in a single stack processor. In *IntegrityStack*, `it` involves pushing the value into the Operational Stack, XORing it with the top of the Security Stack and pushing the result into the Security Stack. This is equivalent with 2 `push`, an XOR and a memory read in traditional processor. The `pop` in *TwinStack* includes reading the top of both stacks and comparing them, which makes it equivalent to 2 `pop` and compare instructions. However, in *IntegrityStack*, it is equivalent with 2 `pop`, memory read, XOR and compare instructions of a single stack processor.

The hardware implementation is done in a way that stack operations on both stacks are performed in parallel without incurring any additional overhead. However, in software implementation the additional instructions are executed sequentially and thus incur execution overhead. In Table 7.5.2 we presents the additional instructions executed by our countermeasure compared to a single stack processor in software implementation. In this table we discussed the effect of the

countermeasure on instructions that access the stack, such as `push`, `pop`, `call` and `ret`.

Affected Instructions	**Software Implementation**	
	TwinStack	**IntegrityStack**
`push`	1	3
`pop`	2	4
`call`	2	6
`ret`	4	8

Table 7.1: Number of additional instructions introduced by the countermeasure

For a further analysis of the performance overhead of the software implementation three programs are selected. These are Advanced Encryption Standard (AES) [199], 4 digit PIN verification and 16-bit modular multiplication. Given three 16-bit numbers C, M and N, the 16-bit modular multiplication computes $(C * M)\%N$. To evaluate these programs we implemented an abstract processor that unrolls the programs and computes the overhead based on how many of the affected instructions are executed. The overall overhead is presented in Table 7.2.

Selected Programs	**TwinStack**	**IntegrityStack**
AES	43.9%	102.4%
PIN Authentication	69.2%	92.3%
16-bit Modular Computation	10.6%	21.9%

Table 7.2: Overall increase in executed instructions of software implementation of the countermeasure

To follow some good hardware programming practice, in all test programs we pushed the initial values of all used registers before a function is invoked and popped them back at the end of function execution. At this point it may be worth noting that with little optimisation the performance overhead can go significantly lower.

7.5.3 Latency

Latency is the number of instructions executed after a fault is injected into the stack and the processor becomes aware of it. For the processor to become aware of

the fault the erroneous item needs to be popped out of the stack. That means the fault can not be detected until the faulty item reaches the top of the stack. The latency measurements are of our proposed countermeasures is listed in Table 7.3 and explained subsequently.

	Implementation	
Countermeasures	**Hardware**	**Software**
TwinStack	N	$N+2$
IntegrityStack	N	$N+4$

Table 7.3: Latency measurements of the countermeasures

In the above table, N represents the number of instructions executed by the processor until the erroneous stack item is read out by the processor. In both hardware implementations, the fault is detected as soon as the erroneous item is read out. Therefore, the latency is the number of instructions that are executed after the fault is introduced until it is read out from the stack. However, in the software implementation there are additional instructions that the processor executes before the it becomes aware of the fault. In *TwinStack*, additional pop and compare instructions are executed. Whereas in *IntegrityStack*, extra pop, read, XOR and compare instructions are executed.

7.5.4 Attack Detection Capability Evaluation

In Section 7.2 we discussed security countermeasures designed/proposed to protect stack items. In addition, in Section 7.3, we defined attack types and attacker capability in regard to our proposal. Now in Table 7.4 we present the attack detection capability of our proposal and other related techniques against our attack model. In the table ✓ means that the countermeasure provides defence against the corresponding attack type and X represents that it does not.

Stack Data Protection	**Attacker Capability 1**				**Attacker Capability 2**			
	Precise Bit	Precise Byte	Unknown Byte	Unknown Fault	Precise Bit	Precise Byte	Unknown Byte	Unknown Fault
TwinStack	✓	✓	✓	✓	X	X	✓	✓
IntegrityStack	✓	✓	✓	✓	✓	✓	✓	✓
Stack Canary	X	X	X	✓	X	X	X	✓
Nonexecutable Stack	X	X	X	X	X	X	X	X
IBMAC	X	X	X	X	X	X	X	X
Redundant check	X	X	X	X	X	X	X	✓
Fault propagation	✓	✓	✓	✓	X	✓	✓	✓
Stack invariant	✓	✓	✓	✓	X	X	✓	✓

Table 7.4: Comparison of the proposed protection techniques with other related works

7.6 Summary

As stated in the beginning of the chapter, attacks on runtime data are increasingly becoming a powerful tool to defeat embedded systems security. Thus, finding a solution that works is paramount. In our work we defined four attack models and two attacker capabilities. Then we proposed a simple but effective countermeasure based on having a second stack memory as an integral component of the processor. We discussed two variants of a countermeasure in detail. Our countermeasure is not specific to any kind of algorithm or device. It can be implemented in hardware or software to prevent any form of fault attack that involve corrupting the state of the stack. Finally, we provided detailed implementation options, performance overhead for affected instructions and selected programs. We also discussed the detection latency and comparative analysis of its effectiveness against other countermeasures proposed to stop attacks on stack items.

Chapter 8

Program Instructions and Control Flow Security

Contents

8.1 Introduction 136

8.2 Basic Blocks 138

8.3 Control Flow 139

8.4 Instructions Integrity 144

8.5 Implementation 147

8.6 Summary 150

We start the chapter by discussing runtime threats to embedded systems control flow and instructions during execution. We then provide a brief explanation of basic blocks. This is followed by a detailed discussion and security proposal for runtime control flow and executed instructions integrity. We then present our implementations and test results. Finally, we summarise the chapter by providing the key points presented.

8.1 Introduction

Embedded systems are becoming increasingly widespread and inter-networked both through wireless and the internet means. This opens them up to a number of security threats and exploits that have traditionally targeted personal computers. Thus, the question of security in embedded systems received huge interest amongst researchers [207]. The extensive security research in the context of general-purpose computing and communication led to advances in security protocols and cryptographic algorithms [208, 209]. While this provides a strong basis for securing embedded systems, it is now well accepted that secure implementation and execution of programs is critical to the overall system security.

A system security can be compromised either through the execution of a program from untrusted source or the corruption of the program binary. An adversary may corrupt the binary while it is being downloaded or after it is stored in the target device's memory. Normally, the origin of a computer program is verified through the means of digital signatures [210]. The program developer digitally signs the program binary and then the user verifies it before installing the program. However, once the program is installed it is still vulnerable to runtime attacks and needs protection against such intrusions.

In the literature several mechanisms have been proposed to protect embedded system programs from runtime intrusions. Shufu et al. [211] proposed a hardware supported execution flow monitor subsystem. In their proposal the program is analysed by an offline tool that extracts a monitoring graph. This graph is then loaded into the monitoring subsystem which compares it with information generated by the main processor during runtime. The authors also discussed address, opcode, load/store, control flow and hash patterns as possible options for such information.

In [212] Arora et al. proposed a hardware assisted runtime monitoring of embedded system programs. In the paper the authors use function call and return tables to verify control flow jumps. This tables are generated by a modified compiler function. In addition, they discussed hash function based executed instructions

integrity verifier. In [213] the authors explored how AES can be used to verify the integrity of a program during execution. They used Multiple Input Shift Registers (MISR) to compute the ciphertexts of executed basic block instructions. An offline tool simply encrypts portions of the program using a pre-chosen key and the resulting ciphertext is then used as integrity value for verification.

Krutartha et al. [214] discussed a monitor processor that verifies execution flow path and basic block execution time in multi-processor applications. The paper proposes a dedicated hardware module that communicates with individual processor cores using FIFO data structure. Prior to installation the program passes through a special function of a compiler that generates a *trace file*. The *trace file* contains valid execution flow paths and expected execution time of individual basic block. This file is then used by the monitor processor during runtime.

All the proposed techniques have one thing in common. That is dividing the program into smaller sections of sequentially executed instructions, also known as basic blocks. Basic blocks are discussed in detail in section 8.2. The monitoring information is computed from each basic block. More information on security of embedded devices can be found in [215, 216, 217, 218]. In our work we explored an alternative approach of protecting program attributes during execution. In our approach we use two lookups (one for function calls and another for basic blocks) to verify inter and intra-procedural control flows and opcode integrity verification. Our approach is also complementary to our program data protection discussed in chapter 7.

The rest of the chapter is organised as follows. In section 8.2 we discuss the notion of basic blocks during embedded program execution. In section 8.3 we discuss control flow jumps that a program may follow during runtime. We explained two types of control flow jumps; the *inter-procedural* and *intra-procedural* control flow jumps. In addition, we discuss our countermeasure against attacks that target control flow jumps. In section 8.4 we discuss how the integrity of instructions can be protected during program execution. In section 8.5 we provide implementation options and results of our techniques. Finally, we conclude the chapter in section 8.6.

8.2 Basic Blocks

A basic block is a linear sequence of program instructions that has only one entry and one exit point [147]. The entry and exit points are the first and last instructions to be executed with in the basic block. A basic block may have multiple predecessors and at most two successors. In case of a loop it may be its own predecessor and/or successor. In a program there are two special basic blocks; the entry and terminating basic blocks. An entry basic block does not have a predecessor and a terminating basic block does not have a successor within the program. Program instructions within a basic block are executed sequentially. The last instruction to be executed within a basic block (exit point) usually is a jump instruction. This jump instruction can be conditional (eg. if $\cdots$ else), unconditional (eg. goto) or a function call. The jump target (the first instruction to be executed after the jump) is the entry point of the next basic block.

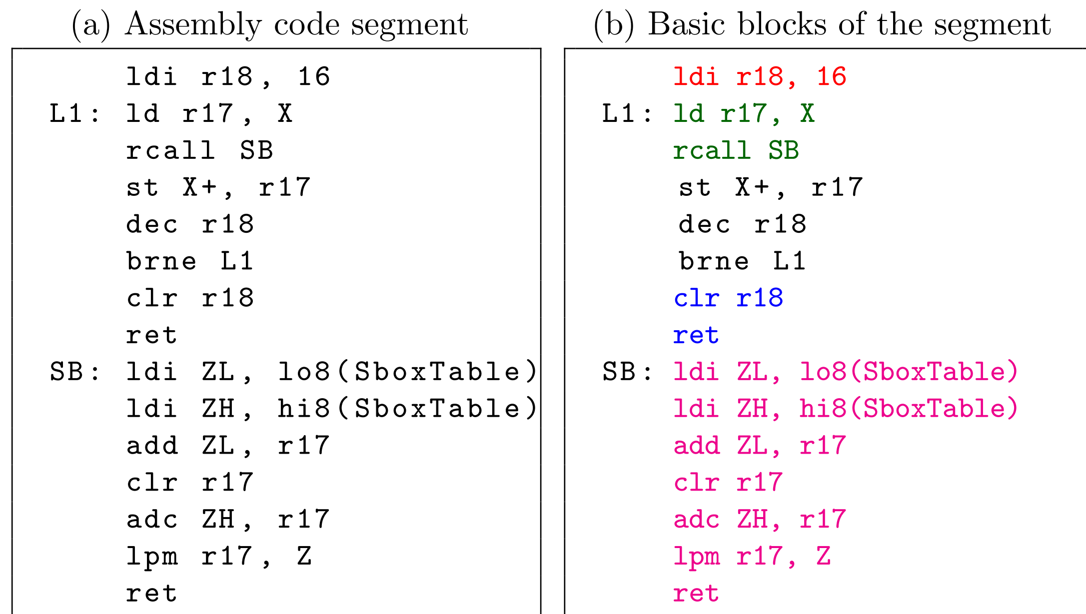


Figure 8.1: Assembly code segment and its basic blocks.

Picture (a) of Figure 8.1 is a code segment from an assembly implementation of one of the four AES round functions (the substitute box function). The code segment starts by loading the value 16 to a register (`r18`) and proceeds to the next

instruction. The next instruction (`ld r17, X`) loads the content of a memory address pointed by `X` to `r17`. However, it is also a jump target of the conditional jump instruction `brne L1`. Therefore, `ld r17, X` is an entry to a new basic block. The `rcall Sbox` invokes the function `SB` and therefore is the exit point of the basic block. Other exit points within the code segment are the `brne L1` and the two `ret` instructions. Using the rules of basic block we can divide the code segment into five basic blocks and are multi-coloured as shown in picture (b) of Figure 8.1.

8.3 Control Flow

A control flow is the order at which function calls and instructions are executed at runtime. Within a computer program there are two types of control flow transfers; *inter-procedural* and *intra-procedural* control flow transfers. *Inter-procedural* control flow transfers are the result of function calls, whereas *intra-procedural* control flow transfers are caused by conditional or unconditional branching statements.

8.3.1 Inter-Procedural Control Flow

Inter-procedural control flow transfer is done through function calls. So, it is common to represent inter-procedural control flow transfers using a *function call graph*. Generally, there are two types of inter-procedural control flow transfers; the function call and return control transfers. Such control flow transfers include a simple function call with in the program, library or system calls.

Listing 8.1: Example code of a function (function A)

```
1 int A(int value) {
2       int a = 0;
3       if(value < 0) {
4               a = B(value);
5       } else {
6               a = C(value);
```

```
7         }
8         return D(a);
9   }
```

To explain inter-procedural control flow further, we consider an example code function A that has three function calls before it completes the execution of the function. The associated code (pseudo code representation) of function A is shown in Listing 8.1 and the corresponding control flow diagram in Figure8.2. Each invocation of a function (e.g. B, C and D in Listing 8.1) diverts the execution to the first instruction within the invoke function and returns back when the execution completes. The return address is normally pushed into the stack when a function is invoked and popped when its execution is completed.

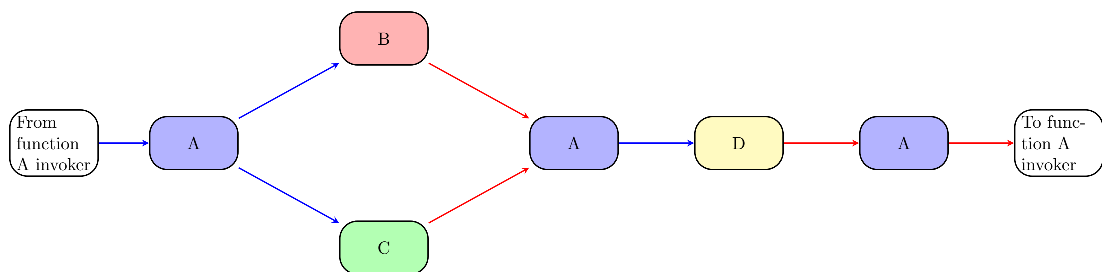


Figure 8.2: Control flow diagram of function A.

To verify the validity of function call control transfers at runtime, the processor needs some pre-computed information. To explain how this information is computed we will use the control flow diagram of function A shown in Figure 8.2. Here, we will only consider the control flow jumps within (ignoring the jumps before and after) the function A. Also it may be worth mentioning that function calls are represented by blue arrows and returns by red arrows in the control flow diagram. When a program is compiled, the compiler generates a function property table. This table is generated by taking every possible valid execution flow of the program. This can be done by using enhanced compiler tools. The property table contains information like unique function identity, list of valid control flow contexts and pointer into another table (which is discussed in detail in section 8.3.2) as shown in Table 8.1.

Table 8.1: Function property lookup table $TABLE_{property}$

Unique_FID	**Address**	**Valid_Contexts**	**Ptr_BB**
⋮	⋮	⋮	⋮

The **Unique_FID** is a unique function identifier assigned by the compiler. In the control flow diagram in Figure 8.2, we used a simple character representation, however, a compiler may use a sequence of bytes. For instance the **Unique_FID** for functions A, B, C and D are `0x01FC`, `0x01FD`, `0x01FE` and `0x01FF` respectively. The **Valid_Contexts** is list of the XOR of **Unique_FID** of all functions that are executed by the processor before the corresponding function is invoked. Using function A as an example, the first possible inter-procedural control flows are to B or C depending the input integer `value`. The inter-procedural control flow can be represented as $A \to B$ or $A \to C$, where $\to$ represents the direction of the execution flow. Therefore, we can represent the **Valid_Contexts** for functions B and C as $A \oplus B$ and $A \oplus C$ (i.e. `0x01FC` $\oplus$ `0x01FD` and `0x01FC` $\oplus$ `0x01FE`) respectively. Another entry in the table, **Address** is the address of the first instruction within the function. This address is used by the inter-procedural control flow jump verifier to identify which function is being currently executed and can be easily acquired by reading the processor's `Program Control` register. The final entry in the table, **Ptr_BB** is an index pointer to another table, $TABLE_{BasicBlock}$, which is also generated by the compiler. The details of the table is discussed in the following section 8.3.2. Based on the above discussion the full $TABLE_{Property}$ for the example function A will look like as shown in Table 8.2.

Table 8.2: Function property lookup table $TABLE_{property}$ for example function.

Unique_FID	**Address**	**Valid_Contexts**	**Ptr_BB**
`0x01FC`	PC_A	$A \oplus B \oplus A$ $A \oplus C \oplus A$ $A \oplus B \oplus A \oplus D \oplus A$ $(A \oplus B \oplus D)$ $A \oplus C \oplus A \oplus D \oplus A$ $(A \oplus C \oplus D)$	···
`0x01FD`	PC_B	$A \oplus B$	···
`0x01FE`	PC_C	$A \oplus C$	···
`0x01FF`	PC_D	$A \oplus B \oplus A \oplus D$ $A \oplus C \oplus A \oplus D$	···

To verify the validity of inter-procedural control flow jumps at runtime, the processor keeps track of all execute function, which we refer to as `context` in our discussion. The `context` is `0x0000` before the execution starts. When it starts executing A, the processor updates the context simply by XORing it with the function identifier (i.e. `0x0000` $\oplus$ A). Now the current value of `context` becomes `0x0000` $\oplus$ `0x01FC` that is the same with `0x01FC`. Using function A as an example, the next function to be executed is B or C depending on the input integer `value`. Therefore, the next possible (valid) `context` values are $A \oplus B$ (`0x01FC` $\oplus$ `0x01FD`) or $A \oplus C$ (`0x01FC` $\oplus$ `0x01FE`). If the current `context` is among the list of **Valid_Contexts** of the current function then the inter-procedural control flow jump is regarded as valid.

8.3.2 Intra-Procedural Control Flow

When a function is invoked the execution flow is transferred to the first instruction of the called function. The execution flow will only return when the execution of the function completes. After the discussion on inter-procedural control flow transfers, the next logical question should be, how can we verify the control flow jumps that occur within the function itself? We refer to such control flow jumps as intra-procedural control flow transfers (jumps). Unlike inter-procedural control flow jumps, intra-procedural control flow jumps are caused by conditional or unconditional branching statements.

Listing 8.2: Example code of a function (function B)

```
1  int B(int value) {
2        int b = -1 * value;
3        int result = 0;
4        if(b%2 == 0) {
5                result = 2 * value;
6        } else {
7                result = 3 * value;
8        }
9        return result;
10 }
```

Intra-procedural control flow jumps only divert the execution flow to an instruction within the function. For instance lets consider function B as shown in Listing 8.2. When function B is invoked the execution flow is transferred to the instruction `int b = -1 * value` and executes the next 3 instructions sequentially. After executing `if(b%2 == 0)`, the control flow may jump to `result = 3 * value` depending on whether `b` is even or odd. Finally, completes the function by executing `return result`. Before we try to verify such control flow transfers we need to breakdown the function into basic blocks. Like the function property table for inter-procedural control flow, basic block information of each function is generated using a customised compiler tool for intra-procedural control flow purposes. The basic block information generated can be represented as a simple lookup table as shown in Table 8.3.

Table 8.3: Basic block information lookup table ($TABLE_{BasicBlock}$)

BB_ID	**BB_Offset**	**S**	**Integrity_Value**
⋮	⋮	⋮	⋮

Each basic block in a program has a row entry on the basic block information table, $TABLE_{BasicBlock}$, and are arranged in functions. In other words, the basic blocks belonging to a functions are represented in consecutive rows. Each row is represented as a tuple of information row_k*(BB_ID, BB_Offset, S, Integrity_Value)*, where k being the row index in the $TABLE_{BasicBlock}$. The *BB_ID* is a unique basic block identifier in the corresponding and *BB_Offset* is the relative address of the basic block from the start of the function. In addition, the table also contains information like a list of unique identifiers of successor basic blocks, *S*, and the message integrity code of the instructions within that basic block, *Integrity_Value*.

To elaborate on this further, lets consider function B, shown in Listing 8.2, as an example. The example code has four basic blocks. The first basic block contains three instructions in line codes 2, 3 and 4. Basic blocks 2, 3 and 4 have only one instruction in line codes 5, 7 and 9 respectively. These basic blocks are represented by 4 consecutive rows arranged in order of appearance in the function in $TABLE_{BasicBlock}$. Assuming the line codes are the actual offset and basic blocks 1, 2, 3 and 4 are represented by unique identifier `0xFA`, `0xFB`, `0xFC` and `0xFD` respectively the $TABLE_{BasicBlock}$ will look like as shown in Table 8.4.

Table 8.4: Basic block information lookup table for function B.

BB_ID	BB_Offset	S	Integrity_Value
0xFA	2	0xFB, 0xFC	...
0xFB	5	0xFD	...
0xFC	7	0xFD	...
0xFD	9	-	...

When a function is called, the inter-procedural control flow verifier validates the jump and passes the value of **Ptr_BB** to the intra-procedural control flow verifier. The intra-procedural control flow verifier will use the information stored in the corresponding row of $TABLE_{BasicBlock}$ to verify the execution flow of the basic blocks. The intra-procedural control flow verifier will use this information to locate the corresponding basic block information of the function in $TABLE_{BasicBlock}$. Every time an intra-procedural control flow jump occurs the verifier will use the basic block information if the target basic block is a valid successor.

8.4 Instructions Integrity

Some embedded system attacks may not result in a control flow violation. For instance, altering instructions inside a basic block may not result in an illegal control flow jump, however it may still result in an erroneous execution of the basic block. In order to detect such attacks we need a different but complimentary approach to checking the integrity of executed instructions at runtime. Normally, integrity of a given message can be verified with the help cryptographic hash functions. The main design principle of a hash function is given a message x and its hash value $H(x)$, it is computationally infeasible to find another message y, where $x \neq y$ and $H(x) = H(y)$. Thus, it is difficult for an attacker to modify the basic block instructions and pass the integrity checking process.

To verify the integrity of every basic block in a program their hash values must be computed beforehand and written into the verifier's memory. Hashing can be computationally expensive and need a dedicated hardware module to keep

up with the processor's execution speed. Moreover, storing the fixed size hash values (normally between 16 to 20 bytes) of each basic block may incur intensive memory load on the resource constrained device. To solve such a challenge only a selected number of bits from the final hash value can be stored and subsequently used for integrity checking during the execution of the program.

The process of basic block integrity verification starts during the program compilation. The compiler divides the program into basic blocks and computes their integrity values. The integrity values are then written into the basic block information lookup table $TABLE_{BasicBlock}$ 8.3. This process is visually presented Listing 8.3 and the highlighted lines refer to the hash value of the basic blocks

Listing 8.3: Basic blocks with integrity vectors.

```
1           ldi r18, 16
2           *!5=#:?-{0
3   Loop1:  ld r17, X
4           rcall Sbox
5           $0-F*}<;.=
6           st X+, r17
7           dec r18
8           brne Loop1
9           /%:_+8*K#
10          clr r18
11          ret
12          !3Da(<+?-&
13   Sbox:  ldi ZL, lo8(Sbox_Table)
14          ldi ZH, hi8(Sbox_Table)
15          add ZL, r17
16          clr r17
17          adc ZH, r17
18          lpm r17, Z
19          ret
20          #$!18%RE9*
```

Program instructions have two parts; the opcode and the operand. The opcode is the binary representation of the instruction and never changes. However, the

operand specifies the data that is to be manipulated by the instruction. Now for the processor to be able to verify the integrity values, it must be computed over the immutable part (the information that will stay the same during compilation and execution) of the instructions. For this reason the basic block integrity values are computed over the opcode of the executed instructions within the basic block.

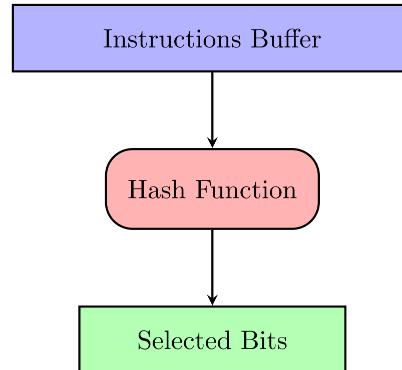


Figure 8.3: Hash generating scheme.

Figure 8.3, illustrates the hash value generation process. The opcode of executed instructions are streamed into a buffer which is later fed to the hash function. Then selected bits of the hash value is used for checking. A hash function maps a variable size input data into a fixed size output. Basic blocks contain a varying number of instructions and deciding the buffer size poses a design challenge. If it is bigger than the basic blocks, the hash generator will end-up appending the buffer and if it is too small it will have to repeatedly compute the hash values of multiple buffers for a single basic block. For this reason we selected symmetric encryption based hash generator algorithm, the Davies-Meyer scheme [219]. The Davies-Meyer architecture is shown in Figure 8.4.

In the Davies-Meyer scheme, the instruction buffer serves as the key, K_i, to the underlying symmetric cipher E. Therefore, the block size of K_i must match the expected size of the specified cipher algorithm. The previous hash function H_{i-1} serves as a plaintext input. The hash output H_i is the XOR result of the ciphertext output C_i and the previous hash value H_{i-1}. For the first round, where there is no previous hash value, H_0 is a pre-specified initial vector. Selected bits of the final hash value H_i are then used for integrity value checking.

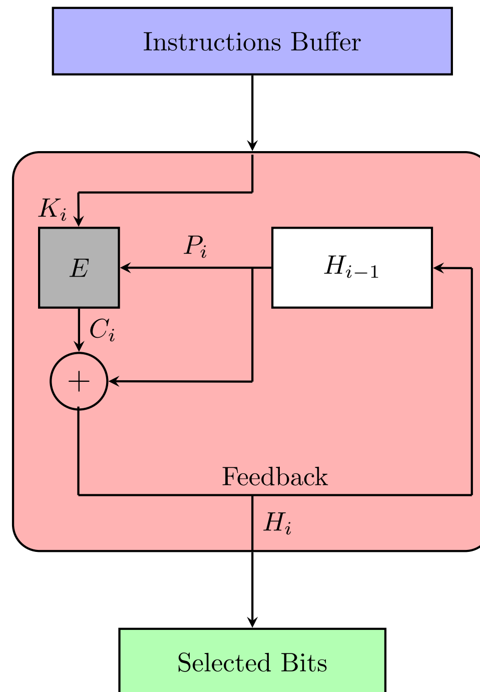


Figure 8.4: Davies-Meyer hash generation scheme.

8.5 Implementation

Practical implementation of the techniques discussed in this chapter pose certain challenges. The inter and intra-procedural control flow use table lookups. Hence, a faster way of searching a memory is critical. However, the integrity verification of instructions use cryptographic hash functions. This presents its own implementation challenges; (a) the hash algorithm should be relatively fast to compute and compare, and (b) it should be difficult (preferably impossible) for an attacker to modify the code and still pass the verification phase. Next we will explore the implementation options of these challenges.

8.5.1 Lookup Tables

In hardware the lookup tables can be directly implemented using *Content Addressable Memory (CAM)*. A CAM is a read-writeable memory that also includes

additional logic circuit to provide fast memory operations, such as lookup table search operations [220]. Smaller CAMs, with only a few hundred entries, can perform a complete search with a latency of only one clock cycle. Larger CAMs may take multiple clock cycles. As a proof of concept we have implemented our own content addressable memory, in VHDL, that can search a 256 byte SRAM for a particular byte in one clock cycle. A CAM design contains a memory, search logic circuit and interface ports.

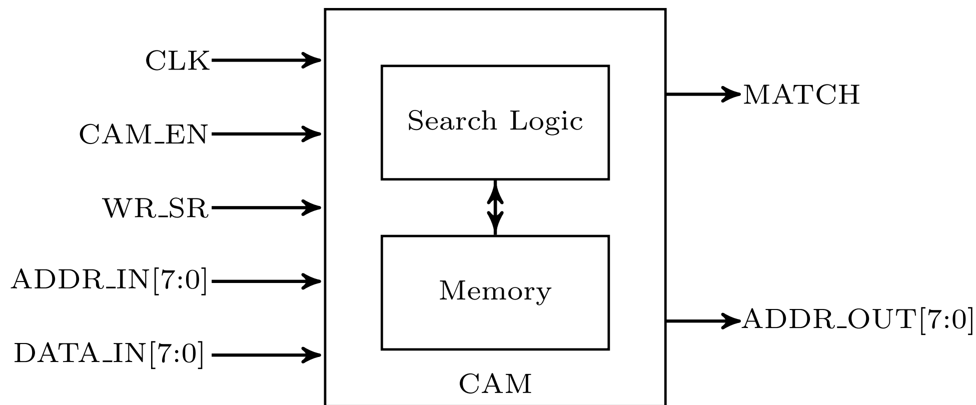


Figure 8.5: Content Addressable Memory (CAM) architecture.

The `Memory` and `Search Logic` in Figure 8.5 refer to the storage and search circuit of the CAM device respectively. The `CAM_EN` is an active low enable signal and activates the storage and searching circuit. The `CLK` and `WR_SR` signals are the clock and write/search signals respectively. The CAM device writes the value of `DATA_IN` into the `Memory` when `WR_SR` is '0' and searches the value of `DATA_IN` from the `Memory` when `WR_SR` is '1'. The `ADDR_IN` the write memory address. The VHDL source code for the CAM and testbench are available for download at [35]. The CAM uses the two output interface ports `MATCH` and `ADDR_OUT` to indicate the search result. When a hit is found the `MATCH` signal becomes '1' and its address in the memory is written to `ADDR_OUT`, otherwise, `MATCH` becomes '0' and the `ADDR_OUT` will be in high impedance state.

Before we could search the memory for a specific byte we needed to write into it first. To fully test out implementation of the CAM we wrote a permutation of bytes `0x00 - 0xFF` into the memory. To do that we selected the AES substitute box with one modification. We replaced the last byte `0x16` with `0x00`. The red

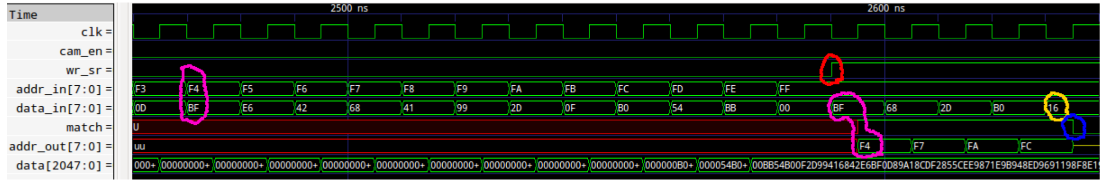


Figure 8.6: 256 byte Content Addressable Memory (CAM).

circle indicates the change of operation from write to search operation. As shown in Figure 8.6 we run the search circuit on bytes `0xBF, 0x68, 0x2D` and `0xB0` and the CAM returned a match found and their address in the memory within a single clock cycle for each search. For instance, the magenta circle (on the left) indicates `0xBF` is written into address `0xF4`. Then later the CAM returned a match found and the address `0xF4` when searching for `0xBF` as highlighted by the magenta on the right. Next we searched `0x16` (highlighted by the yellow circle) which was intentionally not written to the memory, and the CAM search logic returned match not found as shown by the blue circle in the figure.

8.5.2 Hash Computation

As discussed in section 8.4, our hash computation scheme uses a symmetric cipher algorithm. In our work, we have selected Advanced Encryption Standard (AES) as our symmetric cipher algorithm E. The advantage of using AES in Davies-Meyer scheme is that the instruction buffer can be adjusted to 128 or 256 bits, depending on the average size of the basic block, without changing the design. The block size of P_i and C_i is 128 bit. Finally, for the integrity value checking only the selected 16 bits of the final H_i are used.

Since the integrity verification is performed in parallel with the execution of instructions, the verification time needs to be the same or less than the execution time of the next basic block. However, this may not be true for short basic blocks. In this case the verifier module sends a halt signal to the processor until it finishes verifying the current basic block.

Our hash computation implementation only requires 40 clock cycles to compute

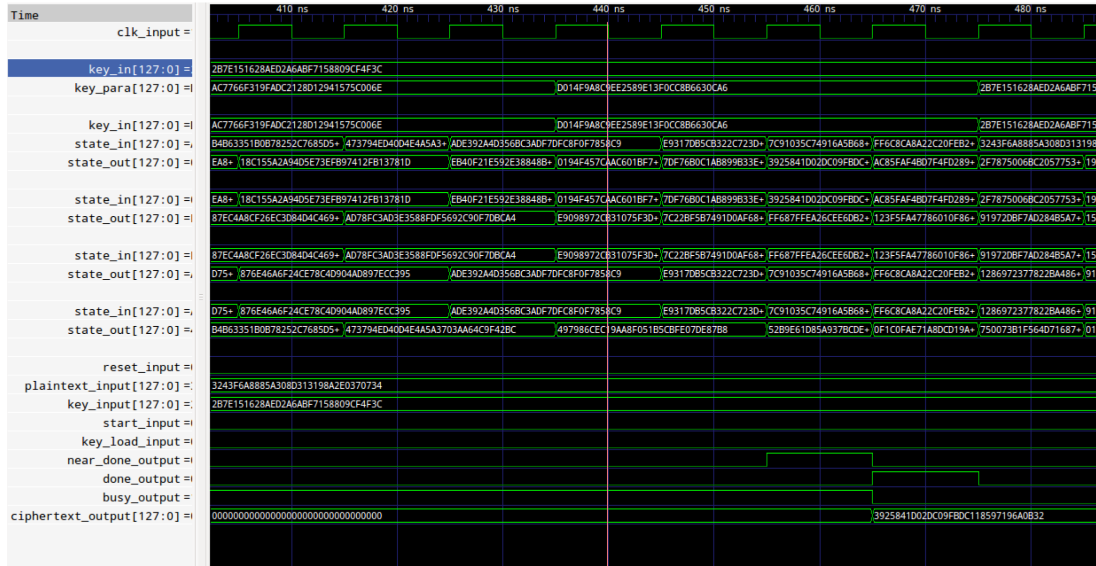


Figure 8.7: AES implementation in VHDL.

and verify. This is presented in Figure 8.7. However, with more optimisation and pipelining of the Davies-Meyer scheme this can be shortened further. Evaluating the performance overhead incurred into the overall execution requires analysing the time consumed while the processor was halted by the verifier module. In addition to that the last round of verification for the last basic block is added to it.

8.6 Summary

We started the chapter by discussing security proposals for embedded systems. We then discussed the notion of basic blocks. Next we discussed our control flow jump verification proposal. Our proposal covers both the inter and intra-procedural control flow jumps. This is achieved by using two lookup tables generated by a modified compiler function. The hardware verifier subsystem uses these lookup tables and runtime generated information to verify the execution flow. In addition, we have also discussed a basic block instructions integrity verifier subsystem. This subsystem uses information from the lookup table and processor registers (like the PC and IR) to verify the integrity of executed instructions. We

used the Davies-Meyer hash computation scheme to generate the integrity values. Finally, we presented our implementation for the proposals.

Chapter 9

Conclusion and Future Work

Contents

9.1 Summary and Conclusions 153

9.2 Recommendation for Future Research 156

In this chapter, we conclude the thesis. Here we summarise the controbutions of our wprk and discuss the future research challenges that may need to be addressed later.

9.1 Summary and Conclusions

The main goal of this thesis was to explore the feasibility of platform verification and secure program execution in embedded devices.

We started the discussion by mapping the evolution of embedded system devices and their applications. Embedded devices are used by individuals and organisations with an ever growing dependency on them, such as commercial and military communication systems, transport (land, sea and air) control systems, etc. Therefore, there needs to be a generic security architecture that can be easily integrated into any of these systems. Such security architecture incorporates two phases of security checks. The first phase is the pre-deployment platform verification. This identifies any unnecessary modules/components incorporated into the embedded device. The second phase is the runtime program execution. This phase protects programs running inside the device during execution.

Prior to the discussion on the core contributions of this thesis, we provided a detailed discussion on embedded systems. We identified the different components that make an embedded system and explained them individually. We then briefly discussed the application development process and tools required to successfully write an application for an embedded device. This discussion is then followed by a list of embedded device applications. Subsequently, we discussed the different type of attacks that target embedded systems. These attacks range from manipulating runtime program data, platform reverse engineering to inserting hardware Trojans and parasite circuits. We then proceeded to a brief discussion of the current security countermeasures used in embedded systems.

As mentioned earlier, the platform verification phase is about identifying if unwanted or malicious modules are present inside the embedded device. Normally, a straight forward method of avoiding unwanted or malicious module would be for the designer to produce the devices at a local and secure foundry. However, commercially this is not feasible today. This statement is supported by the increasing relocation of manufacturing plants to a low cost structured countries. In addition, such modules are very difficult, if not impossible, to be identified

through purely functional testing. Therefore, in our work we selected the side channel leakage of the target device as the main input in achieving this goal.

Before we can use the side channel leakage to identify unwanted modules, we need to build a leakage template first. The templates are built using side channel leakage collected from few genuine devices. The leakage, such as power consumption, is collected while the target device executes a selected program repeatedly under different conditions. In this thesis, we created two types of power consumption templates; instruction and basic block level templates. These two templates are then used to verify executed instructions and control flow jumps of the target program inside the target device. Our templates were created using power traces collected from five identical ATMega163 based blank smart card processor.

The idea behind our proposed technique is the existence of unwanted or malicious modules will be reflected on the side channel leakage. However, this requires a high precision template matching techniques especially for the instruction-level templates. After experimenting with a number of dimensionality reduction and classification techniques, we managed to improve the previously known template recognition rate of 70.1% [32] to a 100% recognition rate. This enabled us to successfully reconstruct executed instructions and verify their integrity. On the other hand, the basic block level templates are used to reconstruct the control flow jumps and verify them. If there is unwanted, malicious or even slight change to the embedded device's modules it will be reflected on the power consumption leakage. This will eventually affect the recognition rate and the verification process will fail.

The above, side channel based verification, is an offline process performed on the target embedded device prior to integrating it into the final product. However, the device still remains vulnerable to threats and exploits during execution. Hence, it is important to deploy runtime security countermeasure to ensure the security of the device's programs during operation. This would require monitoring certain properties of the program at runtime. In our work, we have selected four program attributes for runtime security verification. These are the instructions opcode, program runtime data, inter-procedural and intra-procedural control flow jumps.

Runtime program data is normally stored in a First In Last Out (FILO) data structure, also known as stack. Thus, protecting runtime data requires protecting the stack items. In this thesis, we proposed a dual-stack processor architecture as a runtime data protection method. We have discussed two varieties of the dual-stack architecture. A straightforward approach of verifying the integrity of runtime data is to simply keep a copy of the stack and perform redundant stack operations. This would require an attacker to perform two identical attacks on both stacks to bypass the redundant stack operation check-ups. Although this is considered very difficult, if not impossible, it is a truism that attacks get better in time. Therefore, to stop such an attack we proposed a second variety of the dual-stack architecture. In the second variety the second stack is used for storing integrity value of the items stored in the normal stack. We provided a detailed implementation analysis, performance overhead, detection capability and latency of both varieties of our dual-stack architecture.

The other program attribute that needs monitoring is the control flow jumps. A control flow jump is a change in the execution flow of the program instructions. There are two types of control flow jumps; inter-procedural and intra-procedural control flow jumps. Inter-procedural control flow jumps refer to function call and return control flow jumps. On the other hand, intra-procedural control flow jumps refers to execution flow change within the function. In our thesis, we proposed a lookup table based control flow jump verification method for embedded systems. During compilation the compiler extracts information about inter and intra-procedural control flow jumps such as function address, basic block offset and integrity value. These information is then written into the monitor's memory, which it will later use for runtime verification. The main implementation challenge of such method is fast memory search technique. In our work, we have implemented content addressable memory that is capable of full memory search under one clock cycle.

Finally, the last attribute is the instruction integrity of executed basic blocks. Verifying their integrity requires computation and comparison of each basic block's integrity value. To compliment with our control flow jumps verification, the integrity values are computed over basic block instructions. In our computation

we have used symmetric cipher based hash scheme, the Davies-Meyer scheme. The Davies-Meyer scheme is a feedback loop computation and will be repeated multiple times depending on the size of the basic block. Once the integrity values are computed only selected bits are stored in the monitor to save memory requirement. In this thesis, we have implemented and tested this scheme in VHDL and presented the results. This discussion completed our work of secure program execution.

9.2 Recommendation for Future Research

Our aim was in this research was to explore the security of embedded devices and analyse the feasibility of a generic security architecture. We have achieved our goals by proposing and experimenting on pre-deployment platform verification and runtime program execution techniques. However, we believe that there is still a long journey ahead before our proposal can be considered practical.

So far, we have implemented the platform verification and runtime secure program execution techniques independently. We have also tested an open source based 8 bit AVR microcontroller implemented in VHDL [221]. We consider there is a possible improvement of fully integrating our countermeasures into the microcontroller's main processor. Furthermore, since our proposals use information generated by a modified compiler there needs to be a complete feasibility study of compilers and how they can be modified to support our architecture. Additionally, if our proposal is to be considered practical, there needs to be a detailed study of how it can be extended to support multi-core processors. Program developers often issue updates to application after deployment. Finally, it is important to look into the possibility of a secure program update mechanism.

Bibliography

[1] Starbug and Karsten Nohl. Hardware reverse engineering, (Accessed) 2015. https://events.ccc.de/congress/2008/Fahrplan/attachments/1218_081227.25C3.HardwareReversing.pdf. 9, 49

[2] Bartol Filipovic and Oliver Schimmel. Protecting embedded systems against product piracy: Technological background and preventive measures. Technical report, Fraunhofer research institution for applied and integrated security, April 2012. 9, 50, 51

[3] Design principles of tamper resistant smartcard processors - presentation slides, (Accessed) 2014. http://www.cl.cam.ac.uk/~mgk25/sc99-tamper-slides.pdf. 9, 63

[4] Christopher Tranovsky. Security failures in secure devices, (Accessed) 2015. https://www.blackhat.com/presentations/bh-dc-08/Tarnovsky/Presentation/bh-dc-08-tarnovsky.pdf. 9, 63

[5] Online Article. Konrad zuse - the first relay computer, (Visited) 2014. http://history-computer.com/ModernComputer/Relays/Zuse.html. 18

[6] World Information. 1940s-1960s: First generation computers, (Visited) October 2014. http://world-information.org/wio/infostructure/100437611663/100438659338. 18

[7] Michael Riordan, Lillian Hoddeson, and Conyers Herring. The invention of the transistor. *Reviews of Modern Physics*, 71(2), 1999. 18

[8] Texas Instruments. The chip that jack built, (Visited) 2015. http://www.ti.com/corp/docs/kilbyctr/jackbuilt.shtml. 18

[9] James Tomayko. Computers on board the apollo spacecraft. In *Computer in Spaceflight: The NASA Experience*. NASA History Office, Wichita State University, Wichita, Kansas, 1988. 19

[10] Data Mining International Research & Development Center For Cloud Computing and Warehousing. Embedded systems, (Visited) October 2014. http://www.c3dmw.com/IRDC3DMW/EmbeddedSystem2.jsp. 19

[11] Charles H. Beck. Minuteman computer users group: D17b computer documentation, (Visited) October 2014. 19

[12] Intel Museum. The story of intel 4004: Intel's first microprocessor, (Visited) October 2014. http://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html. 19

[13] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. 20

[14] Computer history museum, (Accessed) 2015. http://www.computerhistory.org/timeline/?year=1946. 20

[15] Intel Corporation. MCS-48 microcontroller user manual, (Online) 1978. 20

[16] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*, volume Third Edition. John Wiley & Sons, Inc., 2003. 20, 63

[17] Bertrand du Castel. Personal history of the java card, (Visited) October 2014. https://sites.google.com/site/personalhistoryofthejavacard/. 21

[18] Oracle. Java card classic platform specification 3.0.4, September 2011. http://www.oracle.com/technetwork/java/javacard/specs-jsp-136430.html. 21

[19] MULTOS. Multos: Secure multi-app os for smart devices, (Visited) October 2014. http://www.multos.com. 21

[20] Oracle. Java card: Overview, (Visited) October 2014. http://www.oracle.com/us/technologies/java/embedded/card/overview/index.html. 21

[21] NFC Forum. Nfc forum, (Visited) November, 2014. http://nfc-forum.org. 21

[22] J. Hller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle. *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence.* Elsevier, 1st edition, 2014. 21

[23] LITEPOINT. Test considerations for nfc enabled devices in manufacturing: Why it is important and how to perform effective tests, (Accessed) August 2015. http://www.litepoint.com/wp-content/uploads/2014/03/NFC_Whitepaper_021814.pdf. 21

[24] Bernard Cole. Architecture overlap applications, March 1995. Electronic Engineering Times. 21

[25] Joseph Sifakis. The embedded systems challenge, Visited, January 2014. http://www.telecom-paristexh.fr/fileadmin/documents/images/Enseignments/FI/Experts_STIC/SIFAKIS_ENST-optim.pdf. 21

[26] Tusted Computing Group (TCG). Trusted platform module main specification, (visited) 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. 22, 65

[27] Jan-Erik Erberg and Markku Kylänpää. Mobile Trusted Mo dule (MTM) - An Intro duction. Technical report, Nokia Research Center, November 2007. 22

[28] ARM Limited. Arm security technology: Building a secure system using trustzone technology. Technical report, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. 22, 67

[29] GlobalPlatform. Globalplatform device technology: Tee system architecture, version 0.4, 2011. 22, 68

[30] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. The b-side of side channel leakage: Control flow security in embedded systems. In Tanveer A. Zia, Albert Y. Zomaya, Vijay Varadharajan, and Zhuoqing Morley Mao, editors, *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, volume 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 288–304. Springer, 2013. 25

[31] Mehari Msgna, Konstantinos Markantonakis, David Naccache, and Keith Mayes. Verifying software integrity in embedded systems: A side channel approach. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 261–280. Springer, 2014. 25

[32] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. Building a side channel based disassembler. *Transactions on Computational Science*, 6340:78–99, 2010. 25, 55, 94, 154

[33] Microchip Technology Inc. Microchip website, (Visited) March 2013. `http://www.microchip.com`. 25, 40

[34] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. Precise instruction-level side channel profiling of embedded processors. In Xinyi Huang and Jianying Zhou, editors, *Information Security Practice and Experience - 10th International Conference, ISPEC 2014, Fuzhou, China, May 5-8, 2014. Proceedings*, volume 8434, pages 129–143. Springer, 2014. 25

[35] Thesis source codes, (Created) 2016. `https://github.com/m-g-msgna/platform-verification-and-secure-executio`. 28, 90, 113, 127, 148

[36] Milan Verle. *Architecture and programming of 8051 MCU's*. MikroElektronika, (Accessed) 2015. 32

[37] Page. *A practical introduction to computer architecture*. Springer, 2009. 34

[38] Raghu Tumati. Digital to analog converter, 2006. https://ece.umaine.edu/ece/files/2012/05/ECE547_RaghuTumati.pdf. 36

[39] Marcel Pelgrom. *Analog-to-Digital Conversion*. Springer, 2015. 36

[40] Helena Handschuh and Pascal Paillier. Smart card crypto-coprocessors for public-key cryptography. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, volume 1820 of *Lecture Notes in Computer Science*, pages 372–379. Springer, 1998. 36

[41] Philip Koopman. Embedded system design issues (the rest of the story). In *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, pages 310–317, Oct 1996. 36

[42] Vijaykrishnan Narayanan and Yuan Xie. Reliability Concerns in Embedded System Design. Technical report, The Pennsylvania State University, January 2006. http://contech.suv.ac.kr/contech/courses/10h1embeddedsystem/ReliabilityConcernsInEmbeddedSystemDesigns.pdf. 36

[43] Pacemaker implantation, (Accessed) 2015. http://www.nhs.uk/conditions/PacemakerImplantation/Pages/Introduction.aspx. 37

[44] NITRD. Cyber physical systems, (Accessed) 2015. https://www.nitrd.gov/nitrdgroups/images/6/6a/Cyber_Physical_Systems_%28CPS%29_Vision_Statement.pdf. 37

[45] RISC, (Accessed) 2015. http://www.eecg.toronto.edu/~pc/research/publications/potentials91.pdf. 38

[46] Michael Freeman. A minimal CISC processor architecture for field programmable gate arrays. Technical report, Department of Computer Science, University of York, UK, (Accessed) 2015. https://www.cs.york.ac.uk/amadeus/papers/minimal%20CISC.pdf. 38

[47] ASIC basics: An introduction to developing application specific integrated circuits, (Accessed) 2015. `http://www.lannierose.com/asicbasics.pdf`. 38

[48] M. Godfrey and D. Hendry. The Computer as von Neumann Planned It. *Annals of the History of Computing, IEEE*, 15(1):11–21, 1993. 38

[49] J. von Neumann. First Draft of a Report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, June 1945. 38

[50] gedit. gedit website, (Visited) March 2014. `https://wiki.gnome.org/Apps/Gedit`. 40

[51] Ideone. Online compiler and ide, (Accessed) July, 2015. `https://ideone.com`. 40

[52] PC Magazine. Definition of: Compiler, (Accessed) 2015. `http://www.pcmag.com/encyclopedia/term/40105/compiler`. 41

[53] GNU Project. Gcc, the gnu compiler collection, (Accessed) 2015. `https://gcc.gnu.org//`. 41

[54] Oxford Dictionaries. Definition of: Simulator, (Accessed) 2015. `http://www.oxforddictionaries.com/definition/english/simulator`. 42

[55] V. Soso. Pic simulator ide, (Visited) May 2009. `http://www.oshonsoft.com/pic.html`. 42

[56] MPLAB PM3 universal device programmer, (Accessed) 2015. `http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=DV007004`. 43

[57] Keith Mayes and Konstantinos Markantonakis. *Smart Cards, Tokens, Security and Applications*. Springer, 1 edition, 11 December 2007. 47

[58] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems*. Wiley, second edition edition, 2008. 47

[59] Assia Tria and Hamid Choukri. Invasive attacks. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 623–629. Springer, 2011. 47

[60] Clemens Helfmeier, Dmitry Nedospasov, Christopher Tarnovsky, Jan Starbug Krissler, Christian Boit, and Jean-Pierre Seifert. Breaking and entering through the silicon. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 733–744. ACM, 2013. 47, 51

[61] Martin Hutle and Markus Kammerstetter. Resilience against physical attacks. In Florian Skopik and Paul Smith, editors, *Smart Grid Security: innovative solutions for modernized grid*. SYNGRESS, 2015. 48

[62] Scanning eletron microscopy, (Accessed) 2015. `http://serc.carleton.edu/research_education/geochemsheets/techniques/SEM.html`. 48

[63] Transmission electron microscopy, (Accessed) 2015. `http://cmrf.research.uiowa.edu/transmission-electron-microscopy`. 48

[64] Randy Torrance and Dick James. The state-of-the-art in IC reverse engineering. In Christophe Clavier and Kris Gaj, editors, *11th International Workshop Cryptographic Hardware and Embedded Systems - CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 363–381. Springer, September 6-9 2009. 48

[65] A7101catk2: Secure authentication microcontroller, (Accessed) 2015. `http://www.nxp.com/products/identification_and_security/authentication/A7101CATK2.html`. 50

[66] Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In Scott B. Guthery and Peter Honeyman, editors, *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999*. USENIX Association, 1999. 50, 62

[67] Hagai Bar-El. Know attacks against smartcards. Technical report, Discretix Technologies Ltd., (Accessed) 2015. `http://www.infosecwriters.com/text_resources/pdf/Known_Attacks_Against_Smartcards.pdf`. 50

[68] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 363–381. Springer Berlin Heidelberg, 2009. 51

[69] Sergei P. Skorobogatov. Local heating attacks on flash memory devices. In Mohammad Tehranipoor and Jim Plusquellic, editors, *IEEE Proceedings International Workshop on Hardware-Oriented Security and Trust (HOST), San Francisco, CA, USA, July 27, 2009.*, pages 1–6. IEEE Computer Society, 2009. 52

[70] Microchip. Pic16f62x data sheet, flash-based 8-bit cmos microcontroller, (Visited) April 2014. `http://ww1.microchip.com/downloads/en/DeviceDoc/40300C.pdf`. 52

[71] Mohammad Tehranipoor and Cliff Wang. *Introduction to Hardware Security and Trust*. Springer, 2012. 53

[72] Lawrence Wagner. *Failure Analysis of Integrated Circuit: Tools and Techniques*. Kluwer Academic Publishers, 1999. 53

[73] Julie Ferrigno and Martin Hlavác. When AES blinks: introducing optical side channel. *IET Information Security*, 2(3):94–98, 2008. 53

[74] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2000. 55

[75] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO '99*, volume 1666 of *Lecture Notes in*

Computer Science, pages 388–397, Santa Barbara, California USA, August 15-19 1999. Springer. 55, 78, 94, 97

[76] Guo liang Ding, Zhi xiang Li, Xiao long Chang, and Qiang Zhao. Differential electromagnetic analysis on aes cryptographic system. In *Web Mining and Web-based Application, 2009. WMWA '09. Second Pacific-Asia Conference on*, pages 120–123, June 2009. 55

[77] David Oswald and Christof Paar. Breaking mifare DESFire MF3ICD40: Power analysis and templates in the real world. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 207–222, Nara, Japan, September 28 - October 1 2011. Springer. 55, 94

[78] Dennis Vermoen, Marc F. Witteman, and Georgi Gaydadjiev. Reverse engineering java card applets using power analysis. In Damien Sauveron, Constantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, volume 4462 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 2007. 55

[79] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design & Test of Computers*, 24(6):535–543, 2007. 55, 94

[80] Johann Heyszl, Stefan Mangard, Benedikt Heinz, Frederic Stumpf, and Georg Sigl. Localized electromagnetic analysis of cryptographic implementations. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 231–244, San Francisco, CA, USA, February 27 - March 2 2012. Springer. 55, 78

[81] Kun Gu, Liji Wu, Xiangyu Li, and Xiangxin Zhang. Design and implementation of an electromagnetic analysis system for smart cards. In Yuping Wang, Yiu ming Cheung, Ping Guo, and Yingbin Wei, editors, *CIS*, pages 653–656, Sanya, Hainan, China, December 3-4 2011. IEEE. 55, 78

[82] Wim Van Eck and Neher Laborato. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4:269–286, 1985. 55, 78

[83] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002. 55, 57

[84] Walter Tuchman. A brief history of the data encryption standard. In Dorothy Denning and Peter Denning, editors, *Internet Besieged*, pages 275–280. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998. 55, 57

[85] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. 55, 100

[86] Dennis Vermoen, Marc F. Witteman, and Georgi Gaydadjiev. Reverse engineering Java Card applets using power analysis. In Damien Sauveron, Constantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *WISTP*, volume 4462 of *Lecture Notes in Computer Science*, pages 138–149, Heraklion, Crete, Greece, May 9-11 2007. Springer. 55, 94

[87] Christophe Clavier. Side channel analysis for reverse engineering (SCARE) - an improved attack against a secret A3/A8 GSM algorithm. *IACR Cryptology ePrint Archive*, 2004:49, 2004. 55, 94

[88] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, Santa Barbara, California, USA, August 18-22 1996. Springer. 55

[89] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications CARDIS '98*, volume 1820 of *Lecture Notes in Computer Science*, pages 167–182, Louvain-la-Neuve, Belgium, September 14-16 1998. Springer. 55

[90] Cyril Arnaud and Pierre-Alain Fouque. Timing attack against protected RSA-CRT implementation used in PolarSSL. In Ed Dawson, editor, *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 18–33, San Francisco, CA, USA, February 25-March 1 2013. Springer. 55

[91] NSA/NCSC Rainbow Series. A guid to understanding data remanence in automated information systems, (Accessed) March, 2015. `http://fas.org/irp/nsa/rainbow/tg025-2.htm`. 56

[92] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. *The Sixth USENIX Security Symposium Proceedings*, 1996. 56

[93] Sergei P. Skorobogatov. Data remanence in flash memory devices. In Josyula R. Rao and Berk Sunar, editors, *7th International Workshop Cryptographic Hardware and Embedded Systems - CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 339–353. Springer, August 29 - September 1, 2005. 56

[94] Robert Slater. Fault injection, 1998. `http://users.ece.cmu.edu/~koopman/des_s99/fault_injection/`. 56

[95] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, May 1997. 56, 119

[96] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *17th Annual International Cryptology Conference (CRYPTO)*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, August 17-21 1997. 57, 120

[97] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the Advanced Encryption Standard. *IACR Cryptology ePrint Archive*, 2002:75, 2002. 57, 120

[98] Junko Takahashi and Toshinori Fukunaga. Differential fault analysis on the AES key schedule. *IACR Cryptology ePrint Archive*, 2007:480, 2007. 57, 120

[99] Chong Hee Kim and Jean-Jacques Quisquater. New differential fault analysis on AES key schedule: Two faults are enough. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 48–60. Springer, September 2008. 57, 120

[100] Chong Hee Kim. Improved differential fault analysis on AES key schedule. *IEEE Transactions on Information Forensics and Security*, 7(1):41–50, 2012. 57, 120

[101] Christophe Giraud. DFA on AES. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *AES Conference*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, May 2004. 57, 120

[102] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012. 57

[103] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001. 57

[104] Andrey Sidorenko, Joachim van den Berg, Remko Foekema, Michiel Grashuis, and Jaap de Vos. Bellcore attack in practice. *IACR Cryptology ePrint Archive*, 2012:553, 2012. 57

[105] Olaf Henniger, Ludovic Apvrille, Andreas Fuchs, Yves Roudier, Alastair Ruddle, and Benjamin Weyl. Security requirements for automotive on-board networks. In *2009 9th International Conference on Intelligent Transport Systems Telecommunications, (ITST)*, pages 641–646, October 2009. 59

[106] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *IACR Cryptology ePrint Archive*, 2004. 60

[107] Oxford Dictionaries. Definition of obfuscate. `http://www.oxforddictionaries.com/definition/english/obfuscate`. 60

[108] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. 1997. 61

[109] Diablo. Diablo is a better link-time optimizer, (Visited) October 2014. https://diablo.elis.ugent.be/. 61

[110] http://keccak.noekeon.org. Note on side-channel attacks and their countermeasures, (Visited) 2015. 62

[111] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013. 62

[112] National Institute of Standards and Technology: Information Technology Laboratory. Security requirements for cryptographic modules (fips pub 140-2), (Visited) April, 2015. http://csrc.nist.gov/groups/STM/cmvp/standards.html#02. 62

[113] Markus Kuhn. The trustno 1 cryptoprocessor concept, (Visited) April 2015. http://www.cl.cam.ac.uk/~mgk25/trustno1.pdf. 63

[114] Robert Best. Crypto microprocessor for executing enciphered programs, 1981. US 4278837 A. 63

[115] Infineon Technologies, (Accessed) 2015. https://www.infineon.com/. 65

[116] Atmel Corporation, (Accessed) 2015. http://www.atmel.com/. 65

[117] Broadcom, (Accessed) 2015. https://www.broadcom.com/. 65

[118] Peter Wilson, Alexandre Frey, Tom Mihm, Danny Kershaw, and Tiago Alves. Implementing embedded security on dual-virtual-cpu systems. *IEEE Design & Test of Computers*, 24(6):582–591, 2007. 67

[119] GlobalPlatform, (Accessed) 2015. https://www.globalplatform.org. 67

[120] GlobalPlatform. GlobalPlatform card security requirement specification 1.0, (Online) 2003. https://www.globalplatform.org/specificationscard.asp. 68

[121] GlobalPlatform. Globalplatform device: Gpd/stip specification overview, version 2.3. Online, 2007. 68

[122] GlobalPlatform. Globalplatform device technology: Device application security management concepts and description document specification. Online, 2008. 68

[123] GlobalPlatform. Globalplatform card specification, version 2.2, 2006. 68

[124] Srivaths Ravi, Paul C. Kocher, Ruby B. Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *Proceedings of the 41th Design Automation Conference (DAC)*, pages 753–760. ACM, June 7-11 2004. 69

[125] Anoop M. S. Security needs in embedded systems. *IACR Cryptology ePrint Archive*, 2008:198, 2008. 69

[126] Defense Advanced Research Projects Agency. Darpa baa06-40, a trust for integrated circuits, Visited, May 2013. https://www.fbo.gov/index?s=opportunity&mode=form&id=db4ea611cad3764814b6937fcab2180a&tab=core&_cview=1. 69

[127] Joseph I. Lieberman. The national security aspects of the global migration of the u.s. semiconductor industry, Visited, May 2013. http://www.fas.org/irp/congress/2003_cr/s060503.html. 69

[128] Defense Science Board Task Force. High performance microchip supply, Visited, May 2013. http://www.acq.osd.mil/dsb/reports/ADA435563.pdf. 69, 71

[129] Srivaths Ravi, Anand Raghunathan, Paul C. Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embedded Comput. Syst.*, 3(3):461–491, 2004. 70

[130] U.S. Department Of Commerce. Defense industrial base assessment: Counterfeit electronics. Technical report, Bureau of Industry and Security, Office of Technology Evaluation, January, 2010. `http://www.bis.doc.gov/defenseindustrialbaseprograms/osies/defmarketresearchrpts/final_counterfeit_electronics_report.pdf`. 70

[131] F. Koushanfar, A.-R. Sadeghi, and H. Seudie. EDA for secure and dependable cybercars: Challenges and opportunities. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 220–228, 2012. 70

[132] An Feng, Michael Knieser, Maher E. Rizkalla, Brian King, Paul Salama, and Francis Bowen. Embedded system for sensor communication and security. *IET Information Security*, 6(2):111–121, 2012. 76

[133] Abdulhadi Shoufan. A hardware security module for quadrotor communication. In *International Conference on Field-Programmable Technology (FPT)*, pages 253–256. IEEE, December 10-12 2012. 76

[134] Attila Jaeger, Hagen Stuebing, and Sorin Huss. A dedicated hardware security module for field operational tests of Car-to-X communication. In *4th ACM Conference on Wireless Network Security (WiSec '11)*, June 2011. 76

[135] Sri Parameswaran and Tilman Wolf. Embedded systems security - an overview. *Design Autom. for Emb. Sys.*, 12(3):173–183, 2008. 76

[136] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the Java Card control flow. In Emmanuel Prouff, editor, *CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer, September 14-16 2011. 76, 121

[137] Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In *ACM Conference on Computer and Communications Security*, pages 15–26. ACM, October 27-31 2008. 76

[138] Gaël Delalleau. Large memory management vulnerabilities: System, compiler and application issues, Visited April 2013.
http://cansecwest.com/core05/memory_vulns_delalleau.pdf. 76

[139] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Trans. VLSI Syst.*, 14(12):1295–1308, 2006. 76

[140] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009. 76

[141] Michael Frantzen and Michael Shuey. StackGhost: Hardware facilitated stack protection. In Dan S. Wallach, editor, *10th USENIX Security Symposium*. USENIX, August 13-17 2001. 77

[142] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, SecuCode '09, pages 19–26, New York, NY, USA, 2009. ACM. 77, 122

[143] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC)*, pages 339–348. IEEE Computer Society, 11-15 December 2006. 77

[144] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pages 346–355, 1998. 77, 121

[145] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96. ACM, October 7-13 2004. 77

[146] A. Fink. *Markov Models for Pattern Recognition.* Springer, 2008. 77, 78

[147] Frances Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, New York, NY, USA, July 1970. ACM. 78, 101, 138

[148] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design & Test of Computers*, 24(6):535–543, 2007. 78

[149] Lawerence Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989. 78

[150] Allan Gut. *An Intermediate Course In Probability*, volume Second edition. Springer, Department of Mathematics, Uppsala University, Sweden, 2009. 80, 99

[151] J. R. Berrendero, Ana Justel, and Marcela Svarc. Principal components for multivariate functional data. *Computational Statistics & Data Analysis*, 55(9):2619–2634, 2011. 82

[152] Gilbert Strang. *Introduction to Linear Algebra*, volume Third edition. Wellesley-Cambridge Press, MA, USA, 2003. 83

[153] Ronald Aylmer Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936. 83

[154] Minoru Fukumi and Yasue Mitsukura. Feature generation by simple-FLDA for pattern recognition. In *CIMCA/IAWTIC*, pages 730–734. IEEE Computer Society, 28-30 November 2005. 83

[155] Lishi Zhang, Dehong Wang, and Shengzhe Gao. Application of improved Fisher Linear Discriminant Analysis approaches. In *International Conference on Management Science and Industrial Engineering (MSIE)*, pages 1311–1314, 2011. 83

[156] David Forney Jr. The Viterbi Algorithm: A personal history. *CoRR*, abs/cs/0504020, 2005. 84

[157] Atmega163 datasheet, (Accessed) 2015. http://www.atmel.com/Images/doc1142.pdf. 86

[158] MATLAB. *Version 7.10.0.499 (R2010a).* The MathWorks, Inc., Natick, Massachusetts, 2013. http://www.mathworks.co.uk/index.html. 90

[159] MATLAB. Hidden Markov Model most probable state path, Visited, March 2013. http://www.mathworks.co.uk/help/stats/hmmviterbi.html. 90

[160] Atkins Limited. MALPAS, Visited, April 2013. http://www.malpas-global.com/. 91

[161] Safecode. Software integrity controls: An assurance-based approach to minimizing risks in the software supply chain, (Accessed0 2015. http://www.safecode.org/publication/SAFECode_Software_Integrity_Controls0610.pdf. 94

[162] David Aucsmith. Tamper resistant software: an implementation. In Ross Anderson, editor, *Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer, 1996. 94

[163] Hoi Chang and Mikhail Atallah. Protecting software code by guards. In Tomas Sander, editor, *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM, Revised Papers*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2002. 94

[164] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997. 94

[165] John Larson. *The Cardio-pneumo-psychogram in Deception.* 1923. http://books.google.co.uk/books?id=b6appwAACAAJ. 94

[166] Keith Mayes, Konstantinos Markantonakis, and Calvin Chen. Smart card platform-fingerprinting. *Advanced Card Technology*, pages 78–82, October 2006. 94

[167] Georg T. Becker, Daehyun Strobel, Christof Paar, and Wayne Burleson. Detecting software theft in embedded systems: A side-channel approach.

IEEE Transactions on Information Forensics and Security, 7(4):1144–1154, 2012. 95

[168] Christopher M. Bishop and Nasser M. Nasrabadi. Pattern recognition and machine learning. *J. Electronic Imaging*, 16(4), 2007. 97

[169] François-Xavier Standaert and Cédric Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 411–425, Washington, D.C., USA, August 10-13 2008. Springer. 97

[170] Christian Rechberger and Elisabeth Oswald. Practical template attacks. In Chae Hoon Lim and Moti Yung, editors, *5th International Workshop on Information Security Applications (WISA)*, volume 3325 of *Lecture Notes in Computer Science*, pages 440–456. Springer, August 23-25 2004. 97

[171] Allam Mousa and Ahmad Hamad. Evaluation of the RC4 algorithm for data encryption. *IJCSA*, 3(2):44–56, 2006. 97

[172] Daniel Larose. *k-Nearest Neighbor Algorithm*, pages 90–106. John Wiley & Sons, Inc., 2005. 99

[173] Liwei Wang, Yan Zhang, and Jufu Feng. On the Euclidean distance of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8):1334–1339, 2005. 100

[174] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer, 2009. 100

[175] Yanet Rodriguez, Bernard De Baets, Maria Garcia, Carlos Morell, and Ricardo Grau. A correlation-based distance function for nearest neighbor classification. In Jos Ruiz-Shulcloper and Walter Kropatsch, editors, *Image Analysis and Applications Progress in Pattern Recognition*, volume 5197 of *Lecture Notes in Computer Science*, pages 284 – 291. Springer, 2008. 100

[176] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor,

EUROCRYPT, volume 1403 of *Lecture Notes in Computer Science*, pages 236–250, Espoo, Finland, May 31 - June 4 1998. Springer. 101

[177] Jean-Sébastien Coron and David Naccache. On the security of RSA screening. In Hideki Imai and Yuliang Zheng, editors, *Public Key Cryptography*, volume 1560 of *Lecture Notes in Computer Science*, pages 197–203, Kamakura, Japan, March 1-3 1999. Springer. 101

[178] Web site. Tutorial for learning assembly language for the AVR-Single-Chip-Processors, [visited] October 2013. http://www.avr-asm-tutorial.net/avr_en/. 105

[179] Web site. AVR freaks, [visited] October 2013. http://www.avrfreaks.net/. 105

[180] Teledyne LeCroy. Teledyne LeCroy website, Visited February 2013. http://www.teledynelecroy.com. 105

[181] Pomona Electronics. 6069A scope probe, website visited October 2012. www.pomonaelectronics.com/pdf/d4550b-sp150b_6_01.pdf. 105

[182] k-fold cross validation, (Accessed) 2016. http://www.csie.ntu.edu.tw/~b92109/course/Machine%20Learning/Cross-Validation.pdf. 110

[183] Teuvo Kohenen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, January 1982. 112

[184] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995. 112

[185] Teuvo Kohenen. *Learning Vector Quantization*. Springer, 2001. 113

[186] Irina Rish. An empirical study of the naive bayes classifier. *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 3(22):41–46, August 2001. 113

[187] Deutsche Bank AG and Contributors. Cryptool 1-4-31, Downloaded, May 2013. http://www.cryptool.org/en/jct-downloads-en. 113

[188] National Institute of Standards and Technology. FIPS 180-2, secure hash standard, federal information processing standard (FIPS), publication 180-2. Technical report, Department Of Commerce, April 1995. http://www.itl.nist.gov/fipspubs/fip180-1.htm. 114

[189] Ronald Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. `http://tools.ietf.org/pdf/rfc1321.pdf`. 114

[190] Microchip. Pic16f87x, (Accessed) 2015. `http://ww1.microchip.com/downloads/en/DeviceDoc/30292D.pdf`. 115

[191] A. N. Myamlin and V. K. Smirnov. Computer with stack memory. In *IFIP Congress (2)*, pages 818–823, 1968. 119

[192] Murat Balaban. Buffer overflows demystified, (Visited) 2015. `http://www.enderunix.org/docs/en/bof-eng.txt`. 119

[193] Aleph One. Smashing the stack for fun and profit, (Visited) 2015. Published on Phrack 49, V. Seven. 119

[194] Loic Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. Power supply glitch induced faults on fpga: An in-depth analysis of the injection mechanism. In *IEEE 19th International On-Line Testing Symposium (IOLTS)*, pages 110–115. IEEE, July 8-10 2013. 119

[195] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, August 13-15 2002. 119

[196] Alfredo Benso, Paolo Prinetto, Maurizio Rebaudengo, and Matteo Sonza Reorda. Exfi: a low-cost fault injection system for embedded microprocessor-based boards. *ACM Transactions Design Automation Electronic Systems*, 3(4):626–634, 1998. 119

[197] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical optical fault injection on secure microcontrollers. In Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi,

editors, *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 91–99. IEEE, September 29 2011. 119

[198] National Institute of Standards and Technology. Data encryption standard (DES), publication 46-3. Technical report, Department Of Commerce, [Reaffirmed] October 1999. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`. 120

[199] National Institute of Standards and Technology. Advanced encryption standard (AES), publication 197. Technical report, Department Of Commerce, November 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. 120, 131

[200] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java card operand stack: Fault attacks, combined attacks and countermeasures. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 297–313. Springer, September 14-16 2011. 121, 122

[201] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 international conference*, volume 6035 of *Lecture Notes in Computer Science*, pages 148–163. Springer, April 14-16 2010. 121

[202] Julien Lancia. Java card combined attacks with localization-agnostic fault injection. In Stefan Mangard, editor, *CARDIS*, volume 7771 of *Lecture Notes in Computer Science*, pages 31–45. Springer, November 28-30 2012. 121

[203] Eric Vétillard and Anthony Ferrari. Combined attacks and countermeasures. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application (CARDIS '10)*, volume 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer, April 14-16 2010. 121

[204] Solar Designer. Non-executable user stack. http://www.false.com/security/linux-stack/. 122

[205] Common Criteria. Application of attack potential to smartcards, March 2009. http://www.commoncriteriaportal.org/files/supdocs/CCDB-2009-03-001.pdf. 123

[206] Tristan Gingold. GHDL home page, (Visited) 2014. http://home.gna.org/ghdl/. 128

[207] Philip Koopman. Embedded system security. *IEEE Computer*, 37(7):95–97, 2004. 136

[208] Bruce Schneier. *Applied Cryptography: Protocols, algorithms and source code in C.* Wiley, 1994. 136

[209] William Stallings. *Cryptographic and network security.* Pearson, 1999. 136

[210] Microsoft. Data origin authentication, (Accessed) 2015. https://msdn.microsoft.com/en-us/library/ff648434.aspx. 136

[211] Shufu Mao and Tilman Wolf. Hardware support for secure processing in embedded systems. *IEEE Trans. Computers*, 59(6):847–854, 2010. 136

[212] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Trans. VLSI Syst.*, 14(12):1295–1308, 2006. 136

[213] Aleksandar Milenkovic, Milena Milenkovic, and Emil Jovanov. An efficient runtime instruction block verification for secure embedded systems. *J. Embedded Computing*, 2(1):57–76, 2006. 137

[214] Krutartha Patel, Sridevan Parameswaran, and Seng Lin Shee. Ensuring secure program execution in multiprocessor embedded systems: a case study. In Soonhoi Ha, Kiyoung Choi, Nikil D. Dutt, and Jürgen Teich, editors, *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS*, pages 57–62, Salzburg, Austria, 2007. ACM. 137

[215] Atul Verma. Get into the zone: building secure systems with arm trustzone technology, (Accessed) 2014. 137

[216] Jérémie Crenne, Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Russell Tessier, and Deepak Unnikrishnan. Configurable memory security in embedded systems. *ACM Trans. Embedded Comput. Syst.*, 12(3):71, 2013. 137

[217] Antonio Kung. Enabling trust for safety and security in embedded systems. *ITEA2 Megazine*, (8), 2013. 137

[218] Herbert Bos, Bart Samwel, Mihai-Lucian Cristea, and Kostas Anagnostakis. Safe execution of untrusted applications on embedded network processors. *IJES*, 3(4):294–303, 2008. 137

[219] Bart Preneel. Davies-meyer. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 312–313. Springer, 2011. 146

[220] John H. Shaffer. Designing very large content-addressable memories, 1992. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.951`. 148

[221] Juergen Saurmann. How to design your own CPU on FPGAs with VHDL, (Accessed) 2015. `http://opencores.org/ocsvn/cpu_lecture/cpu_lecture`. 156

Appendix A

Selected AVR Instructions

Instruction	Description	Operation	Clock Cycles
mov R_d, R_r	Move a byte between registers	$R_d \leftarrow R_r$	1
movw R_d, R_r	Move a word between registers	$R_{d+1} : R_d \leftarrow R_{r+1} : R_r$	1
ldi R_d, K	Load immediate into register	$R_d \leftarrow K, 0 \leq d \leq 31,$	1
ld R_d, P	Load indirect	$R_d \leftarrow (P)$ $P \in \{X, Y, Z\}, 0 \leq d \leq 31$	2
ld $R_d, P+$	Load indirect with post-increment	$R_d \leftarrow (P)$, $P \leftarrow P + 1$ $P \in \{X, Y, Z\}, 0 \leq d \leq 31$	2
ld $R_d, -P$	Load indirect with pre-decrement	$P \leftarrow P - 1$, $R_d \leftarrow (P)$ $P \in \{X, Y, Z\}, 0 \leq d \leq 31$	2
lds R_d, k	Load direct from SRAM	$R_d \leftarrow (k)$ $0 \leq d \leq 31, 0 \leq k \leq 65535$	2
st P, R_r	Store indirect	$(P) \leftarrow R_r$ $P \in \{X, Y, Z\}, 0 \leq r \leq 31$	2
st $P+, R_r$	Store indirect with post-increment	$(P) \leftarrow R_r$, $P \leftarrow P + 1$ $P \in \{X, Y, Z\}, 0 \leq r \leq 31$	2
st $-P, R_r$	Store indirect with pre-decrement	$P \leftarrow P - 1$, $(P) \leftarrow R_r$ $P \in \{X, Y, Z\}, 0 \leq r \leq 31$	2

sts k, R_r	Store direct into SRAM	$(k) \leftarrow R_d$ $0 \leq r \leq 31, 0 \leq k \leq 65535$	2
lpm R_d, P	Load program memory	$R_d \leftarrow (P)$ $0 \leq d \leq 31, P \in \{Z, Z+\}$	3
spm	Store program memory		4
in R_d, P_t	In port	$(P_t) \leftarrow R_d$	1
out P_t, R_r	Out port	$R_r \leftarrow (P_t)$	1
push R_r	Push register on stack	$STACK \leftarrow R_d$	2
pop R_d	Pop register from stack	$R_d \leftarrow STACK$	2
nop R_d, R_r	Do nothing		1
add R_d, R_r	Add two registers	$R_d \leftarrow R_d + R_r$	1
adc R_d, R_r	Add two registers with carry	$R_d \leftarrow R_d + R_r + C$	1
adiw R_d, K	Add register with immediate word	$R_{d+1} : R_d \leftarrow R_{d+1} : R_d + K$	2
sub R_d, R_r	Subtract two registers	$R_d \leftarrow R_d - R_r$	1
sbc R_d, R_r	Subtract two registers with carry	$R_d \leftarrow R_d - R_r - C$	1
sbiw R_d, K	Subtract immediate from a word stored in consecutive registers	$R_{d+1} : R_d \leftarrow R_{d+1} : R_d - K$	2
mul R_d, R_r	Multiply two registers	$R_d \leftarrow R_d \times R_r$	2
eor R_d, R_r	Exclusive or two registers	$R_d \leftarrow R_d \oplus R_r$	1
inc R_d	Increment a register	$R_d \leftarrow R_d + 1$	1
dec R_d	Decrement a register	$R_d \leftarrow R_d - 1$	1
clr R_d	Clear a register	$R_d \leftarrow R_d \oplus R_d$	1
cpi R_d, K	Compare immediate with a value	$R_d - K$	1

	stored in the given register		
cp R_d, R_r	Compare two registers	$R_d - R_r$	1
cpc R_d, R_r	Compare two registers with carry	$R_d - R_r - C$	1
rjmp k	Relative jump	$PC \leftarrow PC + k + 1$	2
jmp k	Direct jump	$PC \leftarrow k$	3
rcall k	Relative subroutine call	$PC \leftarrow PC + k + 1$	3
call k	Direct subroutine call	$PC \leftarrow k$	4
ret k	Subroutine return	$PC \leftarrow STACK$	4
breq k	Branch if equal	if $(Z = 1)$ then $PC \leftarrow PC + k + 1$	1/2
brne k	Branch if not equal	if $(Z = 0)$ then $PC \leftarrow PC + k + 1$	1/2
brcs k	Branch if carry is set	if $(C = 1)$ then $PC \leftarrow PC + k + 1$	1/2
brcc k	Branch if carry is clear	if $(C = 0)$ then $PC \leftarrow PC + k + 1$	1/2
brbc b, k	Branch if flag is clear	if $(SREG(b) = 0)$ then $PC \leftarrow PC + k + 1$	1/2
brbs b, k	Branch if flag is set	if $(SREG(b) = 1)$ then $PC \leftarrow PC + k + 1$	1/2

Table A.1: AVR's instructions seleted for our experiments.