cryptography

MDPI

Article

Hardware-Based Run-Time Code Integrity in Embedded Devices

Taimour Wehbe [1,*], Vincent Mooney [1,2] and David Keezer [1]

[1] School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA; mooney@ece.gatech.edu (V.M.); david.keezer@ece.gatech.edu (D.K.)

[2] School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA

* Correspondence: taimour.wehbe@gatech.edu

Received: 1 August 2018; Accepted: 27 August 2018; Published: 30 August 2018

Abstract: Attacks on embedded devices are becoming more and more prevalent, primarily due to the extensively increasing plethora of software vulnerabilities. One of the most dangerous types of these attacks targets application code at run-time. Techniques to detect such attacks typically rely on software due to the ease of implementation and integration. However, these techniques are still vulnerable to the same attacks due to their software nature. In this work, we present a novel hardware-assisted run-time code integrity checking technique where we aim to detect if executable code resident in memory is modified at run-time by an adversary. Specifically, a hardware monitor is designed and attached to the device's main memory system. The monitor creates page-based signatures (hashes) of the code running on the system at compile-time and stores them in a secure database. It then checks for the integrity of the code pages at run-time by regenerating the page-based hashes (with data segments zeroed out) and comparing them to the legitimate hashes. The goal is for any modification to the binary of a user-level or kernel-level process that is resident in memory to cause a comparison failure and lead to a kernel interrupt which allows the affected application to halt safely.

Keywords: embedded systems security; hardware-based malware detection; run-time monitoring; code modification; security threats; Field Programmable Gate Arrays (FPGA)

1. Introduction

The rapid technological advancement that we are featuring in our lives has led to an increasing reliance on technology in our everyday activities. With the emergence of the Internet-of-Things (IoT), we are seeing embedded devices become increasingly interconnected and widespread spanning the range of applications from simple entertainment consumer electronics to complex and safety critical applications such as medical devices, driverless cars and smart power grids. Successful attacks on such critical applications can potentially cause serious damage.

While conventional security measures provide a strong basis for securing embedded systems [1], relying solely on conventional approaches has proven to be ineffective as newer trends have shown that most attacks take advantage of weaknesses present in the implementation of an embedded device. Specifically, a system's security can be compromised through the corruption of binaries as they are being downloaded or stored on the embedded system or through the execution of untrusted or unknown sources. Techniques that perform checking of executable code at compile and load time have been widely spread and proven to be effective [2–4]. However, techniques that verify correct run-time execution are still a major challenge, especially when targeting resource-constrained embedded devices [5–8]. For example, consider an operating system (OS) that is running a user-level application. The system typically starts by loading the application code from disk to memory. Although there have been a plethora of techniques that ensure the integrity of the code while it is present on

disk and right before load-time, verifying the correct execution of that same code while it is resident in memory presents new challenges. Malicious activities, such as malware running on a system, can try to modify the code content at run-time. In fact, these types of attacks have recently become more prevalent. For instance, G. Holmes classified malware in compromised devices into five variants [9], three of which use a run-time infection method to modify and/or insert malicious code where modifications are made to the in-memory copy of the executable code.

In this work, we propose and implement a hardware-assisted technique to detect such malicious activities where we perform binary code analysis and page-based signature (hash) generation of critical applications running on an embedded system. We perform run-time memory monitoring through a separate and isolated hardware monitor that performs on-the-fly page-based hash generation and testing as shown in Figure 1. Malicious modifications to running executable code are rapidly caught and flagged to the OS indicating the presence of abnormal behavior. Further details of our approach and architecture are presented in Sections 5 and 6. In addition, our architecture modification for assessing kernel process integrity on top of user-level process integrity provides a more robust implementation and guarantees that attacks on critical kernel-level modules are detected in real-time (see Sections 5.3 and 6.2).

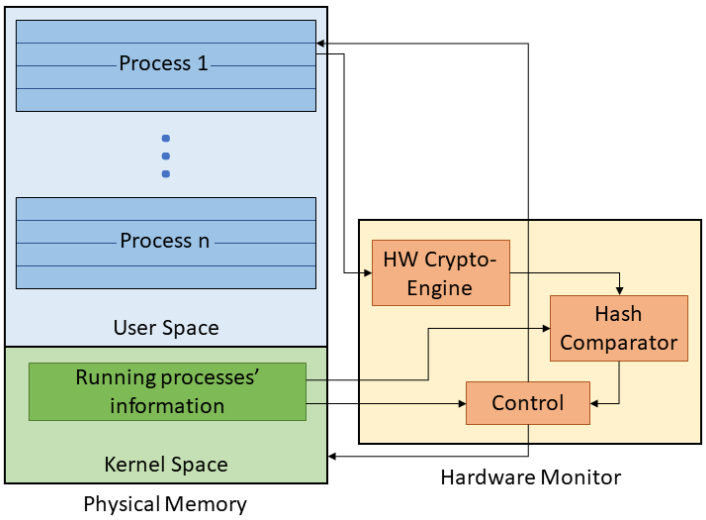

Figure 1. The proposed and implemented process integrity approach showing our hardware monitor tightly coupled to a processor's physical memory.

2. Background

A major portion of attacks on embedded devices is due to the injection of malware especially with the increasing internet connectivity of these devices. Malware attacking such devices can be divided into several categories such as viruses, Trojan horses, spyware, rootkits and other intrusive code [3]. Each of these types of malware performs a specific goal whether it is affecting an application's behavior, leaking sensitive information, spreading network traffic to cause denial of service, or spying on some user's activity.

To better understand and study the weaknesses that these attacks exploit, a division of these vulnerabilities into major classes exists in the Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) standards developed by the MITRE Corporation [10,11]. Some of the major classes involved with embedded systems security are buffer errors, code injection, information leakage, permissions, privileges, access control and resource management. Buffer error vulnerabilities are mainly introduced by allowing code to directly access memory locations outside the bounds of a memory buffer that is being referenced. Code injection weaknesses are usually exploited due to the lack of verification of what constitutes data and control for user-controlled input leading to

the injection of inappropriate code changing the course of execution. Information leakage weaknesses are introduced when the system intentionally or unintentionally discloses information to an actor that is not explicitly authorized to have access to that information through sent data or through data queries. Permissions, privileges, and access control are weaknesses introduced due to improper assignment and enforcement of user or resource permissions, privileges, ownership and access control requirements. Finally, resource management weaknesses are mainly related to improper management and use of system resources, such as making resources available to untrusted parties and improperly releasing or shutting down resources.

3. Prior Work

A wide variety of techniques have been proposed in the literature to detect malicious modifications to processes running on embedded systems. These techniques are broadly divided into anomaly-based detection and signature-based detection methods where the former is involved in detecting abnormal behavior of operation after the detection engine has learned what forms a safe environment, while the latter is involved in looking for known patterns in which an adversary performs the attack on the system [3,12]. Both of these techniques have their own advantages and disadvantages. On the one hand, while the anomaly-based detection techniques help in capturing new (zero-day) attacks, they usually introduce undesirable false positives since a simple deviation from expected behavior could potentially trigger an alarm. On the other hand, signature-based detection techniques provide a good approach to capturing known attacks; however, they are unlikely to detect a new attack if the attack's signature is not present in the signature database.

A majority of these techniques have been mostly or fully implemented in software and are thus vulnerable to software attacks. For example, static analysis software-based techniques try to find possible security vulnerabilities in the code. However, all these techniques are not able to detect or prevent run-time attacks. A survey of common code injection vulnerabilities and software-based countermeasures is presented in [13]. One common weakness among these types of code inspection methods is the infeasibility of discovering all vulnerabilities in a given program by automated static analysis alone [14–17].

Dynamic integrity measurement techniques, such as the ones presented in [5,6,18] have been added as extensions to the Linux Integrity Measurement Unit [2] to detect and prevent return-oriented programming (ROP) attacks. IMUs typically verify the integrity of executable content in an operating system at load-time by inspecting the executable files before loading them; however, the dynamic IMU extensions provide support for run-time detection at the expense of performance overhead. Dynamic software-based techniques usually augment the code by adding some run-time checks so that an attack can be detected. These techniques require either the modification of the target code by adding a new number of executed instructions or the implementation of a separate software monitor in the form of a protected process to keep track of the propagation of data and control during program execution. Therefore, these techniques either require code recompilation or new monitoring code introduction in which both would eventually incur a significant performance overhead [19–21].

Intel SGX and ARM TrustZone have been developed to secure and protect code integrity by isolating user code and allocating private regions of memory [22,23]. These techniques are complimentary to our proposed technique; however, they serve a slightly different goal. Their focus is to separate running applications into secure (trusted) and non-secure (non-trusted) worlds, thus preventing potential attacks on applications running in the secure world, while our work focuses on providing a mechanism to rapidly detect attacks on executable code of any running application. In addition, the implementation of such techniques on resource-constrained embedded systems might turn out to be a challenging task.

Other proposed techniques have relied on a combination of a hardware/software codesign monitoring approach. In these presented approaches, the methods to detect anomalies depend on monitoring the control flow execution of an application [7,8,24,25] or rely on instruction-based

monitoring [26–28]. In the former, static analysis of expected program behavior is extracted and then used by hardware monitors to observe the program's execution trace. Thus, such techniques impose limitations on coding styles and are vulnerable to introducing frequent false positives with any simple deviation from normal behavior, primarily due to the fact that the method relies on having an application follow a predefined control flow without allowing for run-time decision making changes. The latter introduces a significant amount of performance overhead. For example, performing integrity checking on basic blocks as described in [26] results in generating a hash for every set of instructions that fall between two consecutive control transfer instructions. In contrast, our proposed method eliminates this overhead by generating hashes for pages, thus significantly speeding up the integrity checking process.

Therefore, when compared to current state-of-the-art techniques, our work presents the following contributions. Our proposed and implemented architecture provides a novel hardware-based technique to detect malicious code modification at run-time. Our method is able to detect zero-day malware attacks and provides a way of reducing the false positives introduced by other similar techniques by tracking changes in the content of executable pages instead of monitoring the control flow execution of a certain process. In addition, our method relies on securely storing sensitive information (compile-time generated golden hashes) and implementing the monitoring logic in hardware, thus isolating our design from software and the corresponding software attacks.

4. Threat Model

In this work, we primarily focus on threats after deployment. We do not address attacks prior to run-time as there have been a plethora of techniques introduced in the literature to help in protecting compile-time code and allocating secure storage for sensitive information [2,29]. From the different types of common weaknesses and vulnerabilities described in Section 2, we focus on software attacks that end up modifying user process code at run-time. Therefore, attacks that attempt to modify executable memory contents (e.g., via buffer overflow and/or code injection/modification) are primary candidates for the types of vulnerabilities that our work detects. Specifically, a variation of the malware of the evasive type of *hollow process injection* or *process hollowing* constitutes a major focus of our work [30]. Examples of some recent malware that exploited the process hollowing method to inject code into running applications are the BadNews Android malware [4] and Stuxnet [31]. Therefore, our attack model covers the case of a malware that inserts itself into a system and tries to maliciously inject code into another process to end up modifying the process's functionality or leak sensitive information. Attacks of this type are typically performed on the text segment of a process address space and can target code down to instruction-level modification. Therefore, these types of stealthy malware are hard to detect and prevent since they may be hidden anywhere on the system or may be inserted at run-time to target a specific embedded systems application.

In summary, our work addresses malware attacks that maliciously modify the content of a running process's executable pages. Attacks on the processes' page addresses and attacks that end up leaking confidential information without modifying the pages' contents are currently out of the scope of this work.

5. Overall Approach and Method

To protect a process running on an operating system such as Linux, we present an architectural approach composed of a hardware monitor that is tightly coupled with the physical memory of a processor as shown in Figure 1. Our approach involves generating hashes of the monitored process's executable pages at compile-time and storing them in a secure location to be later compared to run-time generated hashes during process execution. Our approach aims to provide a dynamic way of assessing system process integrity while maintaining isolation from software and corresponding software vulnerabilities.

5.1. Detailed Approach

Figure 1 shows a conceptual view of our hardware/software codesign approach implemented at the memory interface with the hardware modules used in our architecture to capture evidence of malicious code execution at run-time. We tightly couple the hardware monitor to the physical memory of a processor to periodically perform memory probing through page-based code analysis. After the compile-time generation of the pages' golden hashes, the kernel informs the hardware monitor once an application is scheduled to run on the processor. The hardware monitor communicates with the kernel to extract the desired running process's information and state. In addition, the monitor is given full hardware access to the physical memory where it is able to read the contents of the process's loaded pages. It is important to note that the hardware monitor only requires read access to the memory, and thus if writes are disabled it is not possible to corrupt the memory through our proposed hardware probing.

5.2. Methodology

To better understand the overall process starting from the preparation of the golden hashes to the verification of code integrity at run-time, we present the flow of our technique in Figure 2. The overall method is divided into two general phases, a compile-time phase and a run-time phase. During the compilation phase and after the target process has been compiled into its equivalent binary, the executable code of the process is partitioned into pages as defined in its Executable and Linkable Format (ELF) file [32]. The content of every page (4 KB) is hashed using a secure cryptographic algorithm to generate a golden signature of the page (in the case of the Secure Hash Algorithm SHA256, the generated hash is a bitstream of 256 bits). Due to the avalanche effect provided by the cryptographic algorithm, any bit change in the page content will result in a significant change in the generated hash of that specific page. The generated hashes are stored and indexed into a database in a secure and trusted location (e.g., a software or hardware root-of-trust [29]).

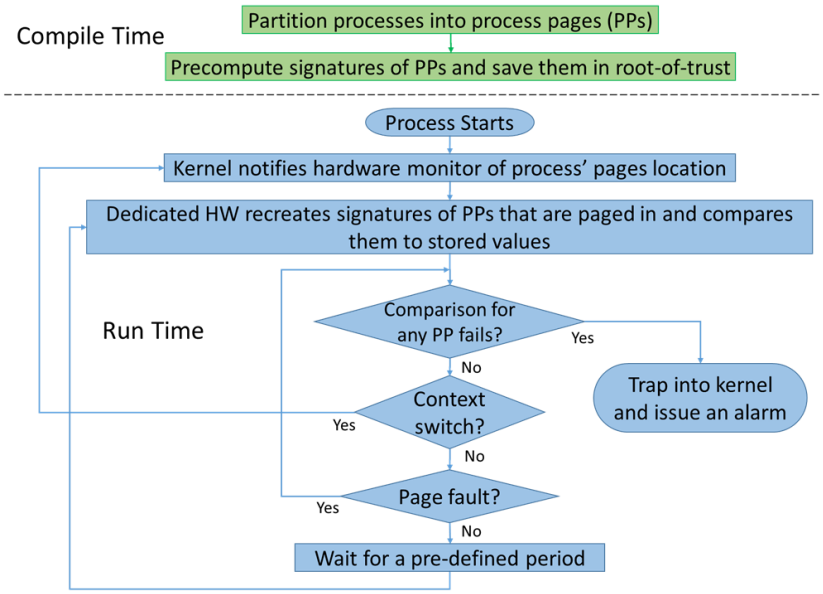

Figure 2. The flow of our proposed approach where golden hashes of process pages are generated at compile-time and then checked during run-time to verify the integrity of the running process.

At run-time, when the process is loaded into memory, the kernel notifies the hardware monitor of the process pages' locations in memory. This is accomplished by inserting a monitoring kernel process that extracts the memory maps of the target process and translates the virtual addresses of the pages to their corresponding physical addresses and page frame numbers (PFNs) before sending them to

the hardware through a communication port set up between the hardware monitor and the kernel. Once the hardware is aware of the locations of the process's executable pages, the monitor grabs the content of every page utilizing a direct memory access (DMA) controller. The content of each page is then passed through a hardware (HW) crypto-engine, and a run-time hash of the page is generated. The newly generated hash is compared to the same page's golden hash as retrieved from secure storage. Currently, the comparison is performed by string matching. However, if faster implementations are required, it is possible to consider techniques that only look for similarities between the golden and regenerated hashes especially since a single bit change in the page contents will result in a significant change in the regenerated hash. The process of regenerating and comparing hashes happens in the hardware monitor and thus is immune to any of the software attack types defined in our threat model (see Section 4).

To allow for continuous monitoring, the hardware monitor's control unit periodically restarts the operation of hash regeneration and comparison once all the process pages have been checked. In addition, to allow for the protection of multiple applications from malicious code modification, the hardware monitor also checks for OS context switches. If the OS starts the execution of any one of the monitored processes, the hardware monitor will similarly grab the code contents of the newly scheduled process, one page at a time, and check for the corresponding pages' integrity. If at any point the comparison between the run-time regenerated hash and the stored golden hash of any page fails, the hardware monitor triggers an alarm by trapping into the kernel to indicate an integrity violation in the running application.

5.3. Extending Our Approach to Assess Kernel Process Integrity

Our proposed approach is scalable and can be expanded to provide assessment of kernel-level process integrity. In that scenario, a dedicated memory has to be attached to the hardware as shown in Figure 3, providing an embedded hardware root of trust. In addition, the dedicated hardware monitor is allowed to interface with and access the kernel address space in memory to read and scan kernel-level processes' pages. Thus, our architecture is now able to check the integrity of specific kernel modules by performing the same technique of hash regeneration and checking at run-time. Of specific interest is our introduced security kernel-level driver which is responsible for interfacing between the kernel and the hardware monitor. In addition, it is imperative to protect kernel-level processes that perform memory management and page mapping/allocation from potential malicious code modification attacks as these kernel-level processes and drivers all play an integral role in our proposed code integrity security model.

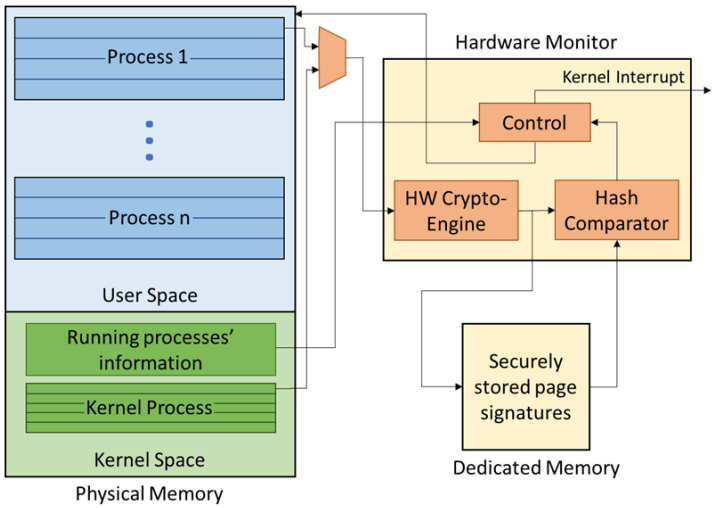

Figure 3. A modified architecture to support kernel process protection.

6. Implementation and Target Architecture

To implement our proposed approach in an embedded system, we target a generic architecture similar to the one shown in Figure 4. Our proposed approach is implemented in custom hardware and interfaces directly with the processor and its main memory. The operating system running on the processor is modified to include a kernel-level driver that interfaces with our custom hardware. In turn, the custom hardware includes components that control the interface with the processor and allow for direct memory access to the main memory and cache. In addition, the hardware is responsible for generating run-time hashes of the process pages, comparing the hashes to the golden references and interrupting the kernel in the case of a hash mismatch. It is important to note that the custom hardware could be an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). Obviously, each of the different architectures provides distinct advantages and disadvantages. For example, implementing our proposed technique on an ASIC chip would provide faster hardware components at the expense of the modularity provided by FPGAs. In particular, having our architecture implemented on an FPGA provides the ability to perform different architectural decisions such as choosing different hashing algorithms during a product's lifetime depending on the needed security of the application.

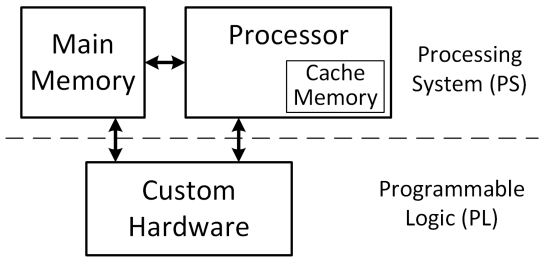

Figure 4. A generic architecture of our proposed approach.

Implementing our approach on a target architecture such as the one shown in Figure 4 with a target ARM processor running Linux presents some challenges. We highlight these challenges along with some of the assumptions we took in our target architecture in the following subsections.

6.1. Assumptions and Challenges

Our architecture aims to maintain system integrity after the system boots. Specifically, our proposed approach is complementary to other techniques that ensure a secure boot process is performed. Thus, we assume that the target processor and operating system boot into a trusted known state before applications start running. In addition, we initially assume that the kernel has varying degrees of protection from attacks, and then we relax these assumptions by expanding our architecture to perform run-time detection of malicious code modifications to kernel processes as well (refer to Sections 5.3 and 6.2). For example, we first assume that the kernel has a high to medium level of assurance of some kernel processes that perform basic tasks such as scheduling, memory allocation, etc. Moreover, we assume that the hardware underneath the operating system is secure and has been provisioned by a system integrator. Finally, our proposed architecture assumes that the kernel can be slightly modified to interact with our hardware module to ensure access to information regarding running processes such as memory maps, compiler, linker and loader information.

To accommodate the implementation of our architecture alongside the Linux operating system, our work is adapted to comply with challenges pertaining to the Linux memory management and virtual memory implementation [32,33]. For example, details of the exact locations of the executable code including shared library code are implemented. In addition, different types of linking code executables are taken into consideration. For instance, our implementation of the proposed security architecture is modified to seamlessly accommodate both static and dynamic linking. Finally, our

technique takes into consideration the implementation of our architecture on embedded systems with Address Space Layout Randomization (ASLR) including caches and their associated coherency protocols [34].

6.1.1. Linux Memory Management and Process Memory Address Space

After a successful compilation of a target application, the generated executable (ELF) file of the application is scanned. The program and section header tables in the ELF file are analyzed to extract the application's executable pages which need to be loaded to memory prior to run-time. The extracted pages are utilized to generate the golden hashes which are then stored in secure memory.

At run-time, to allow for seamless integration between the hardware and the Linux OS kernel, we use specific Linux function calls along with the process information pseudo-file system (`/proc`). For example, the `/proc/pid/maps` pseudo-file is used to extract the virtual addresses of executable pages and code segments of a running process along with the addresses of the shared libraries which the process calls. In addition, the `/proc/pid/pagemap` is used to fetch the physical address of a specific page's virtual address and extract the page's frame number. As mentioned in Section 5.2, the PFN is sent to the hardware monitor to perform on-the-fly hash calculations and comparisons of the monitored processes' pages. Therefore, our architecture supports systems that implement ASLR since virtual-to-physical address translations are performed at run-time.

6.1.2. Unmapped Page Regions

By analyzing the paging process of the Linux OS, we realized that executable code is typically placed in a set of contiguous pages. However, the size of the code in *bytes* is rarely a multiple of the page size (4096 *bytes* in our case). Thus, in most cases, the last page of an executable code would have some unmapped regions. To allow for correct hash generation during compile-time as well as run-time, we devised a technique where the unmapped page regions are masked with a set of binary zero values before being passed through the hash generator as shown in Figure 5. We start by dividing the page to a set of regions according to a predefined granularity. The granularity can be set per application to allow for increased security at the expense of performance. For example, as shown in Figure 5, the granularity is set to 4 *bytes*. Thus, the page is divided into 1024 regions. If, for example, regions 1 and 3 are unmapped, the page is masked such that those regions' binary values are ignored and substituted by zeros prior to being input to the hash generator.

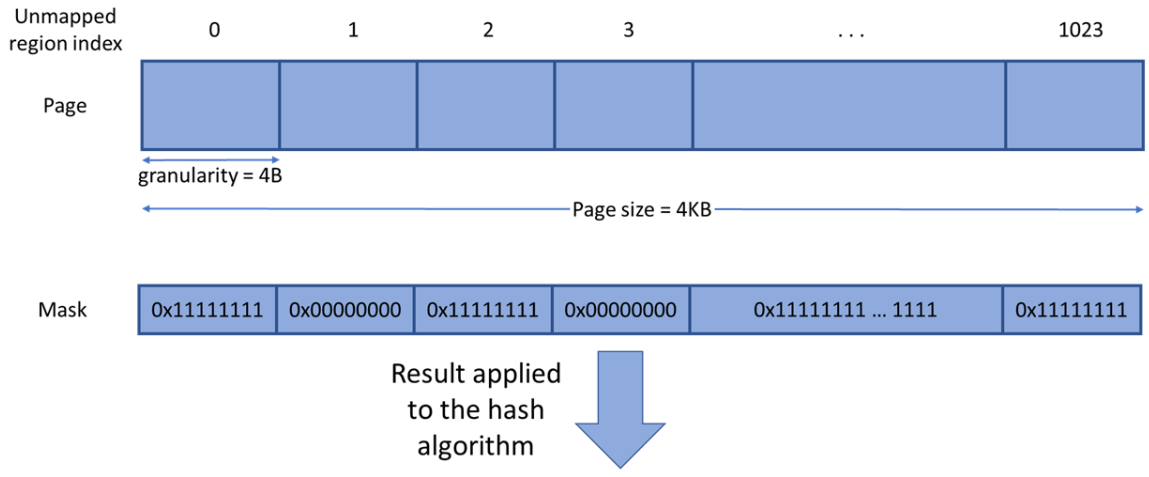

Figure 5. A proposed technique to handle unmapped page regions in an executable binary. The page is divided into equally sized regions where unmapped content is zeroed out before the page is sent to the hash algorithm.

6.1.3. Dynamically-Linked Libraries

To determine the library dependencies inside an application's code, we used specific Linux commands during the compilation phase of the process's binary. For example, the command `objdump` is used to extract the private headers of an ELF file which are then scanned for dynamic library dependencies [32]. The shared object (`.so`) files of the corresponding libraries to be linked are then scanned to extract the libraries' executable page contents. The executable pages are then hashed, one page at a time, and stored in the secure database along with the application's native page hashes. To allow for scalability; furthermore, to reduce any possible performance impact, the database is first checked to see if the shared libraries' page hashes have been previously created by another monitored application before regenerating new hashes. It is important to note that in Linux the actual addresses involved in calling library functions are masked through the indirection provided by the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) in the ELF file [32]. The dynamic loader would use the information provided in these tables to resolve the address of the dynamically linked libraries at run-time. Therefore, the actual code segment of the application remains consistent between compile, load and run-time.

It is worth mentioning that to protect against attacks that try to insert malicious code resulting in new pages that are unaccounted for, our hardware monitor can be configured to trigger an alarm in the presence of an extra number of executable pages as compared to the ones in the original application's code. For example, if the monitored application has 10 golden hashes corresponding to 10 executable pages, the hardware monitor will be expecting to check for the integrity of only these 10 pages at run-time. Thus, if at any instance during the execution of the process, a new executable page is allocated, the hardware monitor will flag an alarm alerting the kernel that the new page does not have any corresponding golden hash. Therefore, our proposed architecture currently limits the support of applications that allow for just-in-time (JIT) compilation and run-time code relocation [35].

6.2. Implementing Kernel-Level Integrity Assessment

When implementing the architectural extension for assessing kernel-level process integrity, we focus on specific critical modules in the kernel. For example, in our current implementation, we monitor the kernel module responsible for the memory management of user-level tasks and applications (`task_mem`). In addition, we monitor the code segment of our kernel-level driver to ensure that the interface between the kernel and our hardware monitor is intact and protected from potential malware. Therefore, at kernel compile-time, we extract the code segments of the modules and drivers that need to be monitored. We then create the hashes of the executable pages of these modules.

At run-time, directly after the secure boot process ends and the kernel fully boots, the hardware monitor starts assessing the code integrity of the critical kernel modules and drivers. The hardware monitor uses a hardware-only accessible register to locate the addresses of the needed critical modules assuming that the kernel is loaded into the same location in memory at boot-time by the bootloader. For systems that have Kernel-ASLR (KASLR) [36] implemented, the hardware is informed of the kernel's loaded address after every system boot. Therefore, in these cases, our architecture is configured to use some Linux pseudo files such as `/proc/kallsyms` to locate addresses of the monitored critical kernel modules directly after the kernel boots and before any applications are run on the system. Next, the hardware monitor starts assessing the integrity of the pages of the critical kernel modules and drivers. Concurrently, the now protected kernel-level driver waits for the monitored user application(s) to start running. Once the operating system schedules one of the monitored processes, the hardware monitor starts assessing the integrity of both the recently scheduled user-level application and the kernel modules and drivers by regenerating the hashes of the monitored user- and kernel-level processes' pages and comparing them to the securely stored corresponding golden hashes. It is worth mentioning that if the kernel monitor triggers an alarm due to a hash violation of one of the kernel's pages, the hardware monitor takes action typically by safely shutting down the system under the assumption that the kernel has been attacked.

7. Experimental Setup

7.1. Experimental Platform

Our experiments were set up targeting the Digilent Zedboard Zynq-7000 ARM/FPGA SoC development board [37]. Thus, the hardware components of our architecture were devised using a combination of VHDL and Verilog code, simulated using Mentor Graphics ModelSim version 10.6a and synthesized using Xilinx Vivado Design Suite version 2017.4 targeting the Xilinx Zynq-7000 FPGA. In addition, as a sample embedded systems application, we developed a heart rate monitor and sample malware targeting an embedded version of Linux (PetaLinux) provided by Xilinx running on top of the dual-core ARM-based Cortex A9 processor on the Zynq-7000 SoC [38]. Our heart monitoring application used electrocardiogram (ECG) data that were captured over 60 s and sampled at a 2 KHz rate from six different individuals [39]. The subjects were healthy and at rest during the capture process. The human subjects measurements process received relevant IRB (Institutional Review Board) approvals.

7.2. Heart Rate Monitor

Heart monitors typically measure a person's heart activity, such as rate and rhythm, and may take the form of a small embedded handheld or portable device. Figure 6 shows an example scenario where an embedded medical system is attached to a treadmill in an exercise facility. The medical device monitors the heart rate activity of an individual while exercising. The person's electrocardiogram (ECG) signals are measured using grip-style dry electrode sensors [40] mounted on the handlebars of the treadmill. The captured ECG data is then used to find the person's heart rate. The calculated heart rate is displayed to the individual in real time. The embedded system is also internet connected to allow for syncing the user's data with the cloud to perform long-term analysis and health diagnosis.

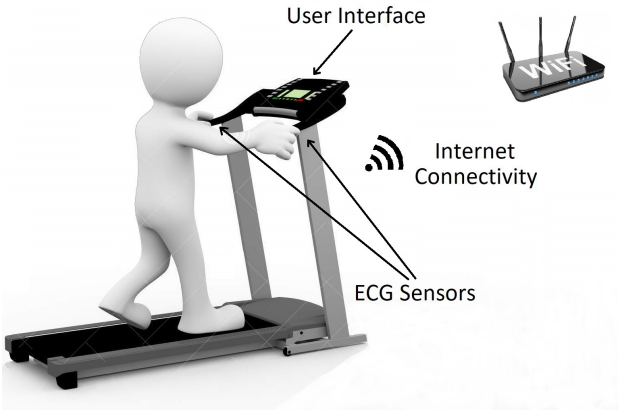

Figure 6. An example scenario of an embedded heart rate monitor attached to a treadmill machine in an exercise facility.

The heart rate monitor used in our experiments can be represented by the block diagram of Figure 7. An ECG sensor is used to capture the user's data which is then amplified and passed through a band-pass filter to improve the data's signal-to-noise ratio (SNR). The captured data is then fed to an analog-to-digital converter (ADC) which quantizes and samples the data at a 2 KHz rate. The filtered and sampled ECG data is then stored in the memory of the embedded device. The system processor runs an application to read the stored data for further processing and analysis. In our experiments, the sample heart rate monitoring application analyzes a user's ECG data to find the heart rate of the individual in real-time. The heart rates are calculated by scanning for consecutive ECG samples and finding the highest value (R-peak) within an ECG signal period. The time difference between two

consecutive ECG R-peaks represents a heart rate value. Figure 8 shows a sample result of the display of the user's heart rate monitor showing a current heart rate of 81.41 bpm.

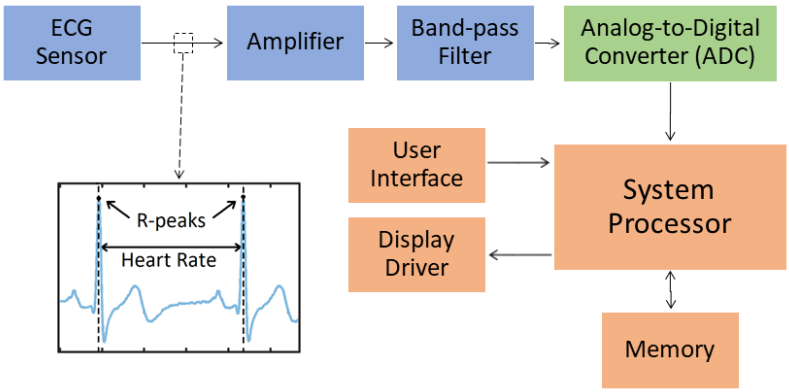

Figure 7. A heart monitor block diagram composed of electrocardiogram (ECG) sensors, amplifiers, filters, analog-to-digital converters (ADCs), a system processor, a memory, a processor interface, a display driver and a user interface.

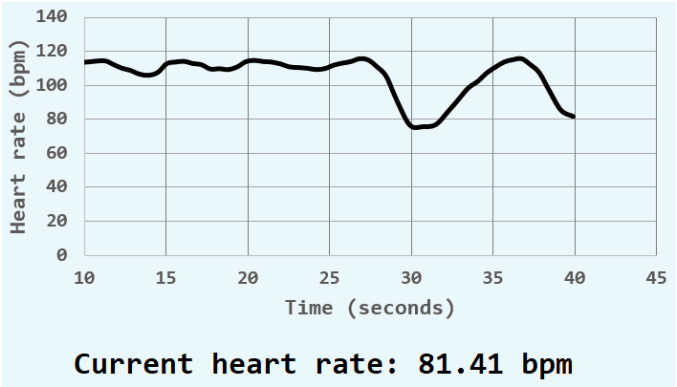

Figure 8. A heart rate monitor displaying a person's current heart rate with a value of 81.41 bpm along with the person's heart rate history over the past 30 s.

7.3. Run-time Memory Corruption Malware

To test our architecture against our target threat model described in Section 4, we developed in-house sample malware targeting the heart rate monitoring application described in Section 7.2. Our malware is assumed to have escalated privileges such that it is able to read and modify other user processes' memory contents. The assumption is that the attacker has enough resources to be able to capture the source and binary code of the target application—heart rate monitor in our case—to devise the attack. The developed malware sample is assumed to have avoided detection so far and attacks the heart rate monitor code at run-time and modifies the code contents of the application by substituting a single instruction. We designed two variations of this malware. Specifically, in the first variation, a subtraction (`rsb`) instruction is replaced with a move (`mov`) instruction, and in the second variation the subtraction instruction is substituted by an addition (`add`) instruction.

Figure 9a shows a code snippet from the original heart rate monitoring application in the C language listing a series of instructions where the heart rate is calculated by subtracting two consecutive values of the ECG R-peak. The heart rate is then displayed to the user. The subtraction instruction is shown in boldface in Figure 9a. Figure 9b shows the equivalent binary code of the compiled and assembled instructions targeting the ARM assembly language. The corresponding assembly

subtraction instruction (`rsb`) is shown in boldface in Figure 9b. Finally, Figure 9c shows the binary code of the attacked application with the first variation of the malware where the `rsb` instruction is substituted by the `mov` instruction. The effect of this change masks the result of the calculation logic of the heart rate monitor leading to the generation of a perfectly normal heart rate value instead of the user's actual heart rate even if the user is having symptoms of a heart failure, thus potentially hiding the need for immediate medical help. The other variation where the `rsb` instruction is substituted by the `add` instruction would make the heart rate calculation display inaccurate values, potentially causing unnecessary efforts by the individual to seek medical help.

```
if (oldDataset > ECGThreshold) {              //R-peak threshold
    peakTime = currentTime-1;                 //current peak time
    heartrate = (peakTime - oldPeakTime);     //calc heart rate
    printBeatDraw(heartBeatsNum++);
    printf("Heart rate: %.2f bpm\n", 60*2000/(double)heartrate);
    oldPeakTime = peakTime;      //set peak time as old peak time
}
```

(**a**)

```
.text:00008768  e51b2038        ldr r2, [fp, #-56]
.text:0000876c  e51b301c        ldr r3, [fp, #-28]
.text:00008770  e0633002        rsb r3, r3, r2
.text:00008774  e50b303c        str r3, [fp, #-60]
.text:00008778  e51b3020        ldr r3, [fp, #-32]
.text:0000877c  e2832001        add r2, r3, #1
```

(**b**)

```
.text:00008768  e51b2038        ldr r2, [fp, #-56]
.text:0000876c  e51b301c        ldr r3, [fp, #-28]
.text:00008770  e300378f        mov r3, #1935
.text:00008774  e50b303c        str r3, [fp, #-60]
.text:00008778  e51b3020        ldr r3, [fp, #-32]
.text:0000877c  e2832001        add r2, r3, #1
```

(**c**)

Figure 9. (**a**) Application code snippet in C. (**b**) Binary and assembly code snippet of the heart monitoring application. (**c**) Binary and assembly code snippet of the attacked application.

8. Hardware Implementation and Experimental Results

Figure 10 shows a diagram of our architecture implementation targeting the Zynq development board. To perform our tests, we generate at compile-time golden hashes of the executable pages (including any required dynamically linked libraries) of the critical kernel modules and drivers along with those of the monitored application (heart rate). The golden hashes are stored in a secure memory (block RAM) which is completely isolated from software (user and kernel space). The hardware monitor uses these hashes to check for the integrity of the user- and kernel-level processes' code at run-time.

To launch our run-time detection mechanism, as soon as the kernel boots, our hardware monitor accesses the pages of the kernel module `task_mem` and starts to periodically assess the pages' integrity. In the meantime, a kernel-level driver is started after the processor boots into a secure state. Once the kernel-level driver starts running, the hardware monitor is informed of the driver pages' addresses, and the integrity of the driver's code is now also continuously assessed. The kernel-level driver sets up the communication interface with the Programmable Logic (PL) in the FPGA and waits on the monitored process to start running. In our current implementation, we only monitor one application (the heart rate monitor); however, monitoring of multiple applications can be seamlessly integrated into our architecture. In fact, to ensure that this integration process can be easily done, we instantiated

multiple runs of the heart rate monitor on different ECG data sets from two different individuals. Once an instance of the heart rate monitoring application is assigned a process id (*pid*), the kernel-level driver begins to continuously monitor the memory mapping of that process and extracts the physical addresses of the statically and dynamically linked executable pages. It then consecutively sends these physical addresses to the hardware monitor.

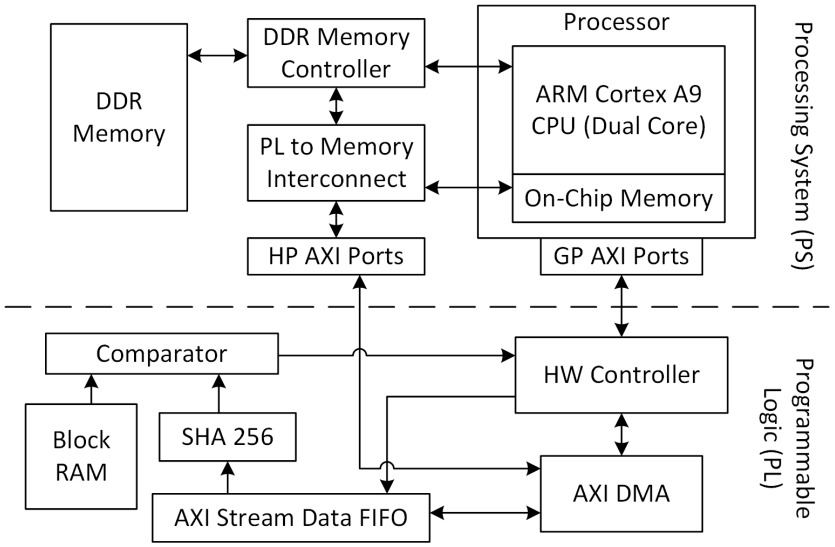

Figure 10. A detailed block diagram of the architectural implementation on the Digilent Zedboard.

The hardware monitor in turn grabs the contents of the process pages from physical memory using the implemented AXI DMA streaming interface shown in Figure 10. Every process page is then fed through a hardware implementation of the SHA 256 algorithm [41], and the generated hash result is compared to the corresponding golden hash stored in the hardware isolated memory (block RAM). Comparison results are then passed back to the kernel in the case of a failure as an interrupt. The kernel-level driver then takes control by halting the affected process. In the case where two instances of the heart rate monitor are simultaneously run, the kernel-level driver would consecutively send the physical addresses of the pages of both processes to the hardware monitor for it to continuously assess the integrity of all the running monitored processes along with the memory management kernel module and the kernel-level driver.

8.1. Performance Analysis

To measure the effectiveness of our implemented architecture, we ran the heart rate monitoring application on a base design architecture excluding any of our proposed security mechanisms and evaluated the performance of the system. The hardware was run at a clock frequency of 70 MHz (the maximum achievable frequency with the current FPGA implementation of SHA256 [41]). The metric involved in our performance evaluation was the time taken by an instance of the heart rate monitoring application to read 120,000 samples of ECG data (i.e., 60 s of ECG data since each second provides 2000 ECG data samples), process them, and continuously calculate and display the heart rate of the individual. This was done on the data sets of all six individuals. The aggregated timing results are shown in Table 1. We similarly re-ran the same heart rate monitoring application having the two malware variations execute and change the calculation of the application as shown in Section 7.3. The malware was randomly triggered using a combination of randomly selected system time and a keyword in keyboard inputs resulting in its execution at different time instances during the application execution. Once triggered, the malware runs for a specific amount of time and then reconfigures the memory back to its normal state trying to mask the damage done. Since in this test we are not running

our proposed security mechanism, the malware was able to fully execute. With the malware running, we again measured the time taken by the heart rate monitoring application to perform the same tasks. The results are reported in Table 1.

Table 1. Performance evaluation results comparing the baseline architecture with two versions of the modified process integrity architecture on the Digilent Zedboard development board.

Architecture	Time to Run (ms)		
	Heart Rate Application	Attacked Application	
		Malware 1	Malware 2
Baseline	60,362	61,855	62,021
Modified for user-level process integrity/Overhead	60,671/0.5%	*halted early*	*halted early*
Modified for user- and kernel-level process integrity/Overhead	60,928/0.94%	*halted early*	*halted early*

Finally, we re-ran the application in the two different cases described earlier; however, this time the architecture was built and modified to introduce our hardware-based monitoring approach for assessing process integrity by modifying the PetaLinux kernel to insert our drivers and configuring the FPGA bitstream to implement our hardware monitor as shown in Figure 10. The performance of two versions of the modified architecture in terms of the aggregated time taken to execute the heart rate monitoring application on data from six different individuals is shown in Table 1. The first modified architecture included monitoring user-level processes only, while the second modified architecture included monitoring both user- and kernel-level processes and drivers. The results show that both versions of our architecture introduce minimal overhead on the performance of the ARM processor, specifically since the actual monitoring is only happening in hardware, and the software (Linux kernel driver) is minimally involved. In fact, the kernel-level driver is only carrying out the process of performing virtual to physical address resolution and sending the addresses to the hardware monitor. Therefore, continuously monitoring the page hashes of the application does not impact the target processor's performance.

Another critical metric that defines the effectiveness of our architecture is the time it takes our hardware monitor to detect the change in memory contents and report that change to the kernel. This was measured on our FPGA platform by starting a timer once the malware was triggered and calculating the time taken by the hardware to trigger an interrupt into the kernel. The aggregated results of this metric for the two versions of our security architecture running on data from the six different individuals are reported in Table 2. This shows that our architecture is capable of detecting a malicious modification (as small as an instruction-level modification) to the heart rate application on average within 250–350 µs if the architecture is monitoring user-level processes only, and within 700–800 µs if the architecture is monitoring both user- and kernel-level processes. In other words, for a malware to be successful and circumvent our architecture (bypass the time-of-check to time-of-use TOCTOU race condition), the malware has to change the executable content of the application, perform its desired task and reconfigure the memory back to its normal state, all within this short time frame of less than a millisecond. Otherwise, our proposed architecture will detect and flag the discrepancy, resulting in a safe halt of the running application since the hardware monitor is continuously scanning all allocated physical pages of the monitored application(s). The periodicity of comparing the same page hash depends on the number of pages present in the application. In our experiments, the hash comparison for all the pages took around 600 µs for the user-level process integrity architecture and 1000 ms for the user- and kernel-level process integrity architecture. These results were validated by the times taken to detect the malware attacks in the worst cases. It is important to mention that the current detection times reported in Table 2 can be dramatically improved if the proposed architecture is implemented on an ASIC chip with dedicated hardware resources as opposed to reconfigurable blocks on an FPGA. In addition, hardware parallelism can be introduced to allow for concurrent hash

computation and simultaneous checking of multiple pages. Moreover, for ease of implementation, we currently implement parts of the DMA controller in software. Ideally, the controller will be fully implemented in hardware achieving faster malware detection. However, in our application scenario, heart rates are typically generated within 0.5–2.5 s and thus no further optimizations to the detection times are needed.

Table 2. Performance evaluation results showing the time taken to detect the malware after it is triggered.

Architecture	Malware Variation	Time to Detect Malware (µs)		
		Best	Worst	Average
User-level process integrity	1	220	543	287
	2	235	601	328
User- and kernel-level process integrity	1	635	985	720
	2	647	996	801

Thus, when comparing our performance results with similar work where code integrity is checked via hardware monitors [26–28], our approach presents a faster detection response with the ability to detect zero-day malware without imposing significant performance degradation on the embedded target processor. As shown in [26], using hashes of basic blocks for checking instruction integrity imposes a substantial overhead. For example, checking for the integrity of all the basic blocks of an application using the technique presented in [26] results in doubling the average clock cycles per instruction (CPI) of a processor. In addition, when comparing our architecture to other software-related approaches, our work provides a higher level of security as our hash generation and checking mechanism is completely isolated from software, and our technique only relies on basic kernel processes that are checked for their integrity using the same proposed method.

8.2. Resource and Power Analysis

To study the impact of our code integrity architecture on resource and power utilization, we implemented both the baseline and the modified design targeting the same Digilent Zedboard Zynq-7000 ARM/FPGA SoC development board. Implementation results reported by the Vivado Design Suite showing area overhead imposed by our hardware architecture are presented in Table 3. In addition, estimates of the power utilization as reported by Vivado are shown in Table 4.

The reported results in Tables 3 and 4 are for the architecture implementing both user- and kernel-level process monitoring. It is important to note that extending the architecture to support kernel-level process monitoring on top of user-level process monitoring did not significantly impact the area results. In fact, the difference between the two architecture versions can primarily be observed in the difference in Block RAM usage. As expected, the architecture implementing both user- and kernel-level process integrity requires more secure memory resources to store the golden hashes. Moreover, as the power utilization results in Table 4 show, the added security components do not incur significant overhead since most of the power is consumed by the Zynq processing system (PS7 in Table 4).

Table 3. Implementation results reported by the Vivado Design Suite showing the hardware overhead imposed by our architecture.

	Zynq-7000 FPGA Resource Utilization					
	LUT	LUT RAM	Flip Flop	Block RAM	IO	BUFG
Utilization	5091	1096	4391	8	8	1
Overall %	9.57	6.3	4.13	5.71	4	3.13

Table 4. Power utilization estimates of the implemented design as reported by the Vivado Design Suite.

Power (mW)								
Clocks	**Signals**	**Logic**	**Block RAM**	**IO**	**PS7**	**Dynamic**	**Device Static**	**Total**
12	15	13	1	6	1532	1579	146	1725

9. Conclusions

In this work, we propose and implement a novel hardware-based run-time code integrity checking architecture to detect malicious modification of application code. Specifically, we generate page-based hashes at run-time and compare them to securely stored golden hashes using a hardware monitor that is tightly coupled to the processor's main and on-chip memory. We also present and implement a method that shows how to further expand our architecture to include assessing kernel-level process integrity. Our experimental results show that our technique introduces minimal resource overhead with negligible performance degradation when implemented on an FPGA/ARM SoC and detects zero-day malware attacks on applications running on an embedded device such as a heart rate monitoring application. The detection occurs in real-time, preventing any potential damage.

In our future work, we plan to address some of the challenges introduced when supporting applications that allow for just-in-time (JIT) compilation and run-time code relocation [35]. One possibility is to include page hash generation both in software as well as in the hardware root-of-trust as an enrollment process for pages with code modifications. The challenge will be to keep this page hash generation process out of the hands of the adversary. We also plan to address cases of dynamic linking that involve object code modification at run-time by allowing for the insertion of new golden hashes at run-time. Some of the possible attack vectors that we also plan to address revolve around finding and exploiting any weakness that our architecture might have. For example, we are currently looking into attacks that take advantage of the unmapped regions in executable pages to insert malicious code modules. One way to address such type of exploitation is by imposing a limitation on the mapping of executable pages where instead of masking the unmapped regions, the kernel would zero out these regions. This would improve the security of our architecture at the expense of a slight performance degradation due to a possible increase in paging overhead. In addition, we are currently looking into ways that not only focus on quick detection but also provide some corrective measures when possible. For example, a fast enough detection could help in preventing corrupted data from being used where the processor can be minimally stalled until the hash comparison is done.

Author Contributions: Conceptualization, T.W., V.M. and D.K.; Investigation, T.W.; Methodology, T.W. and V.M.; Project administration, V.M.; Software, T.W.; Hardware, T.W.; Supervision, V.M. and D.K.; Validation, T.W.; Writing—original draft, T.W.; Writing—review and editing, V.M. and D.K.

Funding: The authors would like to thank the Cisco University Research Program Fund as a part of the Silicon Valley Community Foundation for their generous gift donation CG#1014109 which funded the development and implementation of the proposed ideas in this project.

Acknowledgments: The authors would like to thank Omer Inan and his research group at Georgia Tech for helping in data acquisition and curation.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ravi, S.; Raghunathan, A.; Kocher, P.; Hattangady, S. Security in Embedded Systems: Design Challenges. *ACM Trans. Embed. Comput. Syst.* **2004**, *3*, 461–491. [CrossRef]
2. Loscocco, P.A.; Wilson, P.W.; Pendergrass, J.A.; McDonell, C.D. Linux Kernel Integrity Measurement Using Contextual Inspection. In Proceedings of the ACM Workshop on Scalable Trusted Computing, Alexandria, VA, USA, 2 November 2007; pp. 21–29.

3. Szor, P. *The Art of Computer Virus Research and Defense*; Symantec Press: Upper Saddle River, NJ, USA, 2005.
4. Dunham, K.; Hartman, S.; Quintans, M.; Morales, J.A.; Strazzere, T. *Android Malware and Analysis*, 1st ed.; Auerbach Publications: Boston, MA, USA, 2014.
5. Davi, L.; Sadeghi, A.R.; Winandy, M. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks. In Proceedings of the 2009 ACM workshop on Scalable Trusted Computing, Chicago, IL, USA, 13 November 2009; pp. 49–54.
6. Jaeger, T.; Sailer, R.; Shankar, U. PRIMA: Policy-reduced Integrity Measurement Architecture. In Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies, Lake Tahoe, CA, USA, 7–9 June 2006; pp. 19–28.
7. Arora, D.; Ravi, S.; Raghunathan, A.; Jha, N.K. Architectural Support for Run-time Validation of Program Data Properties. *IEEE Trans. Very Large Scale Integr. Syst.* **2007**, *15*, 546–559. [CrossRef]
8. Sailer, R.; Zhang, X.; Jaeger, T.; van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. In Proceedings of the 13th USENIX Security Symposium, San Diego, CA, USA, 9–13 August 2004; USENIX Association: Berkeley, CA, USA; p. 16.
9. Holmes, G. Evolution of Attacks on Cisco IOS devices. Cisco Blog—Security, Cisco Systems, October 2015. Available online: https://blogs.cisco.com/security/evolution-of-attacks-on-cisco-ios-devices (accessed on 15 November 2017).
10. Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types. MITRE Corp., 2017. Available online: https://cwe.mitre.org/data/published/cwe_v2.11.pdf (accessed on 9 May 2017).
11. Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names. MITRE Corp., 2017. Available online: https://cve.mitre.org/data/downloads/allitems.html (accessed on 17 May 2017).
12. Aycock, J. *Computer Viruses and Malware*; Springer: New York, NY, USA, 2006.
13. Younan, Y.; Joosen, W.; Piessens, F. Code injection in C and C++ : A Survey of Vulnerabilities and Countermeasures. Technical Report CW386, Katholieke Universiteit Leuven, Belgium, July 2004. Available online: http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW386.pdf (accessed on 5 September 2017).
14. Landi, W. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.* **1992**, *1*, 323–337. [CrossRef]
15. Chess, B.; McGraw, G. Static Analysis for Security. *IEEE Secur. Priv.* **2004**, *2*, 76–79. [CrossRef]
16. Abadi, M.; Budiu, M.; Erlingsson, U.; Ligatti, J. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* **2009**, *13*, 4:1–4:40. [CrossRef]
17. Backes, M.; Bugiel, S.; Derr, E. Reliable Third-Party Library Detection in Android and Its Security Applications. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 356–367.
18. Liu, C.; Fan, M.; Feng, Y.; Wang, G. Dynamic Integrity Measurement Model Based on Trusted Computing. In Proceedings of the 2008 International Conference on Computational Intelligence and Security, Suzhou, China, 13–17 December 2008; pp. 281–284.
19. Rinard, M.; Cadar, C.; Dumitran, D.; Roy, D.M.; Leu, T. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In Proceedings of the 20th Annual Computer Security Applications Conference, Tucson, AZ, USA, 6–10 December 2004; pp. 82–90.
20. Yong, S.H.; Horwitz, S. Protecting C Programs from Attacks via Invalid Pointer Dereferences. *ACM SIGSOFT Softw. Eng. Notes* **2003**, *28*, 307–316. [CrossRef]
21. Clause, J.; Li, W.; Orso, A. Dytan: A Generic Dynamic Taint Analysis Framework. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, UK, 9–12 July 2007; pp. 196–206.
22. Anati, I.; Gueron, S.; Johnson, S.; Scarlata, V. Innovative Technology for CPU Based Attestation and Sealing. In Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, 23–24 June 2013.
23. ARM Security Technology—Building a Secure System Using TrustZone Technology. ARM White Paper, April 2009. Available online: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf (accessed on 4 December 2017).
24. Rahmatian, M.; Kooti, H.; Harris, I.G.; Bozorgzadeh, E. Hardware-Assisted Detection of Malicious Software in Embedded Systems. *IEEE Embed. Sys. Lett.* **2012**, *4*, 94–97. [CrossRef]

25. Mao, S.; Wolf, T. Hardware Support for Secure Processing in Embedded Systems. *IEEE Trans. Comput.* **2010**, *59*, 847–854. [CrossRef]
26. Arora, D.; Ravi, S.; Raghunathan, A.; Jha, N.K. Hardware-assisted Run-time Monitoring for Secure Program Execution on Embedded Processors. *IEEE Trans. Very Large Scale Integr. syst.* **2006**, *14*, 1295–1308. [CrossRef]
27. Li, C.; Srinivasan, D.; Reindl, T. Hardware-assisted malware detection for embedded systems in smart grid. In Proceedings of the 2015 IEEE Innovative Smart Grid Technologies—Asia (ISGT ASIA), Bangkok, Thailand, 3–6 November 2015; pp. 1–6.
28. Kanuparthi, A.K.; Karri, R.; Ormazabal, G.; Addepalli, S.K. A high-performance, low-overhead microarchitecture for secure program execution. In Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD), Montreal, QC, Canada, 30 September–3 October 2012; pp. 102–107.
29. Fox, D. Hardware Security Module (HSM). *Datenschutz und Datensicherheit—DuD* **2009**, *33*, 564. [CrossRef]
30. Ligh, M.H.; Case, A.; Levy, J.; Walters, A. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1st ed.; Wiley Publishing: Indianapolis, IN, USA, 2014.
31. Matrosov, A.; Rodionov, E.; Harley, D.; Malcho, J. Stuxnet under the Microscope. Technical Report Revision 1.1. ESET: Antivirus and Internet Security Solutions, 2011. Available online: https://www.esetnod32.ru/company/viruslab/analytics/doc/Stuxnet_Under_the_Microscope.pdf (accessed on 30 March 2018).
32. Mauerer, W. *Professional Linux Kernel Architecture*; Wiley Publshing, Inc.: Indianapolis, IN, USA, 2008.
33. Gorman, M. *Understanding the Linux Virtual Memory Manager*; Prentice Hall: Upper Saddle River, NJ, USA, 2004.
34. Shacham, H.; Page, M.; Pfaff, B.; Goh, E.J.; Modadugu, N.; Boneh, D. On the Effectiveness of Address-space Randomization. In Proceedings of the 11th ACM Conference on Computer and Communications Security, Washington DC, USA, 25–29 October 2004; pp. 298–307.
35. Ansel, J.; Marchenko, P.; Erlingsson, U.; Taylor, E.; Chen, B.; Schuff, D.L.; Sehr, D.; Biffle, C.L.; Yee, B. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. *ACM SIGPLAN Not.* **2011**, *46*, 355–366. [CrossRef]
36. Cook, K. Kernel Address Space Layout Randomization. Linux Security Summit, 2013. Available online: https://outflux.net/slides/2013/lss/kaslr.pdf (accessed on 19 May 2018).
37. ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. Available online: https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board (accessed on 12 July 2017).
38. PetaLinux Tools. Xilinx, Inc. Available online: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html (accessed on 8 August 2017).
39. Wehbe, T.; Mooney, V.J.; Javaid, A.Q.; Inan, O.T. A novel physiological features-assisted architecture for rapidly distinguishing health problems from hardware Trojan attacks and errors in medical devices. In Proceedings of the 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 1–5 May 2017; pp. 106–109.
40. Richard, E.; Chan, A.D.C. Design of a gel-less two-electrode ECG monitor. In Proceedings of the 2010 IEEE International Workshop on Medical Measurements and Applications, Ottawa, ON, Canada, 30 April–1 May 2010; pp. 92–96.
41. Doin, J. SHA 256 Hash Core. Available online: https://opencores.org/project,sha256_hash_core (accessed on 7 September 2017).