Towards Unclonable System Design for Resource-Constrained Applications

by

Md Jubayer al Mahmod

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
August 3, 2019

Keywords: Unclonable system, IoT security, firmware obfuscation

Copyright 2019 by Md Jubayer al Mahmod

Approved by

Ujjwal Guin, Chair, Assistant Professor of Electrical and Computer Engineering
Vishwani D. Agrawal, Professor Emeritus of Electrical and Computer Engineering
Adit D. Singh, James B. Davis Professor of Electrical and Computer Engineering

Abstract

As we forge ahead to achieve the targeted connectivity among devices in this Internet of Things (IoT) era, reliability and security of individual device have become a matter of paramount importance. Cloned electronic devices are threats because they can lead to a major security breach in a system. Besides, these counterfeit devices can be unreliable as they might have been manufactured with subpar materials and may have defects due to lack of full-fledged testing. Protecting a device from being cloned is therefore undeniably important. In this thesis, we present a novel firmware obfuscation method that, in association with hardware assistance, can effectively prevent an electronic system from being cloned. The firmware is obfuscated by swapping a subset of instructions, and the instructions to be swapped are specifically chosen so that an attacker cannot discover their locations. The obfuscated firmware is dynamically reconstructed during execution by a small cache and PUF-generated ID. The cache contains swapped instructions and their relative addresses. An adversary cannot make a program work completely without knowing which instructions have been swapped, as the program will execute in a wrong sequence and produce incorrect results. That is, firmware cannot be reconstructed without the proper ID of the system because the correct execution flow is obfuscated. The scheme does not increase the size of the code and requires only a small cache in the processor which makes it an effective and practical solution for resource-constrained devices.

Contents

List of Figures

List of Tables

List of Abbreviations

IoT Internet of Things

CPU Central Processing Unit

DoS Denial of Service

IP Intellectual Property

RSA Ron Rivest, Adi Shamirand, Leonard Adleman

UART Universal Asynchronous Receiver/Transmitter

SRAM Static Random Access Memory

JTAG Joint Test Action Group

I/O Input/Output

Chapter 1

Introduction

Preventing electronic components from being counterfeited is one of the major and long-standing challenges in this century. Every abstraction level of a design starting from IP to a comprehensive system is threatened by the counterfeiters. The difficulty of stopping counterfeit is increasing in both magnitude and dimension due to the ever-increasing intricacies in the semiconductor supply chain [1,2]. IP, IC, and system all are susceptible to counterfeit. An IP can be overused, pirated or illegally modified whereas an IC can be overproduced in a foundry, cloned, remarked, or recycled as new. Apart from revenue loss by the designers, the counterfeit electronics also brings about security and reliability issues [3]. This is true because a counterfeit component such as a cloned microprocessor may not be produced using standard material, or it may not be thoroughly tested. It is also possible that a component or a whole system may have illegal modifications that could lead to potential security breaches. It stands to reason that the presence of illegitimate devices in a mission-critical application could lead to catastrophic consequences [4].

As the global semiconductor supply chain becomes more and more distributed, counterfeit detection and avoidance are becoming increasingly complicated [5]. Modern horizontal chip manufacturing makes it very convenient for the fabless design house to outsource the chip manufacturing process to globally distributed foundries. This puts foundries in the untrusted zone from the designers' point of view. It would not be uncommon for foundries to overproduce chip than they have been contracted for [6–8] or to sell GDSII files illegally to another design house. The primary motivation for the foundry to overproduce chips is that it does not incur any R&D cost for the chip but still gets to profit directly of the chip. By simply reporting a lower yield to the SoC designers, the foundry can overbuild the

chips with little additional cost. Researchers have expended significant effort to address this IC overproduction issue. Koushnafar proposed a PUF-based active metering technique [9]. Methods such as Secure Split Test (SST) [10] and Connecticut Secure Split Test (CSST) impose security features in the post-manufacturing test data. Authors in [11] proposed a method called *EPIC* that employs an on-chip TRNG to generate RSA key pairs. Guin et al. applied pretty good privacy (PGP) to prevent IC from overproduction [12]. The major disadvantage of these methods is back-and-forth communication between the foundry and SoC design house and large design overhead. Recycled ICs sold as new is another problem that the semiconductor industry is struggling to solve. In general, we can detect an IC whether it is recycled or not by two methods. First, a test method can be developed to detect if an IC is recycled or not [13, 14]. For instance, creating a statistical model that can serve as DNA marking is one way to do that [15–18]. The complications of developing a test scheme come from many different angles including test time, cost, and low detection accuracy [19]. Second, we can design an anti-counterfeit scheme that can detect the age of an IC [3, 20, 21]. However, this approach of recycled IC detection cannot be used for already manufactured ICs. Similar to overproduction and recycling, other counterfeit types such as remarking, cloning, and tempering are also difficult to detect. As mentioned above, distributed semiconductor system design and manufacturing opened a door for Intellectual Property (IP) cloning, tampering, and overuse. It is possible that untrusted SoC designers might clone an IP purchased from a third party IP vendor. Also, they can modify or add features without the authorization of the original IP vendor [12]. In both cases, the IP vendor will lose revenue.

1.1 Security in resource-constrained devices

Ensuring the security of resource-constrained devices is essential because of their ubiquitous presence in the internet of things (IoT), smart grid, autonomous vehicles, industrial automation. Gartner predicted that there will be approximately 20 billion connected devices by 2020 [22]. The concern regarding security and privacy is heavily emphasized because these devices might be potential targets for cyber attacks due to the lack of standard security features. Even if the security features are present, sometimes it becomes inefficient to run these power intensive cryptographic operations due to limited available energy.

A typical application of resource-constrained devices is IoT system. Internet of things provides a great benefit in terms of service but, like all other great technological breakthroughs, it brings about many negative aspects too. The security breach is one of those problematic issues that an IoT system is prone to. Failure to implement standard cryptographic operations in such resource-constrained devices will lead to a large number of unsecured and counterfeit equipments connected to the Internet. In 2016, Dyn, a DNS service provider, faced an unprecedented distributed denial of service attack through a bot-net consisting of IoT devices such as a printer, IP camera, and residential gateway [27]. Therefore, immediate intervention is essential for ensuring the security of these wide varieties of devices.

It is difficult to implement security features in an IoT device due to its lack of resources. A typical example of a resource-constrained device is a microcontroller that is equipped with 256KB flash memory and 1KB RAM. Table 1.1 lists specification of some resource-constrained devices. These devices are designed to operate with a very limited energy supply.

Table 1.1: Example resource-constrained edge device specifications

Device	CPU	Clock Frequency	RAM	ROM	Bandwidth	Protocol
Arduino Uno [23]	8 bit	16MHz	2KB	32KB	250kbs	IEEE 802.15.4
Z1-Mote [24]	32 bit	32MHz	32KB	512KB	250kbs	IEEE 802.15.4
T-Mote Sky [25]	16bit	8MHz	10KB	48KB	250kbs	IEEE 802.15.4
Openmote [26]	32bit	32MHz	32KB	512KB	250kbs	IEEE 802.15.4

A majority of these devices do not use standard cryptographic protocols to ensure secure operations because of these severe resource constraints [28–31]. Therefore, these devices are exposed to firmware and software piracy, and data storage threat.

As mentioned previously, providing security to resource-constrained devices is an intricate problem to solve because of the following reasons:

1. The battery-driven devices are not equipped with high-speed CPU that is an essential requirement for mathematically intensive cryptographic operations. Specifically, providing in-filed security is rather difficult for the devices that are designed to run in low energy supply such as energy harvested devices* [32].

2. Compared to phones and laptops IoT devices do not have large volatile or non-volatile memory. Large memory is important because conventional security schemes are not designed for low memory operation [33].

3. In-field installation of security patches is difficult for the IoT device because it may not be equipped with secure reception and integration capability.

4. The IoT network is composed of many different types of devices. It is difficult to find a comprehensive solution that could address a specific security problem for a massive heterogeneous network.

The attack surface for the resource-constrained devices is quite large because security threat exists in every abstraction level of a design. It is possible that security could be breached from a transistor level design (e.g., Trojan, IP piracy) to system level (e.g., cloning, tampering, etc). This thesis focuses on secure hardware development that provides strong resistance against cloning. Both from the perspective of users and manufacturers, cloning is an important concern. For a user, the presence of cloned devices in a system is a threat because it can act as a backdoor to the home network and ultimately lead to malicious

*Energy harvested devices rely on natural energy sources,e.g., solar power. These devices are inherently designed to operate in a limited power budget.

private information access by an adversary. For a manufacturer, cloning could lead to a huge profit loss.

1.2 Cloning

The hardware and the firmware running on a resource-constrained system are exposed to piracy. An untrusted entity in the supply chain can clone both the hardware and firmware, source them to an untrusted system integrator, and create clones. Any cloned system may have backdoors, which can be exploited for malicious purposes [34, 35]. A recent report from Bloomberg Businessweek revealed that China used a tiny chip, which is not larger than a grain of rice to infiltrate 30 U.S. companies, including Apple and Amazon [36]. The compromised servers were assembled for Elemental Technologies by Super Micro Computer Inc., which is a San Jose-based company and the biggest suppliers of server motherboards for data centers. The report mentioned that the microchips were inserted at Chinese factories, and then supplied to Supermicro. According to Bloomberg, Elemental's servers could be found in the Department of Defense (DoD) data centers, the CIA's drone operations, and the onboard networks of Navy warships. The report also mentioned that an adversary can gain control of the compromised system when the server is switched on and the microchip inserts malicious codes to alter the operating system's core.

An adversary can perform cloning by retrieving a copy of the firmware from an embedded device [37]. It is practically infeasible to develop a cloned product from the original specification as it requires significant investment in the research and development (R&D), what an adversary is unwilling to invest. An easier way of making clones is to illegally obtain a pirated copy of the design. An adversary can also perform reverse engineering, which is a process of extracting the design specification of the inner details of a product [38]. Cloning an electronic system requires the complete reconstruction of the hardware and the firmware. Recently, the hardware becomes increasingly vulnerable to RE due to the availability of very advanced imaging instruments and powerful characterization tools [38]. Similarly, the

firmware can also be easily extracted from an authentic device. The primary challenges for developing a system, which is resistant to cloning, is twofold. First, one needs to design either secure hardware or firmware, so that an adversary cannot perform RE. Second, the solution needs to be low cost, and low resource overhead (area, and power) to be widely accepted to the various IoT and CPS applications.

We address cloning of resource-constrained devices by binding firmware with hardware. The control follow of a firmware is obfuscated in such a way that it can only be reconstructed by unique hardware. In addition, we present a novel low-cost firmware obfuscation method to effectively detect cloned systems. The firmware is obfuscated using reordering of the few selected instructions. The original flow of the instructions is scrambled using an efficient algorithm to obfuscate the correct execution flow of the firmware. The proposed algorithm selects one instruction and looks for a set of instructions that can be swappable so that no errors are observed, and the program produces an incorrect result. The selection of instructions is performed based on a set of rules. The relative addresses of these two swapped instructions are concealed using an identifier (ID) generated from a physically unclonable function (PUF) and a unique key programmed into a tamper-proof nonvolatile memory (NVM). The dynamic reconstruction of the firmware is assisted by a reorder cache. During power-up, the bootloader of a device reads all the instructions from the memory and loads the swapped instructions in the reorder cache. During the execution of a program, the swapped instructions are recovered from the cache. Note that our proposed solution does not prevent an adversary from copying the firmware, rather than making it operational completely, and provide adequate protection against cloning. We show that it is infeasible to reconstruct the original firmware by an adversary considering the current computing resources, which makes our scheme well-fitted in secure IoT and CPS applications.

1.3 Motivation

Firmware can be extracted from a low-cost embedded device using a regular computer and a low-cost microcontroller. To demonstrate the need for an efficient and robust scheme to counter firmware extraction, we perform an attack proposed by Obermaier et al. [39] on an Arm-based system. We focus our attack on The Arm® Cortex®-M4-based STM32F4 high-performance microcontroller [40]. We present different ways, which help an attacker to easily access the firmware stored on the device.

The microcontroller STM32F4 has two levels of on-chip memory protection to defend against firmware extraction. When these protections are deactivated, anyone with access to the debugging interface can access the flash memory. When the first level is activated, it allows the debugger to be connected, but it locks the debug interface if there is a flash memory access. This first level (Level-I) of protection can be deactivated, but the flash memory gets erased once it is deactivated, supposedly preventing an attacker from extracting the firmware. The second level (Level-II) is an irreversible lock, which disables the debug interface entirely, only allowing the processor core to access flash memory. Obermaier et al. demonstrated different attacks on STM32F0, a predecessor to STM32F4, to bypass these protections. We use these attacks to show how an attacker can access the firmware of the STM32F4 at any level of protection. Note that if the memory protection on the system is deactivated, then an attacker can very easily extract the firmware. Without the protections, flash memory accesses can occur through the debugging interface, making it very easy to read the firmware. Any connection to the device's JTAG interface is able to retrieve the full contents of memory.

If Level-I protection is active, the proposed attack is to focus instead on reading the data in SRAM. While the memory protection locks the debugging interface during a flash read, it was reported in [39] that it does not prevent someone from reading the SRAM. This leads to an attack on any device that loads instructions into SRAM, such as when a device is performing a cyclic redundancy check (CRC) to check firmware integrity. As the program

runs, it loads instructions into SRAM, allowing an attacker to read the instructions as they are checked. We have successfully recreated this attack on the STM32F4.

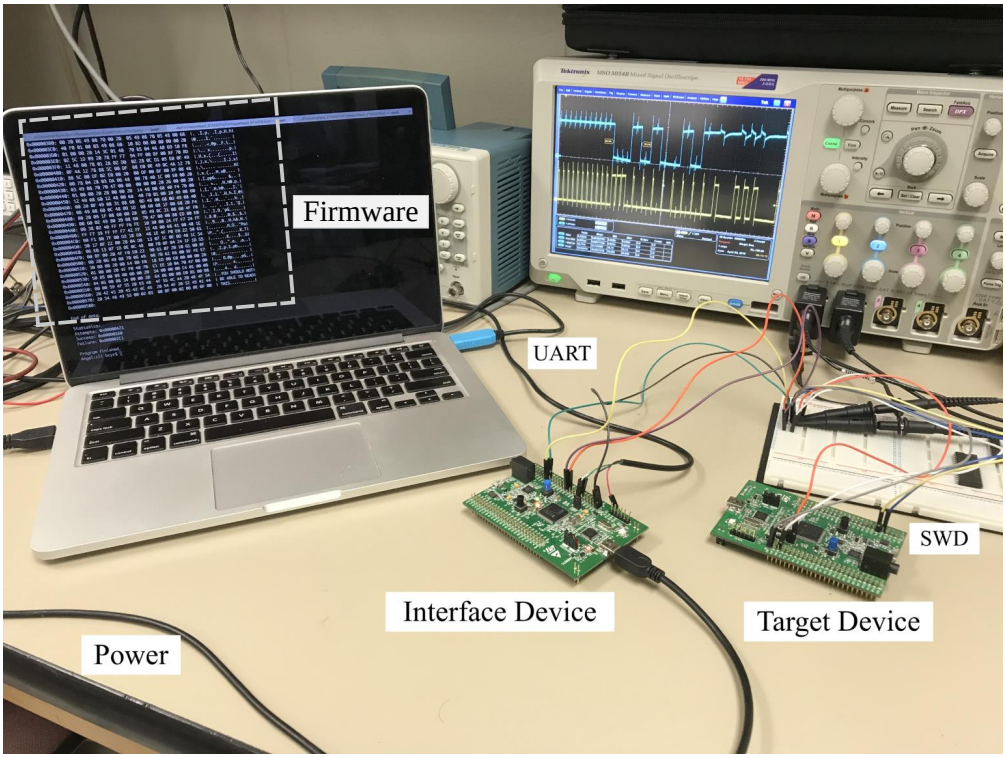

Figure 1.1: Experimental setup to extract the firmware from STM32F4

Figure 1.1 shows the experimental setup to launch this attack. A microcontroller acts as an interface device, which is programmed with the UART module and a driver for the Serial Wire Debug (SWD) interface. The Interface device connects to the target device using the SWD, and controls both the target device's power and reset connections. The interface device reads the SRAM while the target device is performing the CRC, controlling the power to the target device and resetting when necessary. A python script running on the laptop communicates over UART with the interface device, and the SRAM snapshots can be sent to the laptop to extract the firmware. Even with the Level-I firmware protection, we can extract the firmware using this simple measurement.

The attack on Level-II protection is an invasive attack on the microcontroller. After decapsulation, precise UV light is applied to reprogram memory protection bits [39]. Once the memory protection bits are reprogrammed down to Level-I, the above attack can extract

the firmware from the device. No matter what level of protection is used, an adversary can directly access the device firmware. The current memory protections in place for these smaller systems are not enough to prevent an attacker from cloning the firmware. Note that disabling the debug port severely limits the troubleshooting capability of an authentic user, and highly discouraged. Based on the above discussion, it can be concluded that cloning a resource constraint device is rather uncomplicated. Therefore, a new method to provide clone-resistance is essential.

1.4 Contributions

We propose a novel and low-cost method of firmware obfuscation that does not require standard cryptographic methods to protect the firmware against piracy. We identify a few selected instructions from the firmware and reorder them in such a way that it functions incorrectly without letting the attacker know which instructions have been moved. If an adversary downloads the firmware directly from the non-volatile memory (NVM) and runs it on a different device, those selected instructions will execute out of order, causing the program to produce incorrect results.

Only the devices that are authenticated by the manufacturer can reconstruct the firmware at boot time. The devices use a unique device identifier (ID) that can be generated from a PUF such as an on-chip SRAM-PUF [41]. During boot-up, the device uses its ID and a stored program key to generate an obfuscation key, which consists of the relative addresses of the swapped instructions. A bootloader reconstructs the original firmware by storing the swapped instructions in a small cache (we call this a reorder cache). After boot-up, the processor begins fetching instructions from memory like normal, except when there is a hit in the cache. The cache hit would steer the instruction fetching away from the memory to the cache. Therefore, the processor will execute the instruction fetched from the cache instead of the memory. However, the processor still accesses the memory even if it executes the instructions inside the cache. When the device needs an update, the manufacturer can

send the obfuscated firmware update with a new obfuscation key. The device uses the new obfuscation key to reconstruct the updated firmware, allowing it to transition to the updated firmware with a simple reboot.

This solution serves as a low-cost alternative to the existing system-level cloning prevention techniques. Our proposed solution does not require expensive run-time overhead like encryption/decryption. Once the on-chip reorder cache is populated at boot time, there is no extra processing required to execute the firmware. This makes it very practical for IoT/CPS and other small devices with strict resource constraints. While the instructions are still unencrypted and visible to the attacker, it is still very difficult to locate the moved instructions and reorder them to the correct arrangement. The complexity of estimating the correct sequence is $O(N^L)$, where N is the number of instructions from which L pairs of instructions are reordered. Note that for a reasonable size firmware ($\approx 1,000$ instructions), with a small number of swaps (≈ 16), an adversary needs to try approximately 2^{200} trials to make the program completely working, which is infeasible with current computing resources.

1.5 Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 describes related prior works, and Chapter 3 describes our proposed obfuscation and reconstruction methodology. The security evaluation of the proposed approach are presented in Chapter 4. We discussed a hardware implementation scheme of the dynamic firmware reconstruction method in Chapter 5. We discuss potential research scope in Chapter 6 and conclude this thesis.

Chapter 2

Background and Related Work

In this chapter, we discuss a few fundamentals that are essential for understanding the core concept of this thesis. Then, we provide a comprehensive survey of the relevant literature.

2.1 Background

In this section, we discuss the PUF and its application as a hardware *root of trust* and introduce the basic of firmware obfuscation and reconstruction. Then, we introduce system-level mutual authentication that can be used to make an electronic system unclonable.

2.1.1 Physically Unclonable Function (PUF)

PUF is a popular hardware security primitive that can be used as a root of trust in a system. PUF is designed exploiting variation of the physical properties, for example, delay of paths in a circuit, power-up states of memory cells, frequency of ring oscillators, the spectral variation of optical devices. etc. Instead of using NVM for storing keys, we can use PUF for unique key generation and authentication [42]. For instance, when SRAM power-up, the memory cells contain random values. Ideally, the uninitialized value of a particular cell should be 1 or 0 with 50% probability due to the symmetric structure of an SRAM. However, the variation (e.g., length/width, oxide thickness of a transistor) during manufacturing will force some cells to be biased towards 1s and some other towards 0s. Therefore, it is impossible to know the power-up state of an SRAM from circuit design. That is, even though the design is exact same, different SRAM device will have different power-up states. This variation can

be used to identify each device uniquely. Since the physical variation is completely random, the device cannot be cloned using a feasible method.

2.1.2 Firmware obfuscation

In simple term, hiding the functionality and correct execution flow of firmware is obfuscation. We can obfuscate a firmware by hiding its control flow. Note that, control flow is the order of execution of statements or instructions in a program. If the normal flow of firmware is changed in such a way that an attacker cannot devise the correct execution order of the firmware, it is an obfuscated firmware.

2.1.3 Firmware reconstruction

An obfuscated firmware needs to be reconstructed before it can be executed by a processor. Reconstruction method typically consists of a secret (e.g., keys) which is used to obfuscate a firmware. The reconstruction can be entirely software-based or hardware-based. Also, a software-based method can be augmented with hardware assistance. The reconstruction method presented in this work is based on both software and hardware. The software runs the initial phase of reconstruction till a checkpoint during power-up and then leaves the rest of the operation to the hardware during execution of the firmware.

2.1.4 System-level Mutual Authentication (SMA)

SMA was proposed in [37] as an effective method to counter cloning. The core concept of this method is that a system will be authenticated if its firmware gets authenticated by the hardware and vice versa. A firmware is only executable by a specific hardware and so a non-authentic hardware will not be able to run this particular firmware. One the other hand, a hardware is only functional by the firmware that was specifically designed for this hardware. In other words, a non-authentic firmware will not be able to use the hardware.

Ultimately, a system becomes unclonable because the correct execution of its firmware is only possible if it runs on this particular system's (hence legitimate) hardware.

One can develop this method by binding (i.e., obfuscating or encrypting) a firmware with the signature of a hardware. Correct execution of the obfuscated firmware would mean that the device signature is valid. On the other hand, a valid device signature would automatically lead to correct execution of legitimate firmware. Therefore, hardware and firmware mutually authenticate each other to provide a seamless operation of a system. A cloned firmware has no use in an illegitimate device, and a cloned hardware would not have legitimate device signature to ascertain the control flow of a firmware.

2.2 Related work

Researchers have presented numerous solutions to protect both hardware and firmware. The protection of hardware can be ensured cost-effectively by the verification of an unclonable identification number (ID) created from the hardware fingerprint [43–46]. However, a cost-effective solution needs to be developed to protect firmware from piracy, especially from copying or cloning. A variety of solutions have been proposed over the years to protect firmware from various attacks. Li et al. proposed the integrity verification of peripherals' firmware of a computer system by using remote software-based attestation [47]. LeMay et al. developed Cumulative Attestation Kernel (CAK) to verify the integrity of the firmware over an interval of time [48]. The solution provides the cumulative attestation for memory constraint devices by adding a Cumulative Attestation Coprocessor (CAC) that handles the computation and storage. To safeguard the firmware against non-invasive attacks, Schellekens et al. proposed a solution to protect the persistent state of a trusted module by maintaining an authenticated channel between the trusted module and the memory [49]. Maskiewicz et al. proposed a signature verification scheme to prevent the installation of malicious firmware on a mouse [50]. Morais et al. developed a solution that uses integrity verification at different levels of the boot-up process to ensure the loading of proper firmware into the memory [51].

Chakraborty et al. proposed a key-based control flow obfuscation based on a sequential unlocking mechanism to protect piracy and malicious modification to the embedded software [52]. This solution requires a code overhead up to 10%, and the instructions used for validation need to be hidden from an adversary. Zhuang et al. developed a hardware-assisted control flow obfuscation, which relies on additional hardware such as shuffle buffer and block address table cache [53]. There are several implementations of Oblivious RAMs proposed by Goldreich et al. (ORAM) [54], which obfuscate control flow and patterns of memory accesses [55, 56]. There are also a few designs that have been proposed for FPGAs which encrypt the firmware with PUF-generated keys [57, 58]. One other potential solution was proposed by Guin et al., where mutual authentication is performed to prevent system-level cloning [37]. In this approach, the hardware verifies the firmware and the firmware ensures the authenticity of the hardware. The firmware is obfuscated by removing a select number of instructions such that the firmware is inoperable. This method requires the entire firmware reconstruction during the powerup stage, where the reconstructed firmware must be kept in the volatile memory (e.g., SRAM or DRAM) during execution. It is often challenging to store the entire firmware in the memory for resource-constrained devices, as many of these embedded devices may not possess an on-chip SRAM or an off-chip DRAM.

Lee et al. proposed a hardware and software codependent anti-cloning scheme [59]. Here, the authors proposed to obfuscate each instruction I_i with a response from a PUF or a block cipher function F. The memory stores the obfuscated version of instructions as $I_i^{'} = I_i \oplus F(C_i)$. The function F has to be evaluated for a particular challenge C_i for each of the instructions during run-time. Similarly, Zheng et al. [60] proposed to incorporate a device signature during firmware binary generation. Inter-device signature variation makes the firmware uniquely obfuscated for each of the devices. Digitally re-configurable PUF has been employed to counter cloning in [61]. In this method, each of the devices would have a different copy of the obfuscated firmware, and it is bound to specific hardware. The primary limitation of the above methods is excessive timing overhead as each instruction has

to go through a complex and power consuming de-obfuscation/decryption process during execution.

While all of the above solutions can help protect devices from firmware modification and tampering, their applicability in the low-cost, low-power and resource constraint IoT/CPS devices is questionable as the majority of these devices do not use standard cryptographic protocols [28–30]. Integrity and signature verification are often expensive which requires either software support or cryptoprocessor. As these edge devices have limited memory, implementing verification can be infeasible. Moreover, severe energy constraint prohibits IoT edge devices to use standard cryptographic schemes [28]. Signature verification requires additional energy budget, which may pose additional challenges. In addition, adding a coprocessor will significantly increase the cost. Moreover, integrity and signature verification cannot prevent an adversary from copying a firmware. It can be easy for an adversary to tap into the data bus between external memory and the processor and read the firmware. Even devices with on-chip memory and memory protection may not be completely secure from firmware cloning. Recently, Obermaier et al. showed that memory protection can be bypassed by attacks on debug interfaces or even by modifying security bits with UV-C light [39].

In this chapter, we introduced a number of fundamental concepts and surveyed relevant literature that are essential for understanding the work presented in this thesis. We conclude that the existing methods to counter cloning needs improvement even though they provide strong security mainly because of their inefficiencies when it comes to application in the resource-constrained devices.

Chapter 3

Obfuscation and Reconstruction Methodology

The fundamental idea of preventing an adversary from cloning a system is to obfuscate a firmware that runs only on authentic hardware, and an adversary cannot reconstruct the original firmware from control flow analysis. As the existing cloning prevention techniques (e.g., encryption or integrity verification of the firmware) are prohibitively expensive both from the perspectives of development cost and resource consumption during execution in the resource-constrained devices, the industry is in urgent need for a low-cost solution. In this Chapter, we propose a low-cost solution to prevent system-level cloning and discuss the reconstruction method using hardware assistance.

3.1 Firmware obfuscation process

An efficient way of preventing system-level cloning is to add a unique hardware signature to the firmware such that it runs correctly on authentic hardware. We propose to obfuscate the original firmware such that an adversary cannot reconstruct it and works only when the firmware receives an authentic hardware fingerprint. We call this fingerprint as device identification or device ID (ID). The obfuscation is performed by swapping a few selected instructions so that the execution would produce incorrect results. Note that the firmware is not encrypted by using any techniques widely used for data encryption. Our solution is simple and low-cost, which makes it suitable for low-cost IoT and CPS applications.

Figure 3.1 shows the proposed solution to prevent system-level cloning. The trusted system integrator (SI) obfuscates the firmware and loads it into non-volatile memory (e.g., flash memory) of the device. The detailed obfuscation process is described in Algorithm 1. When the firmware is obfuscated, the relative addresses of the swapped instruction pairs are

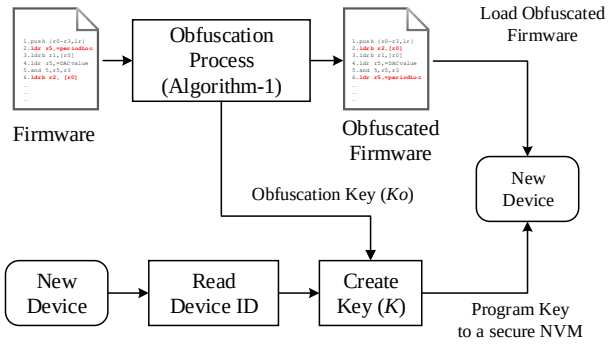

Figure 3.1: Proposed flow for creating a clone-resistant electronic device.

combined into a single key called the obfuscation key (K_O). If the obfuscation has L swaps, then the obfuscation key would be:

$$(Inst_1 \Longleftrightarrow Inst_2),\ \ldots,\ (Inst_{2L-1} \Longleftrightarrow Inst_{2L})$$

$$K_O = [Addr_1, Addr_2, \ldots, Addr_{2L-1}, Addr_{2L}] \tag{3.1}$$

The size of the obfuscation key depends on the address space and the number of instructions used in the swapping process. Since every swap includes two instructions, two addresses need to be stored for each swap. Therefore, the length of the obfuscation key is

$$|K_O| = |Addr_i| \times L \times 2 \tag{3.2}$$

As an example, if there are 32-bit addresses and $L = 16$ swaps, the key length of K_O would be $32 \times 16 \times 2 = 1024$ bits.

To prevent an adversary reconstructing the original firmware by comparing multiple copies of obfuscated firmware, it is necessary to load the same obfuscated copy to all the devices. If an adversary finds multiple copies of the original firmware, he/she can easily launch an attack to find the dissimilar instructions and can reconstruct the original firmware by majority voting. To prevent this attack, we propose to use a single copy of the obfuscated

firmware, which needs to be loaded in all devices. This results in a single obfuscation key (K_O) for every device the trusted SI produces. Programming of this K_O into every device will make an easy target for cloning this key. To prevent this, we propose to derive a unique key (K) from K_O for every device, and then program this derived key K into an electronic device. A physically unclonable function (PUF) [43–45] can be used to derive this K, as a PUF produces an unclonable ID (ID). Since the majority of electronic devices have SRAM-based memory and embedded processors, a stable SRAM PUF [62] can offer a better choice as it does not require any additional cost. To create the key, the trusted SI reads the response (ID) of the SRAM PUF for each device, once it is being tested and becomes defect free. SI then creates K by using the following equation:

$$K = ID \oplus K_O \tag{3.3}$$

Note that once K is programmed into a device, the outside access of PUF responses is disabled. Because of this, each device will have a separate public ID so that the SI can identify each device for the future firmware updates (see Section 3.2). The SI needs to keep a database linking the public ID to the private ID (ID) generated by a PUF.

3.1.1 Algorithm for firmware obfuscation

The proposed obfuscation scheme is a novel way to provide protection to the firmware. Most firmware protection schemes involve encrypting the firmware in some way, but encryption is expensive for embedded applications. It requires special hardware and extra time to decrypt every single instruction from memory. By swapping instructions instead of encrypting them, the firmware is still protected without needing the special decryption hardware, which reduces the cost to implement. The obfuscation, while keeping the majority of the program unchanged, still keeps the firmware secure from cloning because the swapping is done in a way that prevents an attacker from knowing which instructions were swapped.

It is necessary that obfuscated program does not produce any errors during the program compilation so that an adversary finds the swapped instruction simply by debugging. Note that the firmware obfuscation is performed by the trusted system integrator, and only known to it.

Algorithm 1: Firmware obfuscation algorithm

Input : Program (P_E), number of swaps (L_T)
Output : Obfuscated Program and obfuscation key

1 Read the entire program (P_E);
2 Find all valid swappable instructions, $P_{N_T} \leftarrow candidateSwap(P_E)$;
3 Initialize index to 1, $i = 1$;
4 **while** $i \leq L_T$ && $P_{N_T} \neq NULL$ **do**
5 Randomly choose a instruction, $Inst_x$ from P_{N_T};
6 Find all possible instructs to swap, $P_{Inst_x} \leftarrow findPossibleSwaps(P_{N_T}, Inst_x)$;
7 **if** $P_{Inst_x} \neq NULL$ **then**
8 Randomly select one instruction, $Inst_y \in P_{Inst_x}$;
9 Create i^{th} obfuscation key, k_{Oi}, where $k_{Oi} = [RAdd_{Inst_x}\ RAdd_{Inst_y}]$;
10 Update program to include i^{th} swap. $P_E \leftarrow updateProgram(P_E, RAdd_{Inst_x}, RAdd_{Inst_y})$;
11 Drop these two instructions ($Inst_x$, $Inst_y$) from P_{N_T};
12 $i = i + 1$;
13 **end**
14 **else**
15 Drop instruction $Inst_x$ from P_{N_T};
16 $i = i$;
17 **end**
18 **end**
19 Construct obfuscation key, K_O, where $K_O = [k_{O_1}\ k_{O_2}\ \ldots\ k_{O_L}]$;
20 Report obfuscated program, P_E, and obfuscation key, K_O

Algorithm 1 shows the pseudo-code for obfuscating a firmware by swapping a small set of instructions. The algorithm starts with reading all the instructions (E) of a program P (Line 1). It is also necessary to provide the number of swaps (L_T), which is determined based on the size of the device ID and the address bus width as mentioned before. Note that all the instructions of a program cannot be swappable (see details in Section 3.1.2). *candidateSwap*() function stores all swappable instructions to a temporary program variable, P_{N_T} (Line 2). Here, N_T represents the number of instructions that can be swapped. The index (i) for selecting a swap is initialized at Line 2, and the algorithm performs L_T swaps

iteratively (Line 4 - Line 18). In each iteration, an instruction ($Inst_x$) is randomly selected. Note that this instruction cannot be swapped with all $N_T - 1$ instructions (see details in Section 3.1.2). $findPossibleSwaps()$ function returns all possible swaps with $Inst_x$ (Line 6). It is necessary to check whether there is a swappable instruction exists for $Inst_x$. If swappable instructions exist, the algorithm selects one ($Inst_y$) randomly (Line 8). An obfuscation key (k_{O_i}) that represents the relative addresses of these two instructions ($Intr_x$ and $Inst_y$), is created for this swap, where $k_{O_i} = [RAdd_{Inst_x}\ RAdd_{Inst_y}]$ (Line 9). $Intr_x$ and $Inst_y$ are now swapped in the original program, P_E using $updateProgram()$ function (Line 10). The algorithm now drops these two instructions from the temporary program variable, P_{N_T} (Line 11) and increases the index (Line 12). If $findPossibleSwaps()$ function does not find any instruction to swap with $Inst_x$ (Line 6), the algorithm drops $Inst_x$ (Line 15) and keep the index constant. Once the entire program is obfuscated by performing L_T swaps, the complete obfuscation key (K_O) is constructed (Line 19). Finally, the algorithm reports the obfuscated program, P_E, and obfuscation key, K_O (Line 20).

3.1.2 Swapping Rule Check (SRC) for instructions

To ensure the security of our proposed obfuscation method, it is necessary to prevent an attacker from finding out the swapped instructions. The ability to hide a pair of swapped instructions in the obfuscated firmware is dependent on the instruction types and registers used in each instruction. In this section, we propose *Swapping Rule Check (SRC)* to ensure that two instructions (e.g., $Inst_x$ and $Inst_y$) are swappable. The SRC ensures that the swapped instructions in the obfuscated firmware are not obvious to an attacker. Here, we follow ARM assembly language to describe these rules with examples. Note that, instructions are individually checked by the algorithm to examine whether it is swappable with any other instructions or not. Therefore, instruction length variability will not affect the obfuscation technique. Therefore, these rules can be extended to other assembly languages (e.g., x64 [63],

AVR [64], or PIC [65]). However, the cache design will be different for fixed and variable length instruction sets.

The SRC is divided into two sets of rules. The first set determines which instructions are candidates for the swapping algorithm. These rules filter out any instructions that could not be swapped with any other instruction. The second set of rules determines which pairs out of the instruction candidates are indeed swappable. Even among the candidate instructions, only certain pairs can be swapped without tipping off an adversary. These two sets of rules are described in detail below:

• *Set-I: Rules for finding candidates swaps*

These rules define which instructions are candidates for swapping and are implemented in *candidateSwap*() function (see Line 2 of Algorithm 1).

1. *Branches*: Any branch instructions are not allowed for swapping. Branches (e.g., *b*, *beq*, *blt*, *bhi*, etc.) cannot be swapped as it will provide information to an attacker that is dynamically monitoring the memory bus. If the processor branches to an unexpected location because it executed a swapped branch instead of the instruction in memory, it will let the attacker know that the instruction has been swapped.

2. *Function Headers and Footers*: There are instructions that serve as function headers and footers that designate a function block. If these instructions are moved, an attacker will know immediately that a change has occurred. For example, if *halt/return* instructions are misplaced, it will reduce the search space for an attacker. Therefore, the obfuscation algorithm leaves the header and footer of functions in the program.

• *Set-II: Rules for finding pairs*

These rules define which pair of instructions are swappable, and are implemented in $findPossibleSwaps()$ function (see Line 5 of Algorithm 1).

1. *Equivalent Instruction*: Equivalent instructions cannot be swapped as this will not obfuscate the firmware. Two instructions are equivalent if they perform the same function and have the same operands.

2. *Register Initialization*: Instructions cannot be swapped into a location where one of the source registers becomes uninitialized. For example, in Figure 3.2(a) instruction 3 uses register $r5$. Instruction 2 (i.e., *ldr* $r5$, = *periodLoc*) initializes register $r5$. Now, swapping instruction 2 with instruction 6 will give an attacker an indication that instruction 2 has been swapped as $r3$ has no initial value (see Fig. 3.2).

```
1. push {r0-r3, lr}
2. ldr r5, =periodLoc
3. ldrb r1, r5
4. ldr r5, =DACvalue
5. and r3,r2,r5
6. ldrb r0, [r3]
```

Original flow (a)

```
1. push {r0-r3, lr}
2. ldrb r0, [r3]
3. ldrb r1, r5
4. ldr r5, =DACvalue
5. and r3,r2,r5
6. ldr r5, =periodLoc
```

Obfuscated flow (b)

Figure 3.2: ARM assembly code snippet as an example for register initialization. Instructions 2 and 6 cannot be swapped.

3. *Register Utilization*: Instructions cannot be swapped into a location where its destination register is never used. For instance, let us assume that register $r5$ is last used in Line 6 in Figure 3.2(b). If *ldr* instruction in Line 2 in Figure 3.2(a) is swapped with *ldrb* in Line 6, the assignment *ldr* $r5$, = *periodLoc* becomes redundant and will tip off the attacker because the destination register $r5$ has not been utilized in the obfuscated firmware.

4. *Operation Efficacy*: Instructions cannot be swapped into a location where the operation performed is redundant and only extends the operation of the last instruction. For example, assume instructions *sub* $r3$, $r2$, #5 and *add* $r3$, $r3$, #2 have been placed in

consecutive locations after obfuscation. Since, $sub\, r3\,,\, r2,\, \#3$ can replace the previous two instructions. Therefore, one of them is clearly swapped instructions and it narrow downs the search area for an attacker.

5. *Index Distinction*: If the instructions are to be mapped into a cache as described in Section 3.2, part of the instruction address should be reserved as an index into the cache. For example, if a 32-entry direct-mapped cache is used for reconstruction, then 5-bits in the instruction address must be reserved as the index. No two chosen instructions can have the same index, or else there will be a collision in the cache.

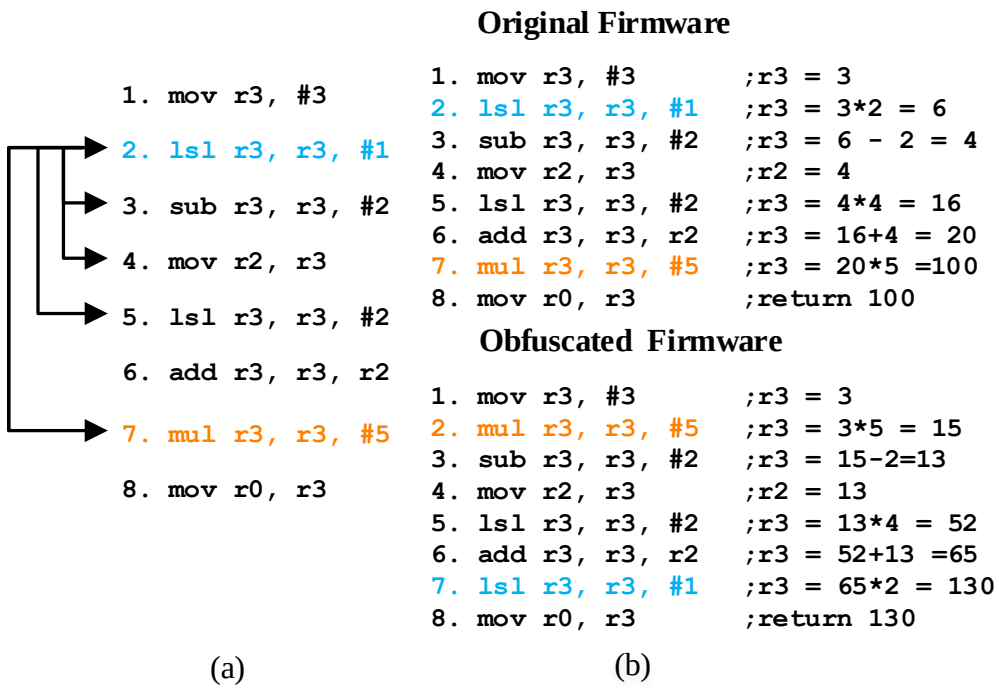

Figure 3.3: A simple obfuscated firmware illustration. A single instruction can swap with many other instructions (a). When a swap is made, it changes the output of the function (b).

A simple and comprehensive example of the obfuscation scheme is illustrated in Figure 3.3. The instructions that usually are not swappable (e.g., branches, push and pops, etc.) have been omitted for simplicity. The eight instructions in the figure can be swapped

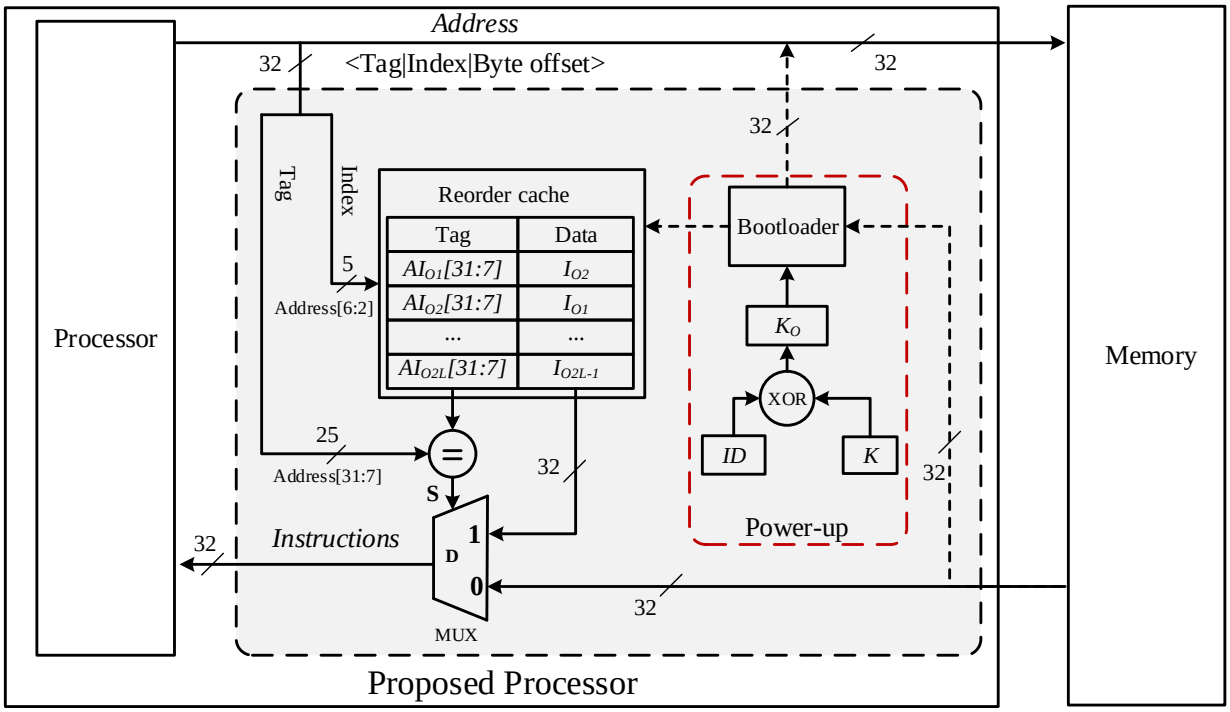

Figure 3.4: Proposed scheme for firmware reconstruction during program execution.

in eleven different ways according to the SRC. If we randomly choose the swappable instruction $ls1\ r3,\ r3,\ r2$ (blue), then we can make four possible swaps that adhere to the SRC (Figure 3.3(a)). We then randomly choose $sub\ r3,\ r3,\ \#1$ (green) as the other instruction to swap with. Figure 3.3(b) shows how the program generates a completely wrong result when these two instructions are swapped. In actual application, a firmware can follow many execution paths. It is difficult to quantify the probability of "incorrect execution" based on a static copy of an obfuscated firmware. However, Rule 5 in Set-I guarantees multiple instructions swapping is spread throughout the firmware. This will maximize the probability of wrong execution of an obfuscated firmware. After swapping the instruction, the obfuscated firmware is simulated in ARMkeil® compiler to verify the rules' efficacy. This is considered a sufficient condition to prove that the rules are in fact capable of obfuscating the proper execution flow of the firmware.

3.2 Reconstruction methodology

Since the obfuscated firmware is stored in the non-volatile memory, additional hardware support is needed on the processor to reconstruct the original firmware. In this section, we present a structure that consists of a small direct mapped cache, which translates the addresses of the swapped instructions. Note that one can use different implementations based on the processor and memory organization for address translation.

Figure 3.4 shows our proposed implementation which requires the bootloader to recover the swapped instructions to reconstruct the firmware back to its proper arrangement. Today, almost all the devices use a bootloader to perform memory partitioning, hardware checks, clearing interrupt flags etc. by obtaining the entire firmware from the non-volatile memory [66]. We propose to generate the swapped instructions during this power-up time. In this initial phase, the bootloader retrieves the system ID from the PUF and program key K from the NVM to reconstruct the obfuscation key K_O. Since K_O is the relative addresses of the swapped instructions, the bootloader maps the relative addresses to physical addresses of the flash memory. Whenever an address matches with an entry in the K_O, the instruction is loaded into the cache location corresponding to the address with which the instruction is swapped. Once this is complete, the reorder cache contains all the instructions that have been swapped in the original firmware. The red dotted portion in the Figure 3.4 highlighted this power-up sequence, and it is executed only once at the system boots up.

Since the swapped instructions and their relative addresses are in the cache, further execution would not require any authentication. During the execution of a program, the instruction memory is accessed by the processor sequentially, unless there is a hit in the reorder cache. Assume that the program counter points to an address AI_{o1}, which is the address for instruction I_{o1} for an obfuscated program. The memory should fetch the instruction I_{o1}; however, the address AI_{o1} leads to a cache hit as it is present in the cache. Consequently, I_{o2} is fetched to the processor from the cache and I_{o1} is discarded by the multiplexer. While

it thwarts information leakage regarding swapped instructions even if the address bus is dynamically monitored, this simple reconstruction method ensures that the firmware can be executed seamlessly by the system.

A direct mapped cache has been employed in the design because its lower hardware overhead compared to fully set associative cache [67]. The cache contains all the swapped-instruction pairs and the tags of their corresponding relative addresses. In this manuscript, we considered the device ID of 1024 bits and the address for instructions of 32 bits. Therefore, we can have 32 cache lines and requires 5 address bits to represent $index$. We also consider instructions are of 4 bytes wide. As a result, we reserve two address bits for byte offset. Therefore, the size of the Tag will be $32 - 5 - 2 = 25$ bits and takes $Address[31 : 7]$ for tag comparison.

Firmware updates are an essential part of secure IoT system development. Usually, recent embedded devices are capable of handling the update process through a built-in device firmware update (DFU) features [68, 69]. In general, the system integrator can achieve the firmware update functionality using a bootloader if no DFU is available. The bootloader is forced into a secure update state during power-up by a hardware interrupt. Before the firmware update, the original manufacturer or system integrator (SI) obtains the public device ID from the device. The SI can then use the public ID to access a database and retrieve the private ID (PUF responses during registration). Using the private ID, the SI generate a new key K for that device. An obfuscated firmware also created using Algorithm 1. Then, the SI can send the obfuscated firmware and the updated key K to the system. Note that the program key K may not have to be updated if the instructions in the obfuscated update still adhere to the SRC guidelines. After the updated obfuscated firmware and program key are stored into flash, the device can reboot or reload its cache with the new instructions. With the cache updated for the updated firmware, the program can begin executing the reconstructed update.

With the additional hardware support and extra code to the bootloader, the firmware can be reconstructed on power-up without extra overhead on computational performance during normal execution. It also allows for simple and secure firmware updates, making it suitable for devices with strict resource constraints.

In this chapter, we proposed a firmware obfuscation method to prevent cloning of an edge device. Instruction swapping is the method of obfuscating a firmware. Essential rules for preventing an attacker from reconstructing firmware using static analysis is also proposed. Then, we discussed the reconstruction method and concluded that a simple hardware modification can allow seamless execution of the obfuscated firmware.

Chapter 4

Security Analysis

In this chapter, we mathematically analyze the security of the proposed obfuscation method. We show that the reconstruction of an obfuscated firmware presented in Chapter 3 without proper key and hardware assistance is infeasible.

4.1 Security analysis

The total number of trials or arrangements of instructions (denoted as attacker's effort, AE) the attacker must perform to ensure the complete reconstruction of the original firmware, is calculated using this model. We calculate AE using few example cases of firmware to show that the obfuscation is practical and secure for real programs.

To find the attacker's effort, the firmware is modeled as a directed acyclic graph (DAG) [37, 52]. This is done by removing loops in the firmware and showing the possible ways the program can execute. Let the vertices I be the instructions in the firmware, and paths P are the different ways the program can execute. Let us assume that there are m paths in the graph that represents all different execution flows, total E number of instructions, and N_T number of instructions that are swappable according to the swapping rule check, SRC. Let L_T be the total number of swaps performed by the algorithm. Let h be the set of path lengths in the DAG. Figure 4.1 shows the DAG model.

The total AE depends on how many paths an attacker can identify as a failing path. These are paths that do not complete or produce an obviously incorrect output. The total AE is the sum of the effort required to reconstruct the failing paths that were identified, and the effort required to reconstruct the rest of the firmware.

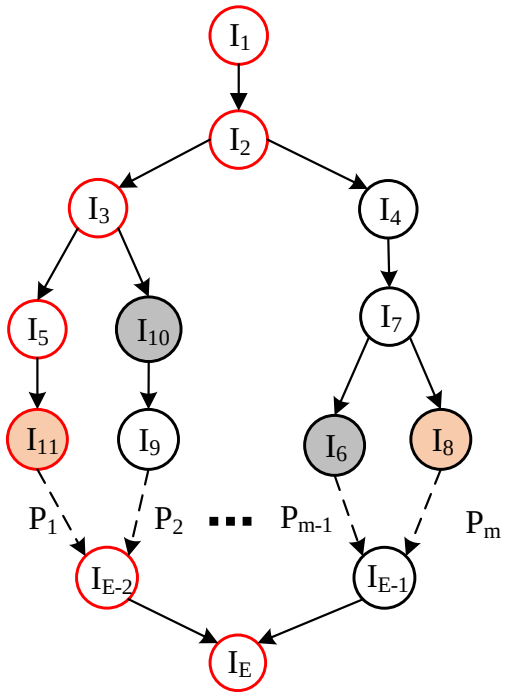

Figure 4.1: Directed Acyclic Graph of a Firmware. The path, P_1 highlighted in red, represents a failing execution. Instruction I_8 is swapped with I_{11}.

$$AE_T = AE_F + AE_U \tag{4.1}$$

When the attacker knows exactly which paths are failing, the effort to find the swapped instructions should be smaller than the effort when he/she does not observe a failing program execution. We will first examine the effort to make a program completely working when an attacker knows the failing execution paths. For example, let us assume that the red path in Figure 4.1 has been identified by an attacker as a failing path. Here, I_{11} and I_8 have been swapped, which causes a failure in P_1. Since the attacker knows that at least one instruction in that path is swapped, it reduces the number of instructions that must be checked. Equation 4.2 shows how many trials an attacker must run to check a single swap in M failing paths.

$$AE_F = h_1(N_T - 1) + h_2(N_T - 3) + \ldots + h_M(N_T - 2M + 1) \quad (4.2)$$

where, h_1, h_2, ..., h_M are the length of path P_1, P_2, ..., P_M respectively. While there may be more swapped instructions in the same failing path that would add complexity, the best-case for the adversary is that every modified path only has one swapped instruction.

Even if the attacker has found a failing path and reduced the number of instructions to check, there may be multiple instructions that cause the path to succeed. For example, if there are multiple occurrences of the same instruction in the firmware, an attacker may make a swap that causes the known path to succeed, but another unknown path to fail. Since each swap is unique, the only way an attacker can be sure that the firmware is correct is to check every possible swap, not just the swaps that fix the one known failing path.

We will now examine the case where an adversary does not know the failing paths. We believe that it is possible to swap instructions from paths that are difficult to execute by an adversary. In addition, there is usually a large number of execution sequences or paths ($\gg E$) for a program. It would be difficult for an attacker to find all the failing paths in the firmware, as he does not know which inputs cause the execution to go down each path. If the attacker cannot find all the failing paths, an attacker must try every arrangement of the remaining swappable instructions to reconstruct the original firmware. Let's say that an attacker has performed M swaps to fix the failing paths that have been discovered. This means those swaps and instructions can be removed from the analysis.

$$L = L_T - M \quad (4.3)$$

$$N = N_T - 2M \quad (4.4)$$

It is useful to represent the remaining swappable instructions (N) as an undirected graph as we do not require to preserve the directivity to calculate attacker's effort. Let G be an undirected graph of N instructions in the firmware that can be swapped based on the swapping rule check, SRC described in Section 3.1. Let every edge between two vertices in the undirected graph indicate that the two instructions can be swapped with each other. A swap occurs when an edge is chosen in the graph. After the first edge is chosen, the two swapped instructions (e.g., I_1 and I_2) cannot swap with any other instruction. The edges must be chosen so that no two edges are adjacent. This is shown Figure 4.2, where Figure 4.2. (a) shows all the potential swaps in a firmware with five swappable instructions. After I_1 and I_2 are swapped in Figure 4.2. (b), all edges adjacent to those instructions are no longer valid swaps and cannot be counted for further arrangements. But the attacker still has to try every other non-adjacent (disjoint) edge to find the second swap.

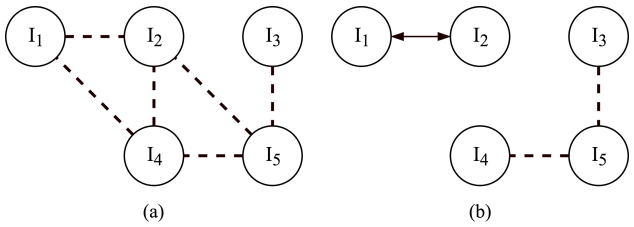

Figure 4.2: Graph model of two instructions being swapped in firmware. Possible swaps are shown with dotted lines (a). The edges adjacent to the swapped instructions are no longer valid when a swap is chosen (b).

The attacker's effort (AE_U) to find L swaps in the firmware can thus be described as the number of ways one can choose L disjoint edges. This is also known as a "matching" of the graph G, or more specifically the "k-edge matching" where $k = L$. This is a difficult problem and has not been solved in closed form for a general graph. Still, using the k-edge matching for special graphs, the upper bound and lower bounds for the attacker's effort can be calculated. Figure 4.3 shows the graphs in which these bounds would occur for $N = 6$.

In the worst-case scenario for an attacker, the graph G is a complete graph, and any instruction can be swapped with any other instruction. In this case, the number of "k-edge matches" is closely related to mathematical "telephone numbers" or "involution numbers"

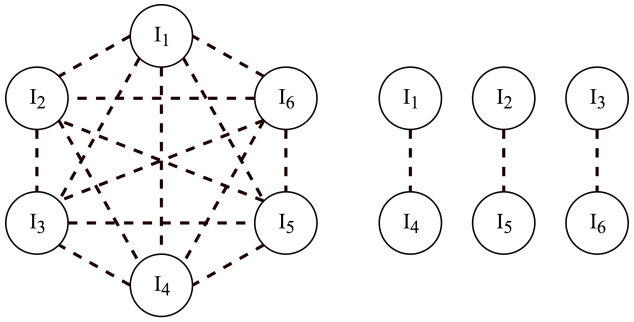

Figure 4.3: Worst-case (a) and best-case (b) graphs for the adversary when $N = 6$

Table 4.1: Attacker's Effort (AE) to reconstruct a complete program.

Program	**# Total Instructions (E)**	**# Swappable Instructions (N)**	**Attacker's Effort (AE)**			
			L = 4 : M = 12	***L = 8 : M = 8***	***L = 12 : M = 4***	***L = 16 : M = 0***
qsort_large.s	181	77	1.56×2^{7}	1.32×2^{24}	1.24×2^{47}	1.96×2^{76}
dijkstra.s	342	237	1.49×2^{13}	1.22×2^{40}	1.48×2^{77}	1.58×2^{107}
fft.s	368	191	1.62×2^{15}	1.10×2^{42}	1.01×2^{71}	1.27×2^{110}
basicmath.s	480	209	1.47×2^{19}	1.82×2^{53}	1.43×2^{87}	1.18×2^{124}
sha.s	577	471	1.09×2^{28}	1.88×2^{72}	1.68×2^{118}	1.51×2^{172}
rsa.s	1658	1297	1.48×2^{37}	1.54×2^{90}	1.37×2^{150}	1.43×2^{211}
aes.s	1757	1134	1.38×2^{35}	1.81×2^{84}	1.73×2^{140}	1.11×2^{203}

[70]. The number of arrangements can be described as the number of ways you can choose 2 instructions out of N instructions, multiplied by the number of ways you can choose 2 instructions out of the remaining $N-2$ instructions, and so on until you have L swaps. Since the order in which the instructions are swapped does not matter, this term is then divided by $L!$. The number of arrangements in the worst-case is then shown to be:

$$\begin{aligned} AE_{U-W} &= \left(\frac{1}{L!}\right)\binom{N}{2}\binom{N-2}{2}\cdots\binom{N-2L+2}{2} \\ &= \frac{N!}{2^L(L)!(N-2L)!} \\ &= \frac{N(N-1)\ldots(N-2L+1)}{2^L(L)!} \end{aligned} \tag{4.5}$$

Equation 4.5 gives the worst-case effort for an adversary to reconstruct an obfuscated firmware and has a complexity of $O(N^{2L})$ as $N >> L$.

In the best-case situation for an adversary, every swappable instruction in the graph can only swap with one other instruction. In this case, the attacker would be choosing L edges from $N/2$ possible swaps, so the effort to find the remaining swaps would be:

$$\begin{aligned} AE_{U-B} &= {}^{N/2}C_L = \binom{N/2}{L} \\ &= \frac{N(N-2)\ldots(N-2L+2)}{2^L(L)!} \end{aligned} \tag{4.6}$$

Equation 4.6 gives the best-case effort for an adversary to reconstruct an obfuscated firmware and of $O(N^L)$ as $N >> L$.

In both of these cases, $AE_U \gg AE_F$ for even small values of L, we can ignore the AE_F term and estimate AE_T to be approximately equal to AE_U.

$$AE_T \approx AE_U \tag{4.7}$$

For further analysis, we consider the specific implementation of the obfuscation where $L_T = 16$. In this case, 16*2=32 instructions need to be swapped. If we use a 32-entry direct-mapped cache, described in Section 3.2, the Index Distinction rule will apply (see Section 3.1.2). This means that for every swap, roughly $2/32 = 1/16$ of the remaining swappable instructions will no longer be swappable. This reduces the number of possible arrangements in both the best-case and the worst-case. The adjusted attacker's effort will then be:

$$AE_{T-W} \approx \frac{(32-2M)!}{32^{2L}} AE_{U-W} \tag{4.8}$$

$$AE_{T-B} \approx \frac{\prod_{i=1}^{L}(2i-1)}{32^L} AE_{U-B} \tag{4.9}$$

where M represents the number of swaps an attacker has found from failing paths, and L represents the remaining unknown swaps.

While the model provides best-case and worst-case scenarios on attacker's effort, it is necessary to calculate the definite number of trials for an adversary, which needs to be performed to reconstruct different benchmark programs. In this analysis, we write a Python script to analyze the firmware and count a potential number of swaps in a given program using the SRC described in Section 3.1. The script parses through the ARM assembly file that is generated by the GNU Arm Embedded Compiler [71] and locates the instructions that are swappable. Then, it counts up the number of possible swaps between all the swappable instructions, before using Algorithm 1 to generate the obfuscated firmware. It finally estimates AE_U by taking the product of the number of possible swaps after each swap in the algorithm.

Table 4.1 shows the actual attacker's effort, which is calculated using the Python script describe above. The example programs listed were benchmark tests from MiBench2 [72]. Here, columns 4, 5, 6, and 7 represent AE for unknown paths L= 16, 12, 8, and 4. Note that, L= 4, 8, 12, and 16 correspond to M= 12, 8, 4, and 0 respectively (See Eqn. 4.3). The length of the ID depends on the number of instructions to be swapped, therefore, must be defined depending on the expected security level during design phase of the system. Here, all calculations assume the ID to be 1024 bits, instruction width 32 bits, $L_T = 16$, and the addresses must fit in a 32-entry direct-mapped cache. For example, lets consider a *sha.s* ARM assembly code that composed of 577 instructions, and algorithm 1 finds total 471 swappable instructions. If the attacker can find four of the failing paths ($L = 12$), then it would require 1.68×2^{118} simulations to ensure reconstruction of the original firmware.

Figure 4.4 shows the comparison between the theoretical and actual attacker's effort. The theoretical worst-case and best-case values for each benchmark are calculated using Equations 4.8, and 4.9, respectively. The actual value is the effort calculated by the Python script. The vertical axes of all the graphs are on the logarithmic scale. Figure 4.4(a) shows the AE_T when an attacker cannot find any failing paths ($L = 16$) for the firmware. We

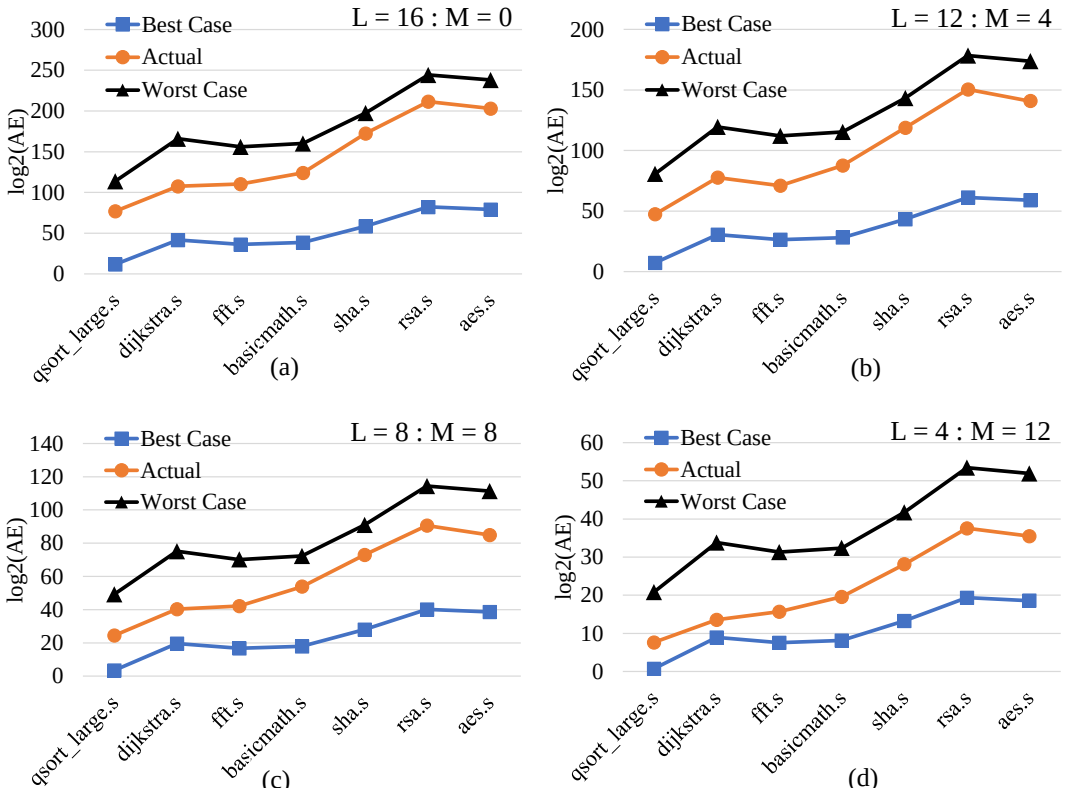

Figure 4.4: Comparison of actual and estimated attacker's effort (AE_T). Worst-case and best-case AEs are estimated from Equations 4.8 and 4.9, respectively. Graphs (a)-(d) show how the attacker's effort changes if an attacker discovers the correct swaps for M failing paths, with L swaps still unknown

expect the actual value should be lower and upper bounded by the best-case and worst-case estimate of the attacker's effort, respectively. Figures 4.4(b)-(d) show the AE_T when an attacker can find 4, 8, and 12 swaps from observing different failing paths. Note that an attacker needs to perform a smaller number of trials once he/she observes an increased number of failing swaps. However, it will be difficult for an adversary to find all failing paths. Based on the discussion above, we conclude that the number of trials to make a program completely work is 1.54×2^{90} for a small program like $rsa.s$, showing the obfuscation to be secure considering current computing resources.

4.2 Tamper resistivity

This proposed firmware obfuscation and reconstruction method can inherently thwart any tampering with the firmware or hardware. This scheme can help us to detect cloned systems without performing expensive and less reliable test methods (visual inspection, X-Ray, etc.). The cloning incidents reported by Bloomberg [36] can easily be detected. These cloned motherboards have a small chip that creates a stealthy doorway for malicious purposes by injecting malicious codes. When an infected code runs in the motherboard, the original address space for the program is modified. This modified program will produce incorrect results as the swapped addresses in the obfuscated program will not be reconstructed properly. By observing a flag, any modifications on the obfuscated program will easily be detected. For example, we illustrated the firmware obfuscation concept in Figure 3.3 and it will be further used to show that how malicious modifications can be detected. If an adversary injects new instructions, the relative addresses of swapped instructions will be changed in the obfuscated program. For instance in the obfuscated program (Figure 3.3.b), insertion of one instruction before instruction 7 ($lsl\ r3,\ r3,\ \#1$) will change the address of instruction 7. When this happens, the reorder cache will swap a wrong instruction, and the obfuscated program will not run. If an adversary inserts a malicious instruction at the first address, instruction 6 ($add\ r3,\ r3,\ r2$) will be swapped with instruction 2 ($lsl\ r3,\ r3,\ \#1$) as the address of instruction 6 in the tampered program is 7, which is present in the reorder cache. As a result, the obfuscated firmware will not be compensated properly, and will produce incorrect results.

In this chapter, the firmware obfuscation scheme is analyzed for its security properties. In addition, we discussed how this scheme can be applied to mitigate the tampering of hardware or firmware.

Chapter 5

Implementation

In this chapter, we demonstrate an implementation scheme for the obfuscation and reconstruction method presented in Chapter 3. A subset of the MIPS-32 processor core, MIPS-16, has been used to demonstrate the hardware operation of reconstruction scheme. Note that, the same technique can be applied to any other processor architecture even though implementation details may slightly vary. First, we show execution of a small firmware in MIPS processor and then customize the core with a few blocks of logic circuits, such as reorder cache, debug core, etc. Second, we illustrate the working principle of the proposed scheme with examples and discuss a few design considerations. Finally, the implementation overhead is explained at the end of the chapter.

5.1 Hardware development

Verilog HDL has been used to develop hardware. The RTL is simulated, synthesized, and implemented in Vivado development environment targeting a Xilinx Artix-7 series FPGA. The design is divided into two major components: a) custom processor b) debug core which is illustrated in Figure 5.1 and described in the subsequent sections.

5.1.1 Processor

This is a single cycle 16-bit MIPS processor core. Since this is a RISC instruction set architecture, the instruction length is fixed. To implement the firmware reconstruction method, understanding the internal structure of a processor is not essential, and it would derail us from the primary purpose of this chapter. Therefore, instruction set and list of the registers are added in the appendix. Note that, the processor core has a debug port

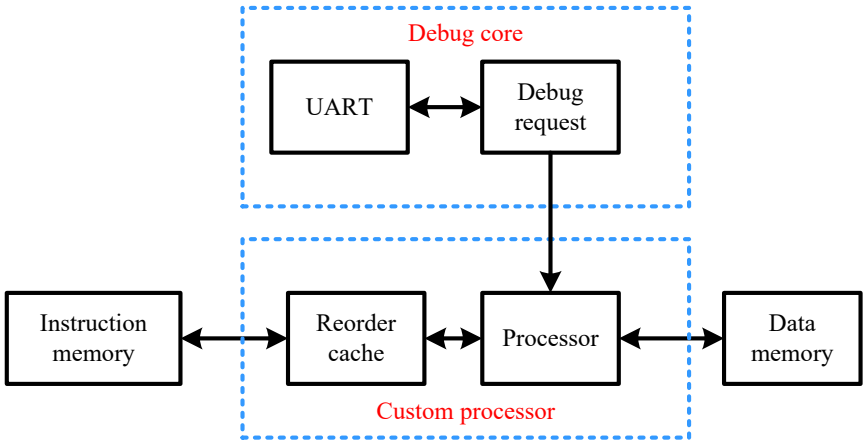

Figure 5.1: Top level implementation block diagram.

(*RegAcess*) that can be used to check the internal register value during the simulation phase but it has been disabled during implementation.

5.1.2 Reorder cache

As discussed in Section 3.2, the reorder cache is placed between instruction memory and CPU. The cache and its hit/miss decision circuitry are integrated into the same HDL module. Note that, the cached content does not change over execution of the firmware so the reorder cache is modeled as a combinational circuit.

5.1.3 Debug core

A debug core is designed and interfaced with the processor to demonstrate the results of a firmware execution through a serial port. The FPGA is connected to a host PC through a USB cable. The debug core is designed to take keyboard interrupt through that cable. Once interrupted, the debug core reads the entire data memory and dump that to a UART module. The debug core consists of a UART, controlling state machines, circuitry for accessing data from the memory. The UART baud rate is setup up to be 9600, and with 1 stop bit, no parity bit, and 8 data bits. The data access circuit reads the data memory sequentially. Since the

data memory of the processor is a single port distributed RAM mapped into FPGA LUTs, it cannot be used by two modules, CPU and debug core, simultaneously. Therefore, instead of queuing reading request we freeze the execution of the CPU until the debug core completes the debug request from the host PC. The debug core operation is as follows:

- Receive a keyboard interrupt from a host PC through URAT port.
- Halt CPU execution and dump the content of data memory to the UART port and eventually to the host PC.

5.2 Firmware development and obfuscation

We write an example firmware to illustrate the working principle of the implementation scheme. The following assembly code generates the first eight unique integers in the Fibonacci sequence starting from 0 to 21. Equation 5.1 is the general expression for a Fibonacci sequence generation. Here, F_n stands for n^{th} Fibonacci number in the sequence. The processor computes a Fibonacci number in every iteration and stores the number in the data memory sequentially.

$$F_n = F_{n-1} + F_{n-2} \tag{5.1}$$

```
;Original firmware
000: ADDI $t0,$0,0x1
001: ADDI $t1,$0,0x1
002: ADDI $a1,$0,0x1
003: STORE $t0,(0x0)$a1
004: STORE $t1,(0x1)$a1
005: ADDI $a1,$0,0x1
006: ADD $t0,$t1,$t0
007: STORE $t0,(0x1)$a1
008: JE $a1,0x5, 0x00B
009: LOAD $t1,(0)$a1
00A: JUMP  0x005;
00B: RET
```

```
;Obfuscated-firmware
000: ADDI $t0,$0,0x1
001: ADDI $a1,$0,0x1; swapped with 005
002: ADDI $a1,$0,0x1
003: STORE $t0,(0x1)$a1; swapped with 007
004: STORE $t1,(0x1)$a1
005: ADDI $t1,$0,0x1; swapped with 001
006: ADD $t0,$t1,$t0;
007: STORE $t0,(0x0)$a1; swapped with 003
008: JE $a1,0x5, 0x00B
009: LOAD $t1,(0)$a1
00A: JUMP  0x005;
00B: RET
```

The machine code for both original and obfuscated version of the firmware is listed in Table 5.1. The green and red machine code indicate swapped instructions pairs. Once loaded

Table 5.1: Machine code for the firmware

Original	**Obfuscated**
1010000001000001	1010000001000001
1010000010000001	1010110110000001
1010000011000110	1010000011000110
1100110001000000	1100110001000001
1100110010000001	1100110010000001
1010110110000001	1010000010000001
0011001010001000	0011001010001000
1100110001000001	1100110001000000
1101110011000010	1101110011000010
1011110010000000	1011110010000000
0001000000000101	0001000000000101
0000000000000000	0000000000000000

with original firmware, the CPU calculates each Fibonacci number which is temporarily stored in register $t1. The sequence of the integers is calculated by the instruction at address 0x006 of the original firmware.

The original firmware is obfuscated by swapping the instruction at address 0x001 with the instruction at address 0x005, and the instruction at address 0x003 with the instruction at address 0x007. Note that, the obfuscation follows the rules that are presented in Section 3.1. When the instruction memory is loaded with obfuscated firmware and directly executed in an unmodified MIPS processor, it calculates integers sequentially instead of Fibonacci sequence- clearly produces wrong results. This proves an important concept that the direct execution of an obfuscated firmware would produce a wrong result.

Now, we discuss a modification in the processor (and design a custom processor) which leads to seamless execution of the obfuscated firmware by a legitimate hardware. As discussed in the Chapter 3, the cache contains swapped instructions and their relative addresses. For

Table 5.2: Reorder cache contents

Address	Instruction
0000001	1010000010000001
0000011	1100110001000000
0000101	1010110110000001
0000111	1100110001000001

example, an instruction at address 0x001 is swapped with an instruction at address 0x005. If the processor request the instruction from the memory address 0x001, the cache should fetch instruction from address 0x005 that is $ADDI\ \$t1, \$0, 0x1$ instead of $ADDI\ \$a1, \$0, 0x1$. The swapped cache addresses and instructions are listed in Table 5.2. If the firmware is copied and loaded into another device even with the customized architecture like the one proposed in Section 3.2, invalid ID would still prevent the correct execution of the firmware. For example, assume that for an illegitimate device, the cached content in row 4 and column 1 in the table 5.2 is changed from 0x007 to 0x006. If the firmware starts to be executed in this state, the memory will be loaded with all "1"s and execution will end up being an infinite loop. This phenomenon is reflected in Figure 5.2(a) [Line-2].

5.3 FPGA implementation

The RTL is synthesized and implemented for an ARTIX-7 (XC7A15T-1CPG236C) FPGA. We applied partial configuration feature facilitated by Vivado IDE to demonstrate the correct and wrong execution to avoid multiple syntheses and loading of the bitstream in the FPGA. Partial configuration is a powerful design technique that we can leverage to reconfigure a part of the design without interrupting the rest of the design [73].

Figure 5.2(a) shows UART access terminal. Line-1 is the correct execution of the firmware as described in Section 5.2, and Line-2 is the wrong execution when the cache is reconstructed incorrectly as described at the end of previous section. Figure 5.2(b) illustrates the floor plan for partially configured reorder cache. Figure 5.2(c) shows a host computer which is working as a debug interface. Figure 5.2(d) shows a CMOD a7-15T board

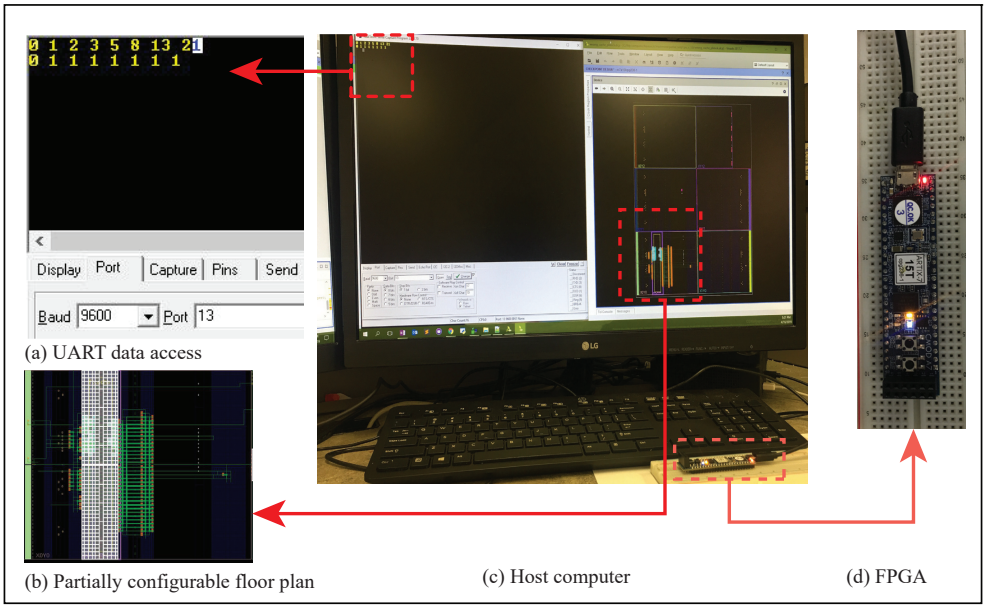

Figure 5.2: Implementation setup for the design.

which is running the design. Note that, the CPU core is clocked with a 12MHz external oscillator present in the evaluation board CMOD a7-15T.

The design process is discussed as follows:

1. The full design has been synthesized, placed, and routed into a Vivado design checkpoint that contains all the modules and correct reconstruction information (e.g., relative addresses and swapped instructions) in the reorder cache.

2. The reorder cache is converted into a reconfigurable module. The highlighted white part of the Figure 5.2(b) illustrates the reconfigurable part of the design.

3. A black box is created in the reconfigurable partition locking the static part of the design in the routing level. Note that, the static part of the design contains a debug core, processor core, and both instruction and data memories. Figure 5.3 illustrates the top level implementation scheme. The reconfigurable modules will be loaded in the reconfigurable partition sequentially to demonstrate the effect of correct and incorrect

firmware deobfuscation. *Black box* contains no logic elements other than input/output in the partial configuration boundary. Reconfigurable module *correct information* contains the correct deobfuscation information whereas the reconfigurable module *incorrect information* contains wrong relative address.

4. The design is updated, synthesized, and implemented with all three reconfigurable modules sequentially.

5. Bitstream generation step produces bitstream of two reorder cache - correct and incorrect - and top module with a black box in the reconfigurable partition.

Demonstration steps are discussed as follows:

1. The FPGA is loaded with top module (that includes both static and configurable partition) through a JTAG interface. The reconfigurable partition has a black box in it. Then, reorder cache with correct reconstruction information is slid into the reconfigurable partition. A UART data reading software is running in the host computer to display the debug information. We used *RealTerm* to capture debug data in the host PC.

2. We send an interrupt from the keyboard of the host PC to the design running in the FPGA. This interrupt freezes the execution of firmware, and the debug core takes control and starts fetching the memory content from the data memory to the host PC. The processor computes the correct Fibonacci sequence because the cache has legitimate reconstruction information. This produces the Line-1 in the Figure 5.2(a) which is correct Fibonacci sequence.

3. To demonstrate the effect of wrong data in the reorder cache, we send the reorder cache with *incorrect information* and repeat step 1. The result is shown in the second line of Figure 5.2(a), and it demonstrates the incorrect execution of an obfuscated firmware that is incorrectly reconstructed.

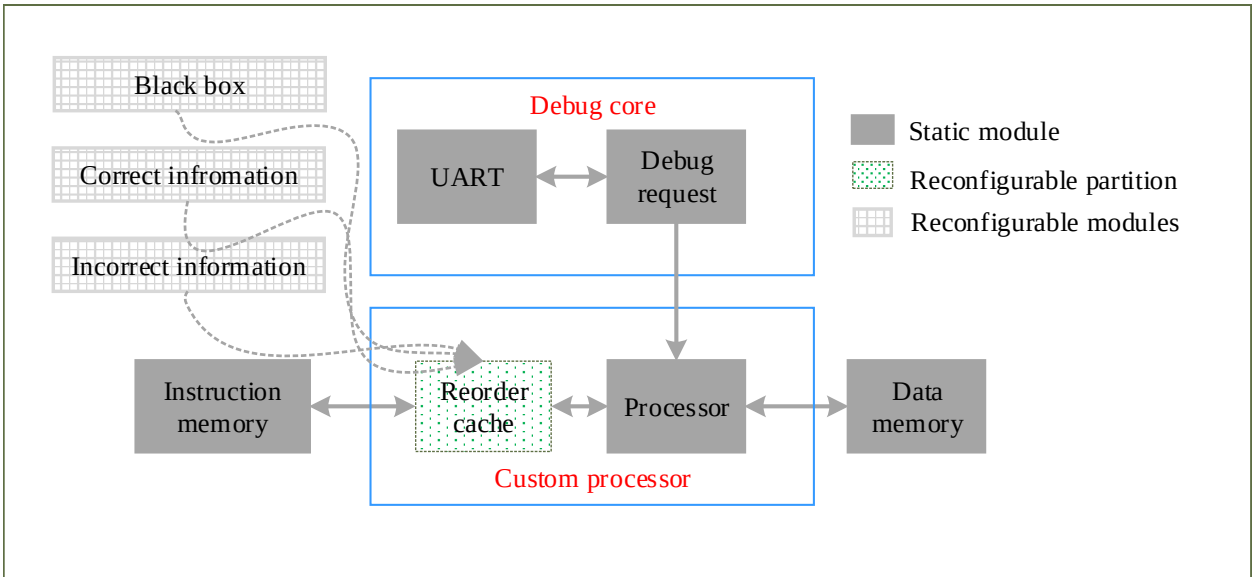

Figure 5.3: Demonstration scheme for the implementation.

5.4 System design consideration

A content addressable memory (CAM) might be used to reduce circuit complexity since the contents of reorder cache should be the same for entire execution. The problem with CAM is that it would stall the execution of the CPU before cache hit/miss gets resolved. Fortunately, our hardware modification does not require individual item search in the cache (hence its a direct mapped cache). The cache in this example demonstration is designed as an asynchronous multiplexer to reduce the access latency.

Based on the processor architecture, it is possible that some instruction cannot be swapped even if the rules mentioned in Chapter 3 are satisfied. For example, successive *ADD $t0, $t1, $t1* and *SUB $t2, $t0, $t0* instructions will need one *NOP* instruction in between to avoid data hazard in a five-stage pipeline architecture. If swapping produces these two instructions in a successive location, it would be a hazard and easily detectable. In general, even if the rules allow, swapping instructions that conflict with the dependencies will tip off an attacker. Therefore, pipeline architecture would require dependencies to be considered in association with the rules described in Chapter 3.

5.5 Overhead analysis

The proposed method does not add any overhead in the firmware size since the overall code size is the same for the original and obfuscated version. However, the boot-loader needs to be modified so that it can handle power-up cache loading and firmware update mechanism. Hardware overhead comes from the reorder cache and obfuscation key memory requirements. Specifics of cache circuit design is out of the scope of this thesis. Nevertheless, we can provide a close approximation of gate counts essential to the design. Maintaining consistency with the discussion in Chapter 3 let us assume 1024-bit ID, 32 cache lines with a 16-bit width, and 25 tag bits for each line. The cache needs 32×16 bit and 32×25 bit cache memory elements for instructions and tags respectively. One 5-to-32 address decoder, 2-to-1 multiplexers, and 25-bit comparator are required for cache hit/miss decision and instruction fetch from cache or memory. It would take approximately 200 gates to implement these components along with 1024 XOR gates for key (K_0) generation. The key storage would take $1K$ bit nonvolatile memory. Note that, we do not require any hardware overhead to generate the ID as a system's SRAM can be used as a PUF. For the implementation presented in this chapter, the overhead is less 0.1% with insignificant power consummation. The logic overhead is insignificant considering the size of modern embedded processors that are common in IoT devices.

In this chapter, we discussed an example implementation of the scheme that could effectively counter cloning. Also, we provided overhead analysis along with a few design considerations.

Chapter 6

Conclusion and Future Research

Preventing electronic systems from being cloned is of paramount importance. Cloning can occur in individually firmware and hardware or in a system as a whole. We presented a novel low-cost method of firmware obfuscation that protects a system from cloning. The proposed technique obfuscates the firmware by swapping a few instructions rather encrypting the entire firmware. We showed how swaps can be selected according to the Swapping Rule Check (SRC) to ensure that the obfuscated instructions are not obvious to an attacker. The relative addresses of these swapped instructions are combined with an unclonable ID to generate a unique obfuscation key that gets stored on each device. Using this obfuscation key and the device ID, a device can reconstruct the relative addresses of the swapped instructions and store them in a small cache. As the program executes, we explained how this cache is used by the processor to execute the program in the correct order. Our proposed solution does not increase the number of instructions. Only a small reorder cache (e.g., direct mapped cache) and a PUF is required to reconstruct the original firmware. Although deobfuscation requires hardware modification, we still can reuse already existing hardware. For example, SRAM is present in almost every embedded device which can be used to generate an unclonable device signature. Swapping a small set of instructions provides exponential complexity and thus infeasible for an adversary to reconstruct the original firmware considering current computing resources. Note that an attacker needs to perform a smaller number of trials, once he/she observes an increased number of failing paths for a program.

Future work will certainly include increasing security and reducing hardware overhead. The selection of instruction for a possible swap needs to be addressed such that an adversary cannot find a failing program execution.

Even if the cache is incorrectly reconstructed it is still possible to bypass the security feature if an adversary has access to any internal CPU registers. As an example, the final result is coming from the memory but the CPU is using resisters to produce that results (see Chapter 5). Any register access will weaken or downright break the security of obfuscation. To prevent this, the debug interface must be locked in this case which could be an issue for future troubleshooting or diagnosing a system malfunction. Therefore, a method needs to be developed to overcome this issue.

The firmware update is an important feature for the post-deployment phase of a device. Since the obfuscation process changes the relative address of swapped instruction, a secure firmware/hardware needs to handle this firmware update process. Therefore, a scheme must be developed that can handle the secure update process without leaking key information.

Tracing the activity of an execution unit through a power bus can be a threat to the obfuscation method. The reorder cache itself has no vulnerability in terms of side channel analysis. However, since the power consumption will be different for fetched and executed instruction in case of a swapped instruction, it is not impossible to identify the swapped instruction by comparing the power profiles of a legitimate and a cloned system. Therefore, an effective method needs to be developed to mitigate side channel information leakage.

The reconstruction scheme is designed focusing on architectures that have a fixed instruction set. For variable length instruction, the method needs modification, and the security features need to be re-evaluated.

Bibliography

[1] M. M. Tehranipoor, U. Guin, and D. Forte, "Counterfeit integrated circuits," in *Counterfeit Integrated Circuits*. Springer, 2015, pp. 15–36.

[2] K. Chatterjee and D. Das, "Semiconductor manufacturers' efforts to improve trust in the electronic part supply chain," *IEEE Transactions on Components and Packaging Technologies*, vol. 30, no. 3, pp. 547–549, 2007.

[3] U. Guin, D. Forte, and M. Tehranipoor, "Design of accurate low-cost on-chip structures for protecting integrated circuits against recycling," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 4, pp. 1233–1246, 2015.

[4] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, "Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.

[5] U. Guin, M. Tehranipoor, D. DiMase, M. Megrdichian *et al.*, "Counterfeit ic detection and challenges ahead," *ACM SIGDA*, vol. 43, no. 3, pp. 1–5, 2013.

[6] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security." in *USENIX security symposium*, 2007, pp. 291–306.

[7] M. Rostami, F. Koushanfar, J. Rajendran, and R. Karri, "Hardware security: Threat models and metrics," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 819–823.

[8] M. Tehranipoor and C. Wang, *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[9] F. Koushanfar, "Provably secure active ic metering techniques for piracy avoidance and digital rights management," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 51–63, 2012.

[10] G. K. Contreras, M. T. Rahman, and M. Tehranipoor, "Secure split-test for preventing ic piracy by untrusted foundry and assembly," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 196–203.

[11] J. A. Roy, F. Koushanfar, and I. L. Markov, "Ending piracy of integrated circuits," *Computer*, vol. 43, no. 10, pp. 30–38, 2010.

[12] U. Guin, Q. Shi, D. Forte, and M. M. Tehranipoor, "Fortis: a comprehensive solution for establishing forward trust for protecting ips and ics," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 4, p. 63, 2016.

[13] P. Chowdhury, U. Guin, A. D. Singh, and V. D. Agrawal, "Two-pattern iddq test for recycled ic detection," in *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*. IEEE, 2019, pp. 82–87.

[14] U. Guin, T. Chakraborty, and M. Tehranipoor, "Functional f max test-time reduction using novel dfts for circuit initialization," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 1–6.

[15] H. Dogan, D. Forte, and M. M. Tehranipoor, "Aging analysis for recycled fpga detection," in *2014 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT)*. IEEE, 2014, pp. 171–176.

[16] Z. Guo, M. T. Rahman, M. M. Tehranipoor, and D. Forte, "A zero-cost approach to detect recycled soc chips using embedded sram," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016, pp. 191–196.

[17] Y. Zheng, A. Basak, and S. Bhunia, "Caci: Dynamic current analysis towards robust recycled chip identification," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[18] X. Zhang, K. Xiao, and M. Tehranipoor, "Path-delay fingerprinting for identification of recovered ics," in *2012 IEEE International symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT)*. IEEE, 2012, pp. 13–18.

[19] S. I. A. (SIA), "Public comments - dna authentication marking on items in fsc5962," 2012. [Online]. Available: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies

[20] M. Alam, S. Chowdhury, M. M. Tehranipoor, and U. Guin, "Robust, low-cost, and accurate detection of recycled ics using digital signatures," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2018, pp. 209–214.

[21] K. He, X. Huang, and S. X.-D. Tan, "Em-based on-chip aging sensor for detection and prevention of counterfeit and recycled ics," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 146–151.

[22] R. van der Meulen, "Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016," 2017, https://www.gartner.com/newsroom/id/3598917.

[23] Misc., "T-mote sky." [Online]. Available: http://wirelesssensornetworks.weebly.com/1/post/2013/08/tmote-sky.html

[24] ——, "Z1 mote." [Online]. Available: http://zolertia.sourceforge.net/

[25] ——, "T-mote sky." [Online]. Available: http://wirelesssensornetworks.weebly.com/1/post/2013/08/tmote-sky.html

[26] O. Mote, "Open hardware for the internet of things." [Online]. Available: http://openmote.com/product/openmote-b-platinum-kit/"

[27] W. C. (2016), "Dyn cyberattack." [Online]. Available: https://en.wikipedia.org/w/index.php?title=2016_Dyn_cyberattack&oldid=763071700

[28] W. Trappe, R. Howard, and R. S. Moore, "Low-energy security: Limits and opportunities in the internet of things," *IEEE Security & Privacy*, vol. 13, no. 1, pp. 14–21, 2015.

[29] K. Rawlinson, "HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack," 2014. [Online]. Available: http://www8.hp.com/us/en/hp-news/press-release.html?id=1744676#.WUrrwWgrKM8

[30] M. B. Barcena and C. Wueest, "Insecurity in the internet of things," *Security Response, Symantec*, 2015.

[31] M. Alam, M. M. Tehranipoor, and U. Guin, "Tsensors vision, infrastructure and security challenges in trillion sensor era," *Journal of Hardware and Systems Security*, vol. 1, no. 4, pp. 311–327, 2017.

[32] C. Suslowicz, A. S. Krishnan, and P. Schaumont, "Optimizing cryptography in energy harvesting applications," in *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security*. ACM, 2017, pp. 17–26.

[33] M. Hossain, R. Hasan, and A. Skjellum, "Securing the internet of things: A meta-study of challenges, approaches, and open problems," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 220–225.

[34] M. M. Tehranipoor, U. Guin, and S. Bhunia, "Invasion of the hardware snatchers," *IEEE Spectrum*, vol. 54, no. 5, pp. 36–41, 2017.

[35] M. M. Tehranipoor, U. Guin, and D. Forte, *Counterfeit Integrated Circuits: Detection and Avoidance*. Springer, 2015.

[36] J. Robertson and M. Riley, "The big hack: How china used a tiny chip to infiltrate u.s. companies." [Online]. Available: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies

[37] U. Guin, S. Bhunia, D. Forte, and M. M. Tehranipoor, "Sma: A system-level mutual authentication for protecting electronic hardware and firmware," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 3, pp. 265–278, 2017.

[38] S. E. Quadir, J. Chen, D. Forte, N. Asadizanjani, S. Shahbazmohamadi, L. Wang, J. Chandy, and M. Tehranipoor, "A survey on chip to system reverse engineering," *ACM journal on emerging technologies in computing systems (JETC)*, vol. 13, no. 1, p. 6, 2016.

[39] J. Obermaier and S. Tatschner, "Shedding too much light on a microcontroller's firmware protection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[40] *STM32F4 series of high-performance MCUs with DSP and FPU instructions*. [Online]. Available: http://www.st.com/en/microcontrollers/stm32f4-series.html?querycriteria=productId=SS1577

[41] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up sram state as an identifying fingerprint and source of true random numbers," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1198–1210, 2009.

[42] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, 2014.

[43] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2002, pp. 148–160.

[44] G. Suh and S. Devadas, "Physical Unclonable Functions for device authentication and secret key generation," in *Proc. of ACM/IEEE on Design Automation Conference*, 2007, pp. 9–14.

[45] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Fpga intrinsic pufs and their use for ip protection," in *International workshop on Cryptographic Hardware and Embedded Systems*, 2007, pp. 63–80.

[46] U. Guin, A. Singh, M. Alam, J. Canedo, and A. Skjellum, "A secure low-cost edge device authentication scheme for the internet of things," in *International Conference on VLSI Design*, 2018.

[47] Y. Li, J. M. McCune, and A. Perrig, "Viper: verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 3–16.

[48] M. LeMay and C. Gunter, "Cumulative attestation kernels for embedded systems," *Smart Grid, IEEE Transactions on*, vol. 3, no. 2, pp. 744–760, June 2012.

[49] D. Schellekens, P. Tuyls, and B. Preneel, "Embedded trusted computing with authenticated non-volatile memory," in *Trusted Computing-Challenges and Applications*. Springer, 2008, pp. 60–74.

[50] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse trap: exploiting firmware updates in usb peripherals," in *Proceedings of the 8th USENIX conference on Offensive Technologies*. USENIX Association, 2014, pp. 12–12.

[51] D. Morais, J. Lange, D. R. Simon, L. T. Chen, and J. D. Benaloh, "Use of hashing in a secure boot loader," Jun. 14 2005, US Patent 6,907,522.

[52] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Embedded software security through key-based control flow obfuscation," in *Security Aspects in Information Technology*. Springer, 2011, pp. 30–44.

[53] X. Zhuang, T. Zhang, H. S. Lee, and S. Pande, "Hardware assisted control flow obfuscation for embedded processors," in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2004, pp. 292–302.

[54] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.

[55] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, 2013, pp. 299–310.

[56] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015, pp. 850–861.

[57] A. Schaller, T. Arul, V. van der Leest, and S. Katzenbeisser, *Lightweight Anti-counterfeiting Solution for Low-End Commodity Hardware Using Inherent PUFs*, 2014.

[58] M. A. Gora, A. Maiti, and P. Schaumont, "A flexible design flow for software ip binding in commodity fpga," in *2009 IEEE International Symposium on Industrial Embedded Systems*, July 2009, pp. 211–218.

[59] R. P. Lee, K. Markantonakis, and R. N. Akram, "Binding hardware and software to prevent firmware modification and device counterfeiting," in *Proceedings of the 2nd ACM international workshop on cyber-physical system security*. ACM, 2016, pp. 70–81.

[60] J. X. Zheng, D. Li, and M. Potkonjak, "A secure and unclonable embedded system using instruction-level puf authentication," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 2014, pp. 1–4.

[61] J. X. Zheng, T. Xu, and M. Potkonjak, "Securing embedded systems and their ips with digital reconfigurable pufs," in *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2016 26th International Workshop on*. IEEE, 2016, pp. 169–176.

[62] W. Wang, A. Singh, U. Guin, and A. Chatterjee, "Exploiting power supply ramp rate for calibrating cell strength in SRAM PUFs," in *IEEE Latin-American Test Symposium*, 2018.

[63] C. Lomont, "Introduction to x64 assembly," 2012. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-x64-assembly

[64] *AVR Instruction Set Manual*, Atmel, 11 2016.

[65] *PICmicro Mid-Range MCU Family Reference Manual*, Microchip, 12 1997, section 29.

[66] A. K. T. Support. (2018) Arm: How to write a bootloader. [Online]. Available: http://www.keil.com/support/docs/3913.htm

[67] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[68] *STM32 microcontroller system memory boot mode transceiver*, STMICROELECTRONICS, 2 2018, rev. 32.

[69] J. Beningo, "Update firmware in the field using a microcontrollers dfu mode," 2018. [Online]. Available: https://www.digikey.com/en/articles/techzone/2018/jan/update-firmware-field-using-microcontroller-dfu-mode

[70] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998, pp. 65–67.

[71] GNU Arm Embedded Toolchain, https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads.

[72] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, 2001.

[73] *Vivado Design Suite User Guide: partial Reconfiguration*, Xilinx Inc, 4 2018.

Appendices

- The CPU assembly code and machine code follows the usual definition of MIPS architecture*.
- CPU core has been tested at 100MHz clock speed.

MIPS-16 Instruction set and register listings.

Table 1: Instruction set

Mnemonics	subroutine	**Operation**	Format
RET	0000	Halts execution	J-type
JUMP	0001	Unconditional jump	J-type
CALL	0010	Calls a subroutine	J-type
ADD	0011	Adds two registers	R-type
SUB	0100	Twos complement subtractions between two registers	R-type
AND	0101	Bitwise *and* between two registers	R-Type
OR	0110	Bitwise *or* between two registers	R-type
SLT	0111	Test magnitude between two registers	R-Type
SLL	1000	Shifts bits left	I-type
SRL	1001	Shifts bits right	I-type
ADDI	1010	Adds a register to an immediate constant	I-type
LOAD	1011	Moves data from the memory	I-type
STORE	1100	Stores data in memory	I-type
JE	1101	Jump if equal	I-type
JNE	1110	Jump if not equal	I-type
JR	1111	Jump to an address at a register	I-type

*https://www.mips.com/products/classic/

Table 2: Register definition

Machine code	**Mnemonics**	**Default operation**
000	$0	Hard-wired zero register.
001	$t0	Temporary registers.
010	$t1	Temporary registers.
011	$s0	Saved registers
100	$v0	Procedure returned value
101	$a0	Loads Procedure Arguments
110	$a1	Loads Procedure Arguments
111	$ra	Return address holder

Table 3: R-Type instruction

Opcode	rs	rt	rd	Function/control
4 bits	3 bits	3 bits	3 bits	3 bits

Table 4: I-Type instruction

Opcode	rs	rt	Immediate
4 bits	3 bits	3 bits	6 bits

Table 5: J-Type instruction

Opcode	Jump address
4 bits	12 bits