ConFirm: Detecting Firmware Modifications in Embedded Systems using Hardware Performance Counters

Xueyang Wang[*], **Charalambos Konstantinou**[*], **Michail Maniatakos**[†], **Ramesh Karri**[*]
[*]Department of Electrical and Computer Engineering, Polytechnic School of Engineering, New York University
[†]Department of Electrical and Computer Engineering, New York University Abu Dhabi
E-mail: {xueyang.wang, ckonstantinou, michail.maniatakos, rkarri}@nyu.edu

***Abstract*—The modernization of various critical infrastructure components has dictated the use of microprocessor-based embedded control systems in critical applications. It is often infeasible, however, to employ the same level of security measures used in general purpose computing systems, due to the stringent performance and resource constraints of embedded devices. Furthermore, as software relies on the firmware for proper operation, no software-level technique can detect malicious behavior of the firmware. In this work, we propose ConFirm, a low-cost technique to detect malicious modifications in the firmware of embedded systems by measuring the number of low-level hardware events that occur during the execution of the firmware. In order to count these events, ConFirm leverages the Hardware Performance Counters (HPCs), which readily exist in many embedded processors. We evaluate the detection capability and performance overhead of the proposed technique on various types of firmware running on ARM- and PowerPC- based embedded processors. Experimental results demonstrate that ConFirm can detect all the tested modifications with low performance overhead.**

I. Introduction

Embedded devices are an integral part of the connected world, used in a wide variety of applications such as simple home automation systems, mobile phones, and mission critical industrial control systems. For instance, the digitization of smart grid control technologies leads to the formation of power systems substantially dependent on embedded devices [1]. The market for embedded systems is expected to generate over $2 trillion in revenue during 2015 [2]. In addition, as the notion of the Internet-of-Things (IoT) continues to evolve, embedded systems as IoT components will dominate the digital planet [3].

Firmware is the layer between the hardware and software of a device, controlling physical components operation based on user-defined programs and other operating information. It basically incorporates the instructions and data necessary to make an embedded device functional. Consequently, given the critical role of firmware, implementation of effective security controls against firmware malicious actions is essential.

To date, numerous mechanisms have been proposed and implemented in order to detect malicious firmwares. Each implementation requires either extra hardware components (e.g. Trusted Platform Module) or verification methodologies, which typically have significant impact on the resource-constrained embedded device (e.g. computation resources, communication bandwidth, power consumption, memory usage, etc.).

In order to overcome the challenges and constraints of existing detection schemes, we propose a novel low-cost firmware modification detection technique called ConFirm. The key idea of ConFirm is based on the observation that a program is composed of a sequence of various types of instructions which during execution can be seen as low-level hardware events. Therefore, the behavior of a program can be uniquely characterized by monitoring the total occurrences of specified hardware events during its execution, as well as the relationship between the occurrences of different monitored events. Such hardware events include retired instructions, branches, returns, floating point operations, etc. In case of firmware malicious modifications, the occurrences of these executed hardware events will be different compared to legitimate firmware runs.

The aforementioned low-level hardware events are efficiently measured using Hardware Performance Counters (HPCs). HPCs count various event types based on the processor architecture and its Performance Monitoring Unit (PMU). Embedded systems based on ARM, MIPS, PowerPC and many other processor architectures include several event types for the performance counters [4]. Although HPCs are typically used for performance tuning, ConFirm leverages performance counters for security purposes. Therefore, ConFirm comes at no extra hardware cost and can be ported to legacy devices that support HPCs. Additionally, since events are automatically counted by HPCs at the hardware level, the performance overhead of our detection technique remains significantly low. This is crucial for embedded systems with real-time requirements. Summarizing, our contributions are as follows:

- We present ConFirm, a host-based verification tool that leverages existing hardware features (HPCs) to detect malicious modifications in embedded systems.
- We implement a prototype of ConFirm on ARM- and PowerPC-based platforms. We demonstrate the feasibility of the proposed technique with two real-world firmwares and attacks. The performance and storage overhead on the monitored system are also evaluated.

The rest of the paper is organized as follows: Section II discusses the prerequisites of using ConFirm. Section III presents the overview of ConFirm design as well as implementation details. The evaluation results are shown in Section IV. Related work on firmware attacks and detection mechanisms are given in Section V. Finally, we conclude the paper in Section VI.

II. Prerequisites

In this section, we introduce the adversarial model used throughout the study, as well as the essential requirements for deploying ConFirm on an embedded device.

A. Threat Model

We target attacks that alter the firmware code of any embedded system. Particularly, we perform firmware modifications able to inject and execute malicious code or call existing functions not included in the code flow execution, similar to common system hijacking. Our threat model allows a sophisticated attacker to have access to exploits that allow execution of arbitrary malicious code. For example, the attacker can launch a simple code injection attack as well as a code reuse attack such as the return-to-libc or return oriented attack. These exploits can be based on either simple firmware code alternations or even zero-day firmware vulnerabilities. Finally, the attack can be performed either offline or online: The offline attack requires device reboot after the attacker uploads the malicious firmware image. On the other hand, the firmware modifications could be injected to the embedded system through a firmware update (online attack). Depending on the embedded device, the malicious firmware image or firmware update can be transferred to the system either locally (physical access) or remotely.

B. Hardware Performance Counters

HPCs are a set of special-purpose registers built into the PMU of a modern microprocessor in order to store information about hardware-related activities [5], [6]. Event selectors specify the user-defined choice of hardware events to monitor. Since performance counters were originally designed for performance debugging of complex software systems, software developers heavily rely on HPC-based profilers in order to understand the runtime behavior of a program and tune its performance.

HPCs provide access to detailed performance information with much lower overhead than software profilers and much higher accuracy. Furthermore, HPCs do not require any source code modifications. The number of available counters, as well as the number of hardware events, vary from one processor model to another. For example, the ARM V8 Cortex-A53 core has 4 HPCs, able to count 62 events [7]. Similarly, the fouth generation of 32-bit PowerPC microprocessors (such as the PowerPC 7450) incorporates over 35 events and 4 HPCs [8].

1) Computational Path Analysis with HPCs: A subroutine of the firmware generally has multiple computational paths in its control flow graph. Each computational path goes through different code blocks (or phases) and therefore executes different code. The executed code generates non-identical vectors in terms of hardware events occurrences. A computational path is primarily determined by the input applied to the subroutine and the state of data structures used by the subroutine. Let $C(E_x)$ denote the count of event x from the execution of a targeted computational path. If m hardware events are monitored simultaneously, an HPC vector V with m elements can be obtained as follows:

$$V = [C(E_1), C(E_2), C(E_3), ..., C(E_m)] \quad (1)$$

With a selected set of hardware events, a vector can be consistent for every execution as discussed in detail in Section III-D (except the situation that the path contains a loop with a dynamic number of iterations, where a measurement only targeting the loop is required). As a result, each vector can be considered as the signature of a specific path. ConFirm uses such a HPC-based signature to verify the execution of computational paths in a subroutine of the monitored firmware. The computational path analysis overview is presented in Figure 1.

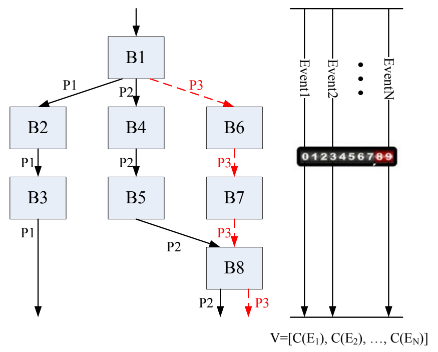

Fig. 1: Computational path analysis with HPCs. The execution of the valid paths P1 and P2 in a monitored subroutine generates different vectors V_1 and V_2 in terms of the occurrences of low-level hardware events. As an example, malicious execution could go through path P3, generating an unexpected HPC vector V_3 which is different from V_1 and V_2.

A wide range of common attacks include control-flow modifications. In other words, an attacker typically needs to hijack the original control-flow of the victim program in order to perform any malicious actions [9]. This is achieved by executing code or calling functions not included in the code flow execution. This is an essential step in many exploits such as stack-based buffer-overflow attacks and heap-based “jump-to-libc” attacks [10], [11]. Enforcing Control-Flow Integrity (CFI) as a defensive technique can thwart such attacks based on the illegal control transfers. However, CFI techniques are generally expensive with overheads that may be as high as 25% [12] to 50% [13]. In addition, CFI techniques are unable to perform instrumentation of modules separately [14]. They typically require all modules of a system (libraries included) to be available at instrumentation time.

C. Root of Trust

The booting of an embedded system, similar to general purpose computing systems, is a multi-phase process. Each phase is responsible for loading the next phase. The first code that is executed in this *boot sequence* resides in a part of memory that is protected from accidental erasure or corruption and is typically called boot Read-Only-Memory (ROM). The boot ROM code locates and executes the second software phase, called the pre-loader. The pre-loader is used to initialize memory and an appropriate subset of the peripheral devices in order to access and load the next stage loader (boot-loader) into main memory. The boot-loader is responsible for locating and loading the final OS or firmware which will run the embedded system application on the device. If the embedded device is a non-OS based system, then the firmware execution tasks are executed in an infinite loop (super loop based execution). Because the three first booting phases (boot ROM, pre-loader and boot-loader) reside in a reserved read- only area of the

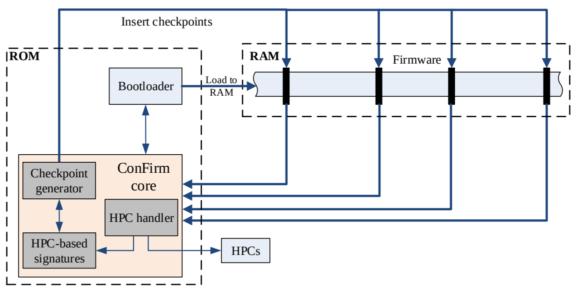

Fig. 2: High-level structure of ConFirm. The ConFirm core consists of three components: an insertion module that inserts checkpoints to the monitored firmware, an HPC handler that drives the HPCs and a database that stores valid HPC-based signatures.

system, they constitute the root of trust of the embedded system [15].

Since write-protected memories are secure and protected by design, we leverage the root of trust by incorporating ConFirm HPC profiling-based scheme within the bootloader. As a result, ConFirm remains in a trusted area, immune to malicious alterations.

III. ConFirm Overview

The high-level structure of ConFirm is shown in Figure 2. A legacy boot-loader is extended with the ConFirm core. The core consists of three components: a) an insertion module that places checkpoints to the monitored firmware, b) an HPC handler that drives the HPCs, and c) a database that stores valid HPC-based signatures. All these components are stored in write-protected non-volatile memory. This prevents attacks from compromising ConFirm while still allowing authorized updates on ConFirm when the monitored firmware needs to get updated[1].

The advantages of employing ConFirm can be summarized as follows:

- ConFirm core resides in the bootloader, thus it is difficult to be detected or deactivated by an adversary.
- ConFirm monitoring can be instrumented within any arbitrary executable regardless of its functionality or location within the system stack.
- ConFirm directly utilizes the hardware features of the host platform, bypassing the layers of overhead produced by operating systems or Virtual Machines (VMs) hosting an OS.
- ConFirm is self-sufficient and does not depend on facilities provided by the monitored firmware. ConFirm treats firmware as an untrusted and foreign entity.

[1]Similar to general-purpose computers using BIOS for booting, an embedded system bootloader can only be updated outside the nominal operation of the device.

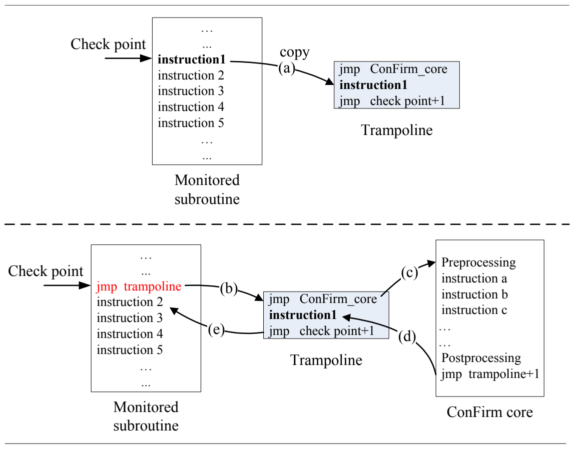

Fig. 3: Control-flow interception in ConFirm. A Detour style inline hooking mechanism is employed to redirect the control-flow to the ConFirm when the execution of the firmware reaches a checkpoint.

A. *ConFirm Modules*

When the firmware image is loaded into memory and before its execution, the boot-loader transfers the control to ConFirm. The checkpoint generator of ConFirm automatically inserts checkpoints to pre-determined locations all across the firmware binary. The firmware validation starts when the first checkpoint is reached. A pair of two adjacent checkpoints form a check window during which the execution of the firmware is monitored. Each checkpoint is the ending point of the previous check window and the stating point of the next check window. Hence, the check windows are allocated continuously without any gaps in between and following the control-flow of the firmware code. More details about checkpoint insertion are presented in Section III-B.

After the checkpoint insertion is completed, the control-flow proceeds to the HPC handler. In this stage, a small sequence of instructions are executed in order to configure, initialize and enable the HPCs. With the HPCs enabled, the control is transferred back to the boot-loader which initiates the execution of the firmware.

Once the execution reaches a checkpoint, ConFirm intercepts the control-flow and redirects it to the core module. The core then communicates with the HPC handler and the HPC-based signature database. Specifically, the event-counts for the previous check window are read and compared with the corresponding signatures in the database. Then the HPCs are reset for the next check window and the execution in the monitored firmware is continued. The HPCs keep counting the occurrences of the hardware events until the next checkpoint is reached.

B. *Checkpoint Insertion*

The current implementation of ConFirm uses a Detours style inline hooking mechanism for control-flow interception and checkpoint insertion [16]. The checkpoint insertion technique is shown in Figure 3. In the monitored firmware, the instruction at a checkpoint is first copied and preserved in a

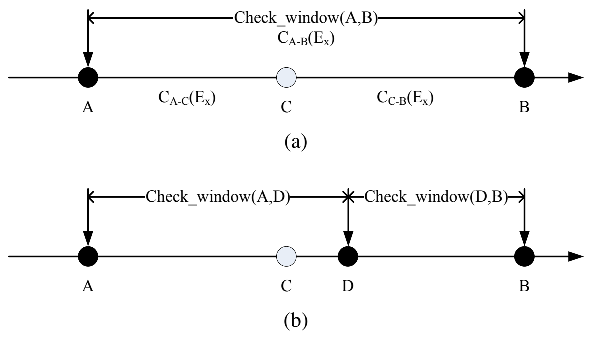

Fig. 4: Checkpoint randomization to avoid prediction of check windows.

certain memory location in RAM which is called a "trampoline" (arrow (a) in Figure 3). Then, the firmware instruction is replaced with a jump instruction to transfer the control to the trampoline (b). In the trampoline, another jump instruction is executed that targets a location in the ConFirm core module (c). The sequence of instructions in the Confirm core module ends with a jump back to the trampoline after the check is completed (d). The copied instruction which is replaced in the original subroutine, is executed in the trampoline before the control is transferred back to the monitored subroutine (e). The next instruction to be executed in the subroutine is the one right after the replaced instruction. With such an inline hooking, the functionality of the subroutine remains the same after applying the check. For each checkpoint, a dedicated trampoline needs to be allocated since the instruction copied and preserved for each checkpoint is different. Also, the jump instruction in different trampoline targets different locations in the ConFirm core module.

1) Checkpoint Randomization: Checkpoints are inserted at the entry and exit points of each monitored subroutine. This opens up the opportunity to an attacker to manipulate the counters, evading detection. Figure 4 (a) shows an example in which an attacker can bypass the check mechanism if he knows the exact checkpoint locations. Points A and B are the entry and exit points of a computation path $P(A,B)$ of a monitored subroutine. Thus, two checkpoints are inserted at locations A and B. Location C is an intermediate point of the computation path $P(A,B)$.

Assume that the attacker modifies the control-flow between A and C. The count of Event x from A to C after the modification is $C_{A-C}(E_x)$. If the attacker knows the "good" value of $C_{A-B}(E_x)$ (the count of the whole check window (A,B)), he can further modify the control-flow between C and B to make $C_{A-C}(E_x)+C_{C-B}(E_x)$ equal to $C_{A-B}(E_x)$, and thus the modification will not detected. To prevent attackers from predicting the check windows, checkpoints are also inserted at arbitrary locations in the monitored subroutines (besides the entry and exit points), as shown in Figure 4 (b). Such locations can be changed every time the system reboots or when the control is transferred at runtime to the ConFirm core module. Specifically, the insertion module maintains multiple sets of locations where the checkpoints can be inserted. When the system boots one set of locations is selected randomly to apply the placing of the checkpoints. Besides the boot-time checkpoint injection, the insertion module selects a new set of locations and updates the checkpoints during runtime (after certain time periods). The addresses of trampolines are also updated when a new set of checkpoints is applied.

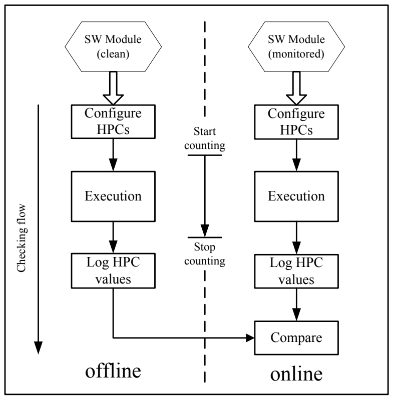

Fig. 5: Offline profiling phase (left-hand side) and online checking phase (right-hand side) of ConFirm comparison-based two-phase detection.

C. Comparison-based Detection

ConFirm performs a comparison-based two-phase detection as shown in Figure 5. In the offline profiling phase, the HPC signatures of all monitored execution paths are generated from a clean copy of the target firmware. In the online checking phase, the same monitored paths are measured and the runtime signatures are compared with the corresponding golden ones.

1) Offline Profiling: This phase is performed before the device is deployed. The clean copy of the firmware is executed on the device with check points inserted into pre-determined locations, splitting the execution flow into multiple check windows. During each check window, the occurrences of m monitored hardware events for a computational path are measured and the HPC vector $V = [C(E_1), C(E_2), C(E_3), ..., C(E_m)]$ is obtained. The measurement is performed for all the n possible paths in the check window to complete the HPC signature of a check window $W = [V_1, V_2, V_3, ..., V_n]$. After total x check windows have been profiled, the HPC-based signature of the monitored firmware $S = [W_1, W_2, W_3, ..., W_x]$ is generated and stored in the database as the golden reference signature.

2) Online Checking: When a check is invoked during the online checking phase, ConFirm reads all the configured counters and obtains the runtime HPC vector V for the current check window. Then, ConFirm refers to the signature database and retrieves the reference vectors generated offline for the same check window. The runtime HPC vector V is matched with the offline references. If a match is found, indicating the tested vector V is valid, the control returns to the monitored firmware and execution resumes. In case the runtime vector does not match to any of the offline references, a deviation will be reported by ConFirm and appropriate actions, specified by the integrator, will be initiated. Possible reactions to anomaly detection include rebooting the system, generating an alarm, disabling the device, etc.

TABLE I: The hardware events of ARM Cortex A15 (a) and PowerPC e300c3 (b) with small C.V against system disturbances. A smaller C.V indicates a better repeatability of an HPC-based signature.

(a)

Hardware event	C.V (%)
Branch Instruction executed	0.72
Instruction architecturally executed	0.93
Branch speculatively executed, return	1.07
Instruction speculatively executed, store	1.27
Instruction speculatively executed, load	1.27
Average over all tested events (∼70)	**18.9**

(b)

Hardware event	C.V (%)
Branch Instructions completed	1.05
Completed Instructions	1.13
Load micro-ops completed	1.59
Store micro-ops completed	1.78
Branch instructions mispredicted	2.35
Average over all tested events (∼40)	**16.7**

D. HPCs Selection

Recent models of mainstream processors have numerous architectural/microarchitectural hardware events available for monitoring. The hardware events on a platform can be categorized into different groups, such as load/store events, cache events, execution unit events, etc. This gives us many options to model an HPC signature. However, not all the available hardware events are good candidates for modeling a recognizable signature, as the occurrence of some events depends on the current, unpredictable system state. For such events, the number of occurrences during execution vary dramatically given different executions. Extracted signatures must be repeatable and robust. Therefore, it is important to carefully select proper hardware events whose occurrences are more consistent every time when the targeted program is executed. Experimental results presented in Section IV-A elaborate on the selection of proper HPCs for the target firmwares.

IV. Experimental Results

In this section we demonstrate the detection capability of ConFirm. We also evaluate the performance and storage overhead when ConFirm is enabled.

A. ConFirm Detection Capability

To demonstrate the effectiveness of ConFirm we test our technique with two real-world firmwares of embedded systems on two different platforms: a Samsung Exynos Arndale board [17] and a Freescale MPC8308RDB platform [18]. The Arndale board contains an ARM Cortex-A15 processor implemented with 6 HPCs and can support about 70 different hardware events [19]. The MPC8308RDB platform contains a PowerPC e300c3 core which has 4 HPCs with around 40 available hardware events [20].

In order to determine which hardware events are more robust for signature modeling, we execute multiple times a set of software modules on the same platform. The repeatability is quantified with the Coefficient of Variation (C.V). Table I lists the hardware events with the smallest C.V for ARM Cortex A15 and PowerPC e300c3. These events are the most robust thus they are considered as good candidates for ConFirm.

In this experiment, we perform proof-of-concept malicious modifications to a monitored subroutine for each firmware. We then measure the event count deviations of the malicious path from all the valid paths in the monitored subroutines. Let $C_{P_{test}}(E_x)$ denotes the count of event E_x from the execution of a path under test P_{test}. Similarly, $C_{P_{ref\_y}}(E_x)$ denotes the event counts E_x from the execution of a valid reference path P_{ref_y}. The deviation of P_{test} from P_{ref_y} on event E_x is presented as follows:

$$D(E_x)(P_{test}, P_{ref\_y}) = \frac{C_{P_{test}}(E_x) - C_{P_{ref\_y}}(E_x)}{C_{P_{ref\_y}}(E_x)} \quad (2)$$

Considering the system disturbances, it is required to set a noise threshold N. In case $D(E_x)(P_{test}, P_{ref\_y})$ is greater than N, ConFirm suggests a malicious modification. Assume x subroutine events are monitored concurrently, where $1 \leq x \leq m$. If for all these events $D(E_x)(P_{test}, P_{ref\_y})$ is less than N, P_{test} is matched to the valid path P_{ref_y}. If no match is found in the checking procedure, a malicious modification will be reported by ConFirm. The results of the deviations $D(E_x)(P_{test}, P_{ref\_y})$ as well as the found threshold N are presented below for two commercial embedded systems[2].

1) Case Study 1: Wireless Access Point Firmware: We first test ConFirm with a commercial ARM-based firmware. The embedded device is an all-in-one wireless access point and access gateway specifically designed for public hot-spot providers and enterprises. The firmware contains a VxWorks real-time operating system widely used in embedded systems [21].

In this experiment a Denial of Service (DoS) modification attack is performed [22]. The attack targets the task scheduling module of the firmware. Specifically, we add a function hook to the *checkTaskSwitch* subroutine to modify the normal control-flow of the task scheduling algorithm. When a task with a specific ID is running it will occupy the processor without being switched out, thus it will impact the availability of other tasks (DoS).

The modified firmware is evaluated and tested with ConFirm on the Samsung Exynos Arndale board. The boot-loader used on the platform is U-Boot, a multi-platform, open-source, universal boot-loader with comprehensive support for loading and managing boot images [23]. Based on the repeatability presented in Table I, we choose 4 events as the monitored ones: architecturally executed instructions (INST), executed branch instructions (BRAN), speculatively executed load instructions (LOAD) and speculatively executed store instructions (STORE). The event return branch is not selected since it does not occur in the monitored subroutine.

[2]We do not include the names of the target devices due to a nondisclosure agreement.

There are seven valid paths in the original *checkTaskSwitch* subroutine, named as P_{ref_1} to P_{ref_7}. During runtime the HPC-based signature will be compared with a subset of the valid signatures (those included in the randomly selected checking window) to check if a matching occurs. The experimental results are shown in Table II (a) for 3 randomly selected check windows.

We observe that the minimum deviation $D(E_x)$ for a malicious path P_{test} to be detected is 8.7% (check window 3). In this case, the monitored LOAD event is implicitly defines the minimum noise detection threshold $N\%$. For example, a detection threshold of 5% is adequate to identify the malicious modifications in every chosen check window. Since there is not any matching between the tested malicious path and any valid path, the test case indicates a successful detection.

2) Case Study 2: Recloser Controller Firmware: The second tested firmware is a recloser controller firmware which runs on a PowerPC-based microprocessor. The embedded system provides protection functionality in the power grid by tripping and reclosing distribution power lines. Specifically, the controller causes a recloser (i.e. circuit breaker) to trip and reclose in case of faults (e.g. short-circuits overcurrents) [24]. In addition, the controller provides information related to restoration operations and functions able to locate the faulted phases, determine the status of a device, check tripping counters etc.

The recloser controller firmware modification includes a Man-in-the-Middle (MitM) attack that performs Ethernet packet sniffing [25]. Specifically, the attack targets the Ethernet packet receiving subroutine, named *tfEtherRecv*, in order to capture the packets of data flowing across the Ethernet network. The modification intercepts the control-flow in the subroutine and copies the received packets to a specific memory location. As a result, an attacker can retrieve the critical information in the received Ethernet packets.

The detection capability of ConFirm on this PowerPC-based firmware is evaluated on the Freescale MPC8308RDB platform with U-Boot as the boot-loader. In similar fashion as in case study 1, we select the events with the smallest C.Vs according to Table I. Because the MPC8308RDB platform has only 4 counters, we select 4 events similar to the ARM-based platform: branch Instructions completed (BRAN), completed Instructions (INST), load micro-ops completed (LOAD) and store micro-ops completed (STORE).

The HPC-based signature of the malicious path exhibits large deviations when compared with the signatures of the five valid paths in *tfEtherRecv*. Similarly with the previous firmware case, the generated signature is compared with the valid signatures included in the checking window to validate any matching occurrence. The results are presented in Table II (b) for 3 randomly chosen check windows.

The lowest deviation $D(E_x)$ in this scenario is equal to 4.2% due to the LOAD event counts of check window 1. Selecting the appropriate threshold $N\%$, for instance 4%, ConFirm differentiates between valid and malicious paths in order to detect the packet sniffing attack.

TABLE II: ConFirm detection capability. The numbers are the event count deviations $D(E_x)(P_{test}, P_{ref\_x})$ (%) of the malicious path from all the valid paths in monitored subroutines *checkTaskSwitch* (a) and *tfEtherRecv* (b) from the ARM- and PowerPC-based firmware respectively. For each path, the bold number indicates the largest deviation among all events. The tested path (malicious) is not matched to any valid path indicating a successful detection.

(a)

Path	Hardware event (E_x)			
	INST	BRAN	LOAD	STORE
Check window 1				
1	27.3	33.3	37.5	**50.0**
2	77.1	71.4	**81.5**	50.0
3	73.3	71.4	**76.2**	60.0
4	51.5	**60.0**	58.3	0.0
5	65.9	50.0	**75.0**	60.0
6	69.8	71.4	**76.2**	33.3
7	62.8	**71.4**	66.7	33.3
Check window 2				
1	77.8	33.3	**150.0**	0.0
2	44.8	60.0	16.7	**66.7**
Check window 3				
1	33.8	**175.0**	31.6	40.0
2	**16.5**	15.8	8.7	12.0

(b)

Path	Hardware event (E_x)			
	INST	BRAN	LOAD	STORE
Check window 1				
1	65.0	**266.7**	78.6	250.0
2	10.0	10.0	0.0	**55.6**
3	41.4	**83.3**	16.7	33.3
4	6.5	22.2	4.2	**47.4**
5	5.7	10.0	7.4	**33.3**
Check window 2				
1	**95.8**	76.8	62.1	30.0
2	19.5	51.1	**70.6**	30.0
3	**65.0**	46.7	37.5	44.0
Check window 3				
1	30.3	12.0	16.7	**47.4**

B. Performance and Storage Overhead

Here we evaluate the performance overhead on the monitored firmware when ConFirm is enabled. The experiment is performed on the presented firmwares and evaluated on the ARM- and PowerPC-based platforms. The runtime performance overhead is mainly due to the execution of extra instructions at each checkpoint: transfer control instructions in the trampoline, HPC handler instructions that read the counters, and ConFirm core instructions that compare the HPC read values with known references. Therefore, the performance overhead tightly depends on the frequency of checks i.e. the size of a check window.

Figure 6 shows the execution time overhead on the monitored firmwares when different check window sizes are applied. For instance, a check window size of 500 instructions

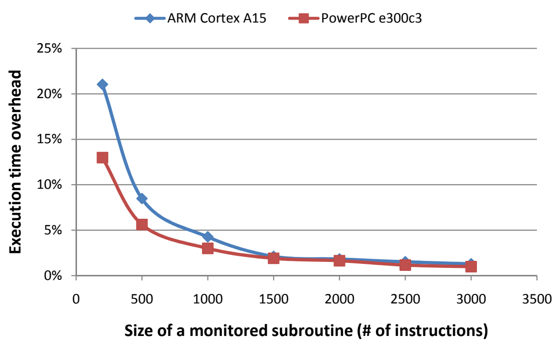

Fig. 6: The execution time overhead of a monitored subroutine with different sizes in terms of number of total instructions when ConFirm is enabled.

leads to an average execution time overhead of 8.48% on the ARM Cortex-A15 platform and 5.62% on the PowerPC e300c3 platform. For the test cases presented in Section IV-A, the performance overhead for the scenario that includes all the subroutine paths is 14.2% and 7.3% for the ARM and PowerPC case respectively.

The storage overhead of ConFirm is mainly for storing the ConFirm components instructions and trampolines. The requirement for storing the unknown valid HPC-based signatures can be fine-tuned according to user's requirements on intrusiveness and security benefits.

Assume a HPC vector in a check window for a valid path that contains the counts of 5 hardware events. Also suppose that 2 bytes are used to store each counted number (i.e. up to 65536 event counts - large enough for the occurrences of any event within a typical check window). In this case, the storage for the signature of a valid path is 10 bytes. If 10 check windows are applied and there are 10 valid paths in each window, then the required storage is 1 KB. The storage size requirement for ConFirm instructions and the trampolines is less than 10 KB. Consequently, the total storage size requirement in this example is around 10 KB. In the extreme scenario where the firmware image size is 1MB then the storage overhead is translated to 1% of the firmware code size.

V. Related Work

The concept of exploiting vulnerabilities in firmware for malicious purposes has been reported in several studies. For instance, it has been shown how to completely reprogram a smart battery by modifying the firmware on it [26]. Cui *et al.* have shown that vulnerabilities in a printer firmware update mechanism can lead to arbitrary malware execution [27]. Furthermore, a recent study in over 693 firmware images uncovered 38 previously unknown vulnerabilities [28]. In the past, several other firmware modification exploits have been presented in a variety of devices (e.g. keyboards [29]).

The attack surface of firmware attacks also includes embedded devices typically used for mission critical systems. Basnight *et al.* have presented how the firmware of a Programmable Logic Controller (PLC) can be modified in order to alter its version [30]. Moreover, it has been presented how to upload an altered firmware over the network enabling arbitrary code execution on Ethernet modules used in Distributed Control System networks [31].

On the detection side, Li *et al.* propose VIPER, a software-only attestation method to verify the integrity of peripherals firmware [32]. However, the authors assume that the OS on the host CPU is trustworthy during verification. Similar work on enhancing the security of I/O devices has also been presented [33]. The proposed framework requires supported System Management Mode (SMM) which is not present in many embedded architectures.

LeMay and Gunter leveraged flash microcontroller units to provide remote kernel attestation assurances in order to audit application firmware integrity [34]. Schellekens *et al.* have studied the integration of a trusted module into a system-on-chip design to achieve an authenticated channel between the trusted module and the external non-volatile memory [35]. Maskiewicz *et al.* have presented a signature verification method to prevent malicious firmware from being installed on a mouse [36]. The verification code though must fit in the bootloader. In their implementation, the verification code requires more space than is currently available in the mouse bootloader, highlighting the multiple constrains in terms of timing, power, area, memory and other resources in the embedded devices domain.

A patented verification method hashes the residing ROM code in order to be verified against a digital signature value that defines an expected value for this hash [37]. This verification mechanism has overhead issues, especially if the main system MCU does not have a multiplier. The idea of protecting software code by guards has also been implemented [38]. The technique inserts a collection of interdependent protection routines into the binary program. Similarly with the hash-mechanisms, guards introduce significant resource cost.

The already implemented mechanisms on firmware verification mainly rely on some form of checksum algorithms. Depending on the embedded system design, checksum algorithms vary from simple error-detection codes such as Cyclic Redundancy Checks (CRC) to cryptographic hash functions. Besides the resource cost and overhead that some checksum algorithms may have, studies have shown that binary modification of the firmware can result in printing out the correct checksum value without even calculating or reverse engineering the checksum routine [39], [40], [27]. Some other existing implementations rely on simple opcode modifications [41], network connectivity (digital signatures) [42], or even require extra hardware [43]. In comparison with the currently developed detection mechanisms, ConFirm, by operating during the booting process of a device, does not have any real-time computing constraints neither requires extra hardware.

VI. Conclusions and Future Work

In this work, we propose ConFirm, a low-cost technique to detect malicious modifications in the firmware of embedded systems by measuring the number of low-level hardware events that occur during execution. In order to count these events, ConFirm leverages the HPCs which readily exist in many embedded processors. We evaluate the detection capability and performance overhead of the proposed technique on various types of firmware running on ARM- and PowerPC-based embedded processors. Experimental results demonstrate its practicality and effectiveness.

Future work on this topic involves enhancing the detection precision of ConFirm. We plan to investigate whether machine learning techniques can decrease the noise threshold, thus strengthening the detection capability of ConFirm, while maintaing a low memory and performance footprint.

Acknowledgment

This work was supported by Consolidated Edison. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Consolidated Edison.

The authors would like to thank Serena Lee, Patricia Robison, Paul Stergiou and Steve Kim from Consolidated Edison, for their great help in improving the presentation of this paper.

References

[1] J. Holler, V. Tsiatsis *et al.*, *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Academic Press, 2014.

[2] IDC Research Company, "Intelligent Systems to Exceed $1 Trillion in 2019 as the Market Continues to Disrupt Traditional Industries Including Manufacturing, Energy, and Transportation," [Online]: http://www.idc.com.

[3] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[4] "OProfile, statistical profiler for Linux systems," [Online]: http://oprofile.sourceforge.net/.

[5] "Hardware performance counters," [Online]: http://en.wikipedia.org/wiki/Hardware_performance_counter.

[6] "Intel 64 and IA-32 Architectures Developer's Manual," [Online]: http://www.intel.com.

[7] "ARM Cortex-A53 Processor Technical Reference Manual," [Online]: http://infocenter.arm.com/.

[8] "4th generation of 32-bit PowerPC microprocessors," [Online]: https://en.wikipedia.org/wiki/PowerPC_G4.

[9] J. Petroni, L. Nick, and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 103–115.

[10] J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *Security Privacy, IEEE*, vol. 2, no. 4, pp. 20–27, July 2004.

[11] U. Erlingsson, "Low-Level Software Security, Attacks and Defenses," in *Foundations of Security Analysis and Design IV*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4677, pp. 92–134.

[12] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM conference on Computer and Communications Security*, 2011, pp. 29–40.

[13] M. Abadi, M. Budiu *et al.*, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security*, 2005, pp. 340–353.

[14] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 577–587.

[15] Wikipedia the free encyclopedia, "Booting," [Online]: http://en.wikipedia.org/wiki/Booting.

[16] G. Hunt and D. Brubacher, "Detours: binary interception of win32 functions," in *Proceedings of the 3rd conference on USENIX Windows NT Symposium*, ser. WINSYM'99, July 1999.

[17] "Arndale Board," [Online]: http://www.arndaleboard.org/wiki/index.php/WiKi.

[18] "MPC8308RDB Reference Platform," [Online]: http://www.freescale.com/.

[19] "Cortex-A15 Technical Reference Manual," [Online]: http://infocenter.arm.com/.

[20] "e300 Power Architecture Core Family Reference Manual," [Online]: http://cache.freescale.com/.

[21] "VxWorks: The Real-Time Operating System for the Internet of Things," [Online]: http://www.windriver.com/products/vxworks/.

[22] V. Durcekova, L. Schwartz, and N. Shahmehri, "Sophisticated denial of service attacks aimed at application layer," in *ELEKTRO, 2012*, May 2012, pp. 55–60.

[23] "U-Boot," [Online]: http://www.stlinux.com/u-boot.

[24] E. Godoy, A. Celaya *et al.*, "Tutorial on Single-Pole Tripping and Reclosing," in *Western Protective Relay Conference*, 2012, pp. 1–21.

[25] A. Ornaghi and M. Valleri, "Man in the middle attacks," [Online]: https://www.blackhat.com/, 2003.

[26] C. Miller, "Battery Firmware Hacking," in *DEF CON 19*, 2011.

[27] A. Cui, M. Costello, and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," in *NDSS*. The Internet Society, 2013.

[28] A. ACostin, J. Zaddach *et al.*, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 95–110.

[29] K. Chen, "Reversing and exploiting an Apple firmware update," 2009.

[30] Z. Basnight, J. Butts *et al.*, "Firmware Modification Attacks on Programmable Logic Controllers," *International Journal of Critical Infrastructure Protection*, 2013.

[31] D. Peck and D. Peterson, "Leveraging ethernet card vulnerabilities in field devices," in *SCADA Security Scientific Symposium*, 2009, pp. 1–19.

[32] 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*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 3–16.

[33] F. Zhang, H. Wang, and otjers, "A framework to secure peripherals at runtime," in *Computer Security - ESORICS 2014*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8712, pp. 219–238.

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

[35] D. Schellekens, P. Tuyls, and B. Preneel, "Embedded Trusted Computing with Authenticated Non-volatile Memory," in *TRUST*, ser. Lecture Notes in Computer Science, vol. 4968. Springer, 2008, pp. 60–74.

[36] J. Maskiewicz, B. Ellis *et al.*, "Mouse trap: Exploiting firmware updates in usb peripherals," in *Proceedings of the 8th USENIX Conference on Offensive Technologies*, ser. WOOT'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 12–12.

[37] D. Morais, J. Lange *et al.*, "Use of hashing in a secure boot loader," 2010, uS Patent 7,676,840.

[38] ARXAN Technologies, "Guarding Technology," [Online]: https://www.arxan.com/why/guarding-technology/.

[39] graeme@lolux.net, "Developing a Trojaned Firmware for Juniper ScreenOS Platforms," *The Circle of Lost Hackers*, vol. 0x0d, no. 0x42, pp. 223–246, 2009.

[40] C. Brunschwiler, "Energy Fraud and Orchestrated Blackouts - Issues with Wireless Metering Protocols (wM-Bus)," 2013.

[41] A. M. J. Garcia, "Firmware Modification Analysis in Programmable Logic Controllers," Ph.D. dissertation, Air Force Institute of Technology, 2014.

[42] L. K. Shade, "Implementing Secure Remote Firmware Updates," 2011.

[43] V. Zimmer and M. Rothman, "Method for performing a trusted firmware/bios update," 2005, uS Patent App. 10/607,367.