Low-Cost and Secure Firmware Obfuscation Method for Protecting Electronic Systems From Cloning

Benjamin Cyr, *Graduate Student Member, IEEE*, Jubayer Mahmod, *Member, IEEE*, and Ujjwal Guin, *Member, IEEE*

***Abstract*—The continuous growth of the cloning of electronic devices poses a severe threat to our critical infrastructure that uses the Internet, as cloned devices can transmit secret information and cause security concerns. Cloned devices can also be unreliable as they may be manufactured with inferior quality materials, and they may have many defects as they may not be tested properly. It is thus extremely important to protect these electronic devices from cloning. An efficient way to prevent a device being cloned is to prevent the firmware from being copied because, without the proper firmware, the device will not function like the original. In this paper, we present a novel firmware obfuscation method without encrypting the entire memory. The firmware is obfuscated by swapping a subset of instructions. The instructions to be swapped are specifically chosen so that an attacker cannot discover their location. During operation, the hardware reconstructs the original program using a physically unclonable function-generated identifier and a small memory that stores the swapped instructions. An adversary cannot make a program work completely without knowing which instructions have been swapped, as the program will execute in the wrong sequence and produce the incorrect result. Our proposed solution requires only a small overhead to reconstruct the firmware, making it practical for devices with strict resource constraints. This solution also allows remote updates of new obfuscated firmware without any modification and is practical for the rising trend of ubiquitous computing.**

***Index Terms*—Cloning, firmware, Internet of Things (IoT), obfuscation.**

I. Introduction

With the advent of ubiquitous computing under the umbrella of Internet of Things (IoT) and cyber-physical systems (CPS), the number of connected electronic devices is expected to grow exponentially in the following decade. Gartner predicted that there will be approximately 20 billion connected devices by 2020 [1]. Widespread use of these edge devices in critical applications (e.g., smart grid, autonomous vehicles, industrial automation, etc.) will present us with unique security challenges. Ensuring security of these wide variety of devices requires immediate intervention. Due to the severe resource constraints, a majority of these devices do not use standard cryptographic protocols to ensure secure operations [2]–[5]. Moreover, the hardware and the firmware running on this system exposed to piracy. An untrusted entity in the supply chain can copy both the hardware and firmware, source them to an untrusted system integrator (SI), and create clones. Any cloned system may have backdoors, which can be exploited for malicious purposes [6], [7]. 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 [8]. 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 Department of Defense 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 [9]. It is practically infeasible to develop a cloned product from the original specification as it requires significant investment in the research and development, 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 [10]. Cloning an electronic system requires the complete reconstruction of the hardware and the firmware. Recently, the hardware become increasingly vulnerable to RE due to the availability of very advanced imaging instruments and powerful characterization tools [10]. 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

Manuscript received August 28, 2018; revised December 17, 2018; accepted December 25, 2018. Date of publication January 1, 2019; date of current version May 8, 2019. This work was supported by the National Science Foundation under Grant CNS 1755733. *(Corresponding author: Jubayer Mahmod.)*

B. Cyr is with the Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: bencyr@umich.edu).

J. Mahmod and U. Guin are with the Department of Electrical and Computer Engineering, Auburn University, Auburn, AL 36849 USA (e-mail: jubayer@auburn.edu; ujjwal.guin@auburn.edu).

Digital Object Identifier 10.1109/JIOT.2018.2890277

2327-4662 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

needs to be low cost, and low resource overhead (area, and power) to be widely accepted to the various IoT and CPS applications.

In this paper, 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 be well-fitted in secure IoT and CPS applications.

A. 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 and Tatschner [11] on an Arm-based system. We focus our attack on The Arm Cortex-M4-based STM32F4 high-performance microcontroller [12]. We will 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 and Tatschner 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.

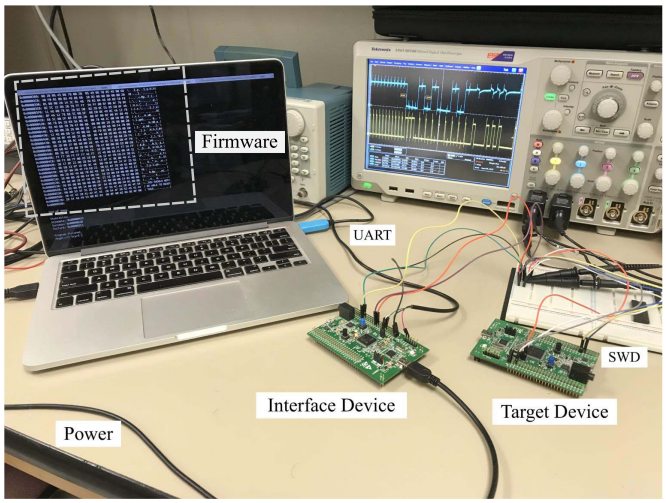

Fig. 1. Experimental setup to extract the firmware from STM32F4.

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 [11] 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.

Fig. 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 are able to 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 [11]. 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.

B. Contributions

In this paper, 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 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 [13]. 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$1000 instructions), with a small number of swaps ($\approx$16), an adversary needs to try approximately 2^{200} trials to make the program completely working, which is infeasible with current computing resources.

The rest of this paper is organized as follows. Section II describes related prior works. Section III describes our proposed obfuscation methodology. The firmware reconstruction methodology is presented in Section IV. The security evaluation and overhead analysis of our proposed approach are presented in Section V. This paper is concluded in Section VI.

II. 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 [14]–[17]. 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.* [18] proposed the integrity verification of peripherals' firmware of a computer system by using remote software-based attestation. LeMay and Gunter [19] developed cumulative attestation kernel to verify the integrity of the firmware over an interval of time. The solution provides the cumulative attestation for memory constraint devices by adding a cumulative attestation coprocessor that handles the computation and storage. To safeguard the firmware against noninvasive attacks, Schellekens *et al.* [20] proposed a solution to protect the persistent state of a trusted module by maintaining an authenticated channel between the trusted module and the memory. Maskiewicz *et al.* [21] proposed a signature verification scheme to prevent the installation of malicious firmware on a mouse. Morais *et al.* [22] 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. Chakraborty *et al.* [23] proposed a key-based control flow obfuscation based on a sequential unlocking mechanism to protect piracy and malicious modification to the embedded software. 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.* [24] developed a hardware-assisted control flow obfuscation, which relies on additional hardware such as shuffle buffer and block address table cache. There are several implementations of Oblivious RAMs proposed by Goldreich and Ostrovsky [25], which obfuscate control flow and patterns of memory accesses [26], [27]. There are also a few designs that have been proposed for FPGAs which encrypt the firmware with PUF-generated keys [28], [29]. One other potential solution was proposed by Guin *et al.* [9] where mutual authentication is performed to prevent system-level cloning. 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.* [30] proposed a hardware and software codependent anti-cloning scheme. Here, 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.* [31] proposed to incorporate a device signature during firmware binary generation. Interdevice signature variation makes the firmware uniquely obfuscated for each of the devices. Digitally reconfigurable PUF has been employed to counter cloning in [32]. In this method, each of the devices would have a different copy of the obfuscated firmware, and it is bound to a 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 [2]–[4]. 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 [2]. 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 and Tatschner [11] showed that memory protection can be bypassed by attacks on debug interfaces or even by modifying security bits with UV-C light.

III. Obfuscation Methodology

The fundamental idea of preventing an adversary to develop cloned systems is to design an obfuscated firmware that runs only on authentic hardware, and an adversary cannot reconstruct the original firmware. 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 constraint devices, the industry is in urgent need for a low-cost solution. In this section, we propose a low-cost solution to prevent system-level cloning.

A. 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 an 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.

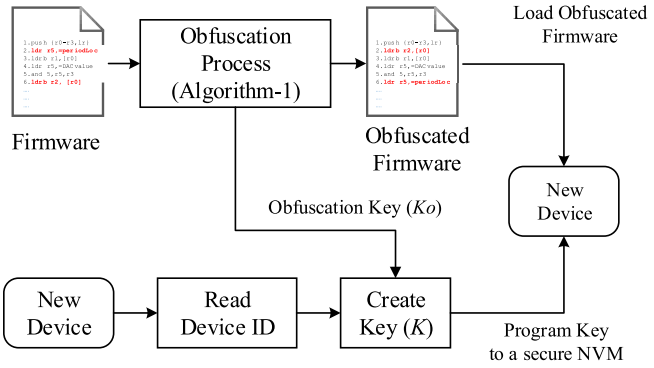

Fig. 2. Proposed flow for creating a clone-resistant electronic device.

Fig. 2 shows the proposed solution to prevent system-level cloning. The trusted SI obfuscates the firmware and loads it into an NVM (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 combined into a single key called the obfuscation key (K_O). If the obfuscation has L swaps, then the obfuscation key would be

$$\begin{aligned}&(\text{Inst}_1 \Longleftrightarrow \text{Inst}_2), \ldots, (\text{Inst}_{2L-1} \Longleftrightarrow \text{Inst}_{2L})\\ &K_O = \left[\text{Addr}_1, \text{Addr}_2, \ldots, \text{Addr}_{2L-1}, \text{Addr}_{2L}\right]. \quad (1)\end{aligned}$$

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

$$|K_O| = |\text{Addr}_i| \times L \times 2. \quad (2)$$

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 PUF [14]–[16] 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 [33] 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.

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_{O_i}, where $k_{O_i} = [RAdd_{Inst_x}\ RAdd_{Inst_y}]$;
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

SI then creates K by using the following equation:

$$K = \mathrm{ID} \oplus K_O. \tag{3}$$

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 IV). The SI needs to keep a database linking the public ID to the private ID (ID) generated by a PUF.

B. 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 SI, and only known to it.

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 III-C). *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 (lines 4–18). 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 III-C). *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 $k_{O_i} = [R\mathrm{Add}_{\mathrm{Inst}_x}\ R\mathrm{Add}_{\mathrm{Inst}_y}]$ (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).

C. Swapping Rule Check 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. In this paper, 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 [34], AVR [35], or PIC [36]). 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 without alerting an attacker. 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.

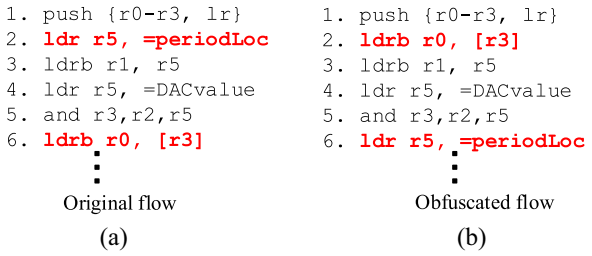

Fig. 3. ARM assembly code snippet as an example for register initialization. Instructions 2 and 6 cannot be swapped.

1) 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).

a) *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.

b) *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.

2) 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).

a) *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.

b) *Register Initialization:* Instructions cannot be swapped into a location where one of the source registers becomes uninitialized. For example, in Fig. 3(a) instruction 3 uses register *r*5. Instruction 2 (i.e., ldr *r*5, = periodLoc) initializes register *r*5. Now, swapping instruction 2 with instruction 6 will give an attacker an indication that instruction 2 has been swapped as *r*3 has no initial value (see Fig. 3).

c) *Register Utilization:* Instructions cannot be swapped into a location where its destination register is never used. For instance, let us assume that register *r*5 is last used in line 6 in Fig. 3(b). If ldr instruction in line 2 in Fig. 3(a) is swapped with ldrb in line 6, the assignment ldr *r*5, = periodLoc becomes redundant and will tip off the attacker because the destination register *r*5 has not been utilized in the obfuscated firmware.

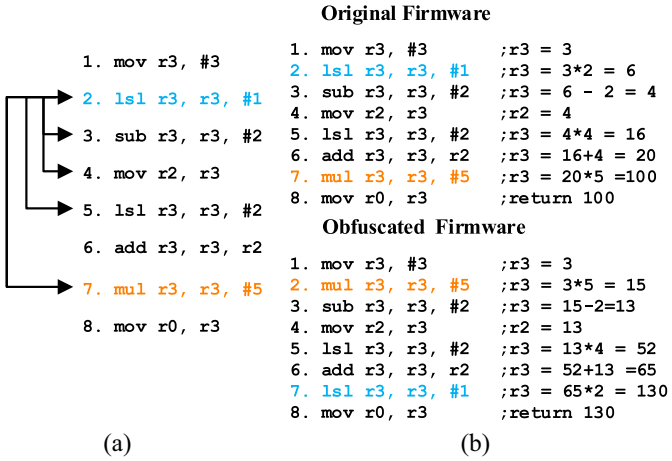

Fig. 4. A simple illustration of obfuscated firmware. (a) A single instruction can swap with many other instructions. (b) When a swap is made, it changes the output of the function.

d) *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 *r*3 , *r*2, #5 and add*r*3 , *r*3, #2 have been placed in consecutive locations after obfuscation. Since, sub *r*3 , *r*2, #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.

e) *Index Distinction:* If the instructions are to be mapped into a cache as described in Section IV, 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.

A simple and comprehensive example of the obfuscation scheme is illustrated in Fig. 4. 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 in eleven different ways according to the SRC. If we randomly choose the swappable instruction *ls*1*r*3, *r*3, *r*2 (blue), then we can make four possible swaps that adhere to the SRC [Fig. 4(a)]. We then randomly choose sub *r*3, *r*3, #1 (green) as the other instruction to swap with. Fig. 4(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.

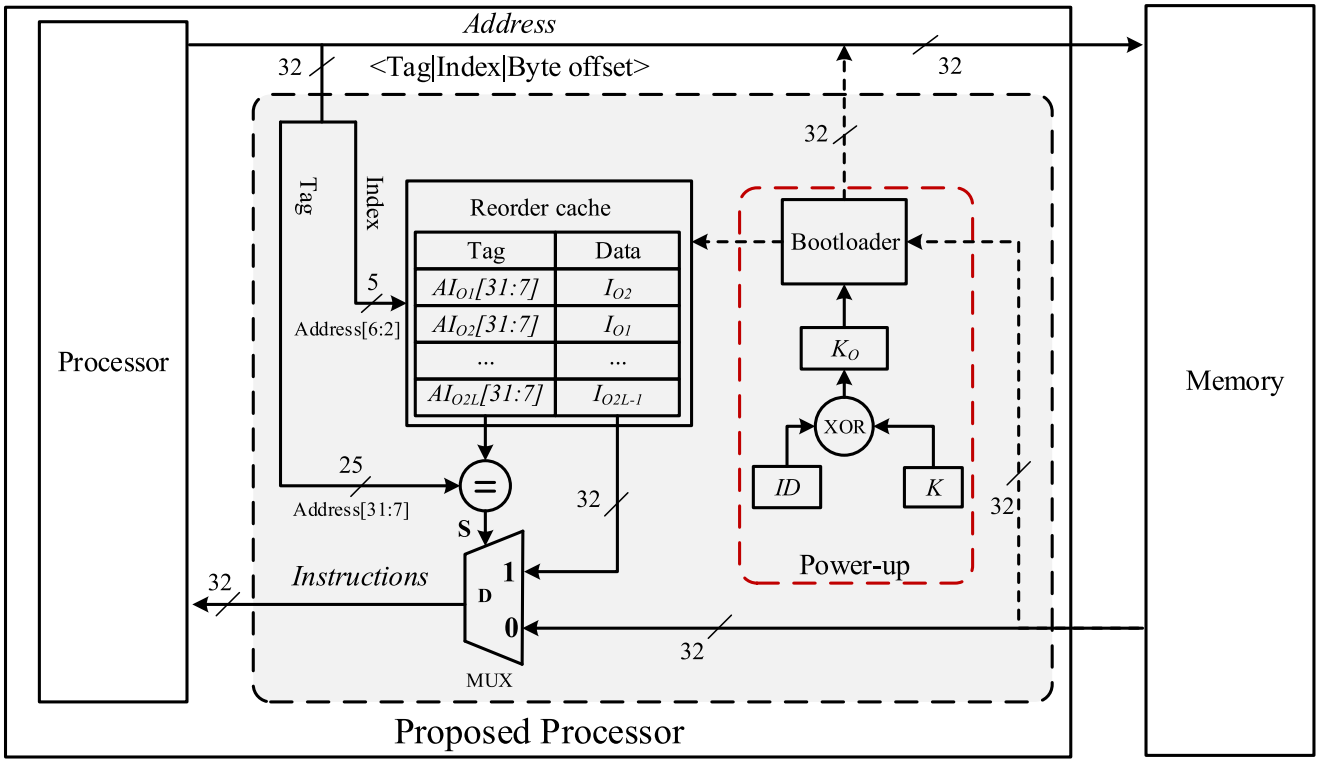

Fig. 5. Proposed scheme for firmware reconstruction during program execution.

IV. Reconstruction Methodology

Since the obfuscated firmware is stored in the NVM, 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.

Fig. 5 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 NVM [37]. 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 Fig. 5 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 [38]. The cache contains all the swapped-instruction pairs and the tags of their corresponding relative addresses. In this paper, 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 five address bits to represent index. We also consider instructions are of four 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 [39], [40]. In general, the SI 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

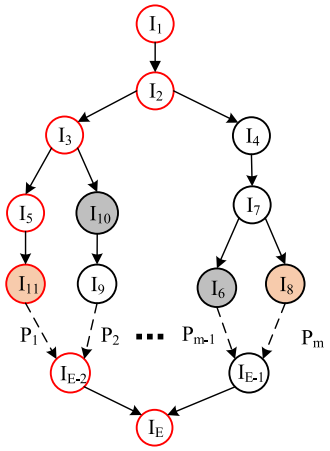

Fig. 6. DAG of a firmware. The path, P_1 highlighted in red, represents a failing execution. Instruction I_8 is swapped with I_{11}.

update, the original manufacturer or 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.

V. Analysis and Overhead

In this section, we develop a mathematical model to analyze the security of the proposed obfuscation method. We also provide an estimate of the area overhead of our proposed scheme.

A. 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 AE, the firmware is modeled as a directed acyclic graph (DAG) [9], [23]. 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 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. Fig. 6 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

$$\mathrm{AE}_T = \mathrm{AE}_F + \mathrm{AE}_U. \tag{4}$$

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 Fig. 6 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 (5) shows how many trials an attacker must run to check a single swap in M failing paths

$$\begin{aligned}\mathrm{AE}_F = {} & h_1(N_T - 1) + h_2(N_T - 3) \\ & + \cdots + h_M(N_T - 2M + 1)\end{aligned} \tag{5}$$

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 us 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

$$L = L_T - M \tag{6}$$

$$N = N_T - 2M. \tag{7}$$

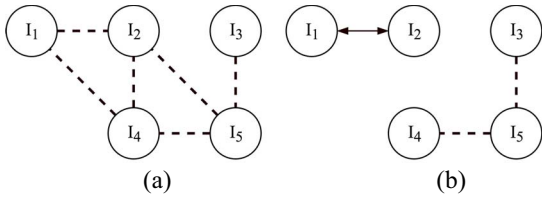

Fig. 7. Graph model of two instructions being swapped in firmware. (a) Possible swaps are shown with dotted lines. (b) The edges adjacent to the swapped instructions are no longer valid when a swap is chosen.

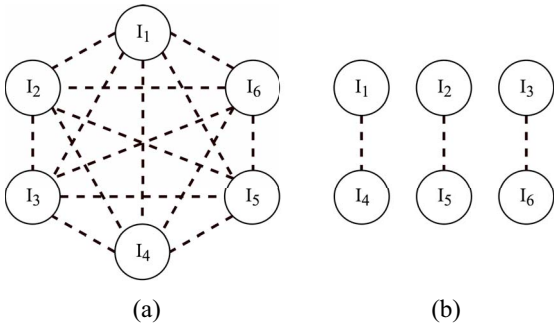

Fig. 8. (a) Worst-case and (b) best-case graphs for the adversary when $N = 6$.

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 AE. 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 III. 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 Fig. 7, where Fig. 7(a) shows all the potential swaps in a firmware with five swappable instructions. After I_1 and I_2 are swapped in Fig. 7(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 nonadjacent (disjoint) edge to find the second swap.

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 [41]. Still, using the k-edge matching for special graphs, the upper bound and lower bounds for the AE can be calculated. Fig. 8 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" [42]. 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

$$\begin{aligned} AE_{U-W} &= \left(\frac{1}{L!}\right)\binom{N}{2}\binom{N-2}{2}\cdots\binom{N-2L+2}{2} \\ &= \frac{N!}{2^L(L)!(N-2L)!} \\ &= \frac{N(N-1)\cdots(N-2L+1)}{2^L(L)!}. \end{aligned} \tag{8}$$

Equation (8) 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

$$\begin{aligned} AE_{U-B} &= {}^{N/2}C_L \\ &= \binom{N/2}{L} \\ &= \frac{N(N-2)\cdots(N-2L+2)}{2^L(L)!}. \end{aligned} \tag{9}$$

Equation (9) 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, $\mathrm{AE}_U \gg \mathrm{AE}_F$ for even small values of L, we can ignore the AE_F term and estimate AE_T to be approximately equal to AE_U

$$\mathrm{AE}_T \approx \mathrm{AE}_U. \tag{10}$$

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 IV, the index distinction rule will apply (see Section III-C). 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

$$\mathrm{AE}_{T-W} \approx \frac{(32-2M)!}{32^{2L}} AE_{U-W} \tag{11}$$

$$\mathrm{AE}_{T-B} \approx \frac{\prod_{i=1}^{L}(2i-1)}{32^L}\mathrm{AE}_{U-B} \tag{12}$$

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 III. The script parses through the ARM assembly file that is generated by the GNU Arm Embedded Compiler [43] and locates the instructions that are swappable. Then, it counts up the number of possible swaps between all

TABLE I
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}

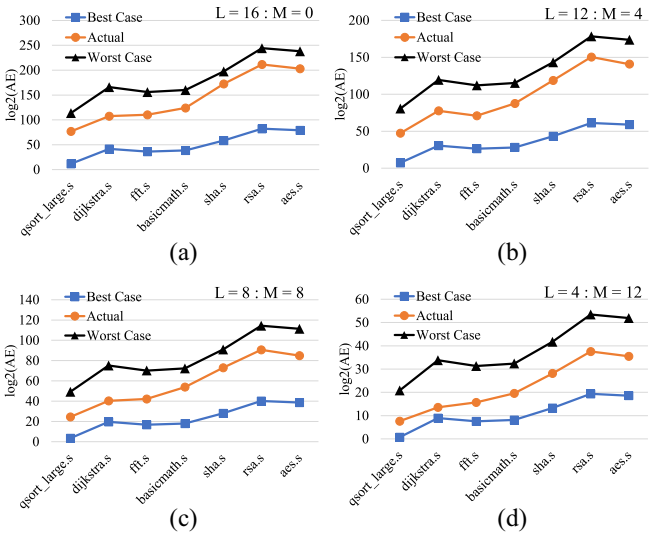

Fig. 9. Comparison of actual and estimated attacker's effort (AE_T). Worst-case and best-case AEs are estimated from (11) and (12), respectively. (a) L=16; $M = 0$. (b) $L = 12$; $M = 4$. (c) $L = 8$; $M = 8$. (d) $L = 4$; $M = 12$.

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 I shows the actual attacker's effort, which is calculated using the Python script describe above.

The example programs listed were benchmark tests from MiBench2 [44]. Here, columns 4–7 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 (6)]. 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.

Fig. 9 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 (11) and (12), respectively. The actual value is the effort calculated by the Python script. The vertical axes of all the graphs are on the logarithmic scale. Fig. 9(a) shows the AE_T when an attacker cannot find any failing paths ($L = 16$) for the firmware. We expect the actual value should be lower and upper bounded by the best-case and worst-case estimate of the attacker's effort, respectively. Fig. 9(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.

B. 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 [8] 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 Fig. 4 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 [Fig. 4(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.

C. 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 paper. Nevertheless, we can provide a close approximation of gate counts essential to the design. Maintaining consistency with the previous discussion let us assume 1024-bit ID, 32 cache lines with 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 NVM. Note that, we do not need to require any overhead to generate the ID as system's SRAM can be used as a PUF. The logic overhead is insignificant considering the size of modern embedded processors that are common in IoT devices.

VI. Conclusion

In this paper, 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 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. 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. Our future work will address the selection of an instruction for a possible swap, such that an adversary cannot find a failing program execution. In addition, we plan to implement our proposed scheme into a custom chip and further test the effectiveness of this obfuscation in preventing system-level cloning.

References

[1] R. van der Meulen, *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*, Gartner, Stamford, CT, USA, 2017.

[2] 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, Jan./Feb. 2015.

[3] K. Rawlinson. (2014). *HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack*. Accessed: Mar. 2018. [Online]. Available: http://www8.hp.com/us/en/hp-news/press-release.html?id=1744676.WUrrwWgrKM8

[4] M. B. Barcena and C. Wueest, *Insecurity in the Internet of Things, Security Response*, Symantec, Mountain View, CA, USA, 2015.

[5] M. Alam, M. M. Tehranipoor, and U. Guin, "Tsensors vision, infrastructure and security challenges in trillion sensor era," *J. Hardw. Syst. Security*, vol. 1, no. 4, pp. 311–327, 2017.

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

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

[8] J. Robertson and M. Riley. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. Accessed: Nov. 2018. [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

[9] U. Guin, S. Bhunia, D. Forte, and M. M. Tehranipoor, "SMA: A system-level mutual authentication for protecting electronic hardware and firmware," *IEEE Trans. Depend. Secure Comput.*, vol. 14, no. 3, pp. 265–278, May/Jun. 2017.

[10] S. E. Quadir *et al.*, "A survey on chip to system reverse engineering," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 1, p. 6, 2016.

[11] J. Obermaier and S. Tatschner, "Shedding too much light on a microcontroller's firmware protection," in *Proc. 11th USENIX Workshop Offensive Technol. (WOOT)*, 2017, pp. 1–47.

[12] *STM32F4 Series of High-Performance MCUs With DSP and FPU Instructions*. Accessed: Mar. 2018. [Online]. Available: http://www.st.com/en/microcontrollers/stm32f4-series.html?querycriteria=productId=SS1577

[13] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1198–1210, Sep. 2009.

[14] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," in *Proc. ACM Conf. Comput. Commun. Security*, 2002, pp. 148–160.

[15] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Proc. ACM/IEEE Design Autom. Conf.*, 2007, pp. 9–14.

[16] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFS and their use for IP protection," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2007, pp. 63–80.

[17] 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 *Proc. Int. Conf. VLSI Design*, 2018, pp. 85–90.

[18] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the integrity of peripherals' firmware," in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 3–16.

[19] M. LeMay and C. A. Gunter, "Cumulative attestation kernels for embedded systems," *IEEE Trans. Smart Grid*, vol. 3, no. 2, pp. 744–760, Jun. 2012.

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

[21] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse trap: Exploiting firmware updates in USB peripherals," in *Proc. 8th USENIX Conf. Offensive Technol. USENIX Assoc.*, 2014, p. 12.

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

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

[24] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande, "Hardware assisted control flow obfuscation for embedded processors," in *Proc. Int. Conf. Compilers Archit. Synth. Embedded Syst.*, 2004, pp. 292–302.

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

[26] E. Stefanov *et al.*, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, 2013, pp. 299–310.
[27] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On tightness of the Goldreich–Ostrovsky lower bound," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, 2015, pp. 850–861.
[28] A. Schaller, T. Arul, V. van der Leest, and S. Katzenbeisser, "Lightweight anti-counterfeiting solution for low-end commodity hardware using inherent PUFs," in *Proc. Int. Conf. Trust Trustworthy Comput.*, 2014, pp. 83–100.
[29] M. A. Gora, A. Maiti, and P. Schaumont, "A flexible design flow for software IP binding in commodity FPGA," in *Proc. IEEE Int. Symp. Ind. Embedded Syst.*, Jul. 2009, pp. 211–218.
[30] R. P. Lee, K. Markantonakis, and R. N. Akram, "Binding hardware and software to prevent firmware modification and device counterfeiting," in *Proc. 2nd ACM Int. Workshop Cyber Phys. Syst. Security*, 2016, pp. 70–81.
[31] J. X. Zheng, D. Li, and M. Potkonjak, "A secure and unclonable embedded system using instruction-level PUF authentication," in *Proc. IEEE 24th Int. Conf. Field Program. Logic Appl. (FPL)*, 2014, pp. 1–4.
[32] J. X. Zheng, T. Xu, and M. Potkonjak, "Securing embedded systems and their IPs with digital reconfigurable PUFS," in *Proc. IEEE 26th Int. Workshop Power Timing Model. Optim. Simulat. (PATMOS)*, 2016, pp. 169–176.
[33] W. Wang, A. Singh, U. Guin, and A. Chatterjee, "Exploiting power supply ramp rate for calibrating cell strength in SRAM PUFs," in *Proc. IEEE Latin–Amer. Test Symp.*, 2018, pp. 1–6.
[34] C. Lomont. (2012). *Introduction to X64 Assembly*. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-x64-assembly
[35] *AVR Instruction Set Manual*, Atmel, San Jose, CA, USA, Nov. 2016.
[36] *PICmicro Mid-Range MCU Family Reference Manual*, Microchip, Chandler, AZ, USA, Dec. 1997.
[37] Albert Kennedy Trust Support. (2018). *ARM: How to Write a Bootloader*. [Online]. Available: http://www.keil.com/support/docs/3913.htm
[38] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publ. Inc., 2011.
[39] *STM32 Microcontroller System Memory Boot Mode Transceiver, Rev. 32*, STMicroelectronics, Geneva, Switzerland, Feb. 2018.
[40] J. Beningo. (2018). *Update Firmware in the Field Using a Microcontrollers DFU Mode*. [Online]. Available: https://www.digikey.com/en/articles/techzone/2018/jan/update-firmware-field-using-microcontroller-dfu-mode
[41] L. Lovaísz, *Matching Theory*. New York, NY, USA: North–Holland, 1986.
[42] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley, 1998, pp. 65–67.
[43] *GNU Arm Embedded Toolchain*. Accessed: Mar. 2018. [Online]. Available: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads
[44] M. R. Guthaus *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization*, 2001, pp. 3–14.

Benjamin Cyr (GS'18) received the B.E. degree from the Department of Electrical and Computer Engineering, Auburn University, Auburn, AL, USA, in 2017. He is currently pursuing the Ph.D. degree at the Computer Science and Engineering Department, University of Michigan, Ann Arbor, MI, USA.

His current research interests include hardware and embedded systems security.

Jubayer Mahmod (S'17–M'18) received the B.Sc. degree in electrical and electronic engineering from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in 2016. He is currently pursuing the M.S. degree at the Department of Electrical and Computer Engineering, Auburn University, Auburn, AL, USA.

He was an Intern Engineer with the Hardware Design Team, ForteMedia, Inc., Santa Clara, CA, USA, in 2018. His current interests include hardware security, very large-scale integration design, and Internet of Things security.

Ujjwal Guin (S'10–M'16) received the B.E. degree from the Department of Electronics and Telecommunication Engineering, Bengal Engineering and Science University, Howrah, India, in 2004, the M.S. degree from the Department of Electrical and Computer Engineering, Temple University, Philadelphia, PA, USA, in 2010, and the Ph.D. degree from the Electrical and Computer Engineering Department, University of Connecticut, Mansfield, CT, USA, in 2016.

He is currently an Assistant Professor with the Electrical and Computer Engineering Department, Auburn University, Auburn, AL, USA. He co-authored *Counterfeit Integrated Circuits-Detection and Avoidance* (Springer, 2015). He has authored several journal articles and refereed conference papers. He has been actively involved in developing a Web-based tool, Counterfeit Defect Coverage Tool, to evaluate the effectiveness of different test methods used for counterfeit IC detection. SAE International has acquired this tool from the University of Connecticut. He has developed several on-chip structures and techniques to improve the security, trustworthiness, and reliability of integrated circuits. His current research interests include hardware security and trust, supply chain security, cybersecurity, and very large-scale integration design and test.

Dr. Guin is an active participant on the SAE International's G-19A Test Laboratory Standards Development Committee. He is a member of the ACM.