creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. DISSERTATION

Efficient Anomalous Behavior Detection on ARM using the Debug Interface

ARM 프로세서의 디버그 인터페이스를 활용한 효율적인 이상 행위 탐지 방법

BY

Yongje Lee

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Efficient Anomalous Behavior Detection on ARM using the Debug Interface

ARM 프로세서의 디버그 인터페이스를 활용한 효율적인 이상 행위 탐지 방법

BY

Yongje Lee

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Efficient Anomalous Behavior Detection on ARM using the Debug Interface

ARM 프로세서의 디버그 인터페이스를 활용한 효율적인 이상 행위 탐지 방법

지도교수 백 윤 흥
이 논문을 공학박사 학위논문으로 제출함

2018년 2월

서울대학교 대학원

전기 컴퓨터 공학부

이 용 제

이용제의 공학박사 학위 논문을 인준함

2018년 2월

위 원 장:	김 태 환	(인)
부위원장:	백 윤 흥	(인)
위 원:	이 혁 재	(인)
위 원:	김 장 우	(인)
위 원:	이 병 영	(인)

Abstract

In recent years, the security and privacy of smart embedded devices become increasingly important problems. Attackers attempt to acquire privileges to control system behaviors at their disposal mostly by exploiting exposed vulnerabilities of a program running on the victim device. As a result, the victim exhibits an abnormal behavior such as control flow diversion. A typical method to detect the anomalous behavior of the currently running program is to monitor the runtime execution flow and check if the monitored flow is legitimate based on a set of pre-defined rules. Therefore in order to detect attacks instantly at the moment when they manipulate the victim device to behave deviantly, a massive amount of CPU execution information representing program behaviors is required. For this reason, we must somehow provide a special mechanism to gather at runtime the CPU execution information and to quickly deliver the gathered information to detection algorithms as the inputs for detection of attacks on the running programs. A lot of researchers have endeavored to address this issue by proposing security solutions that can attain high level of security while minimizing performance overhead introduced to the system. However, we have witnessed that these mechanisms have rarely been accepted to the market. If the mechanism is implemented in software, it obviously will impose too much performance burden on the CPU to be deployed in practice. Even the hardware solutions incur non-negligible modifications to the host architecture internals and thus would substantially increase the design time and manufacturing cost.

This thesis proposes the efficient anomalous behavior detection schemes on smart devices. We choose an ARM processor as our host CPU since ARM has been a dominant player in the mobile CPU market for years. To collect the CPU execution information, we exploit the ARM CoreSight debug interface that has been widely deployed in recent processors for real-time debugging and tracing of software. Using the debug

interface, a hardware-assisted SoC-level mechanisms that are designed to perform the detection task with acceptably low overhead even in performance-constrained devices. In order to show the validity of our approach and explore the implication of using the ARM debug interface for anomalous behavior detection, we first present security monitoring systems that addresses the well-known security issues :data leakage and core-reuse attacks. Then, we present a mixed HW/SW approach that gives users the flexibility to design their own defenses utilizing the ARM debug interface. The experiments also reveal that the area overhead of the hardware is acceptably small when compared to the normal sizes of today's mobile processors.

keywords: Information Security, Hardware-based Anomalous Behavior Detection, Debug Interface, ARM, CoreSight

student number: 2013-30249

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

ARM has developed a great number of processors which have lower power consumption, yet high performance, since it launched its first processor architecture in 1985. Today, ARM processors are undoubtedly deemed as the de-facto standard CPUs for diverse smart mobile devices including smartphones and tablet PCs. With the advent of diverse smart devices in recent years, there is a growing need to protect security and privacy of the data against many different types of software attacks on these devices. The mechanism of attacks may vary, but they all seek to corrupt devices, steal critical data, or to seize control of device platforms. As a result, they usually result in anomalous behavior different from what can be seen in a normal program during execution. The rationale behind this argument is that in the case of attacks, to infiltrate a device, attackers usually gain control over program execution on the victim device by exploiting exposed vulnerabilities, and resultantly divert the execution flow at their disposal, which will eventually cause the victim to behave abnormally.

A typical solution to detect anomalous behavior determines the presence of an attack based on a model that characterizes the runtime patterns of a program. Typically, a model is created by analyzing sequences of important marking events that occur during normal execution. The anomalous behavior detector gets the program execution flow as inputs, and tests if the input belongs to the model. Therefore, in order to protect

the systems and the currently running programs from the attacks, we have to somehow implement a special mechanism to monitor the program's execution flow continuously. Generally, the execution flow can be reconstructed based on the branch behavior such as taken/not-taken of direct branches and the target addresses of the indirect ones. The most straightforward method to acquire the branch behaviors is to instrument the victim code to trace executed branches. Not surprisingly, this software instrumentation is susceptible to substantial performance degradation because for every control transfer monitored during code execution, they must consume CPU cycles to record the branch location.

To tackle this performance problem, much research has attempted to rely on hardware support to efficiently probe the branch traces at runtime [2, 3, 4, 1, 5]. These hardware solutions tend to exhibit high performance by accelerating the tracing process with the assistance of customized hardware logics for the task.

Despite their dramatic performance enhancement, they have a few drawbacks. The first one is that they may require the redesign of the existing processor architecture if one wants to maximize the overall performance while attaining high level of security. For instance, authors in [6, 7, 8] present solutions in which their hardware logics are tightly coupled with the host CPU for close monitoring of every control transfer during code execution. Such a close coupling requires major modifications to processor internal components such as pipeline datapaths or the structure of registers[9, 6, 7, 8], which would stymie the direct deployment of these solutions into commercial platforms. However, they have created basically a new architecture for their CPU (i.e., a variant of the x86 architecture) to implement their solution in hardware. Therefore, their solution requires invasive modifications to the microarchitecture of an existing processor.

In addition, if one chooses to avoid the aforementioned problem and places the anomalous behavior detector outside the host processor as proposed in [10, 11, 12, 13, 14, 15], they may either have their system experiencing non-negligible performance

overhead or have their detector less useful than it should be. Undoubtedly, in their approaches, the host processor can concentrate on the execution of its own code while the time-consuming monitoring task is offloaded to the specialized hardware module outside the processor; in the literature, they empirically demonstrated that their detection scheme can be carried out in a great speed by external hardware, relieving significant burden for the extra computation from the host.

Nevertheless, there still remains an inefficiency. It originates from the limited ability of an external module to watch every internal state change dynamically made by the code running on the host. For precise security monitoring, the external detector should be able to receive from the host various runtime information such as branch targets and context switches. Without such information, the effectiveness of such hardware detectors are substantially reduced, and are allowed to perform only simple tasks such as monitoring memory access patterns as proposed in [13, 11, 12]. To conduct more sophisticated monitoring tasks, researchers in [15, 14] suggest the use of share memory regions to explicitly deliver the essential information from the host to the monitor. Even though their approaches exhibit a substantial performance improvement compared to software-based approaches, there still remains non-negligible performance overhead mainly due to the tremendous amount of traffic for communication. As reported in [15, 14], the overhead is up to 30% of the total execution time even after all their optimizations through hardware communication buffers and special instructions.

In this thesis, proposed is a new anomalous behavior detection scheme that can solve the problems of the previous studies. The proposed anomalous behavior detector can carry out various security tasks with negligible performance overhead on ARM while keeping the host internal architecture intact. To the best of our knowledge, this is the first work that builds a complete SoC prototype to empirically demonstrate the plausibility of an anomalous behavior detection mechanism workable on the ARM-based device. We can build our solution by employing an off-the-shelf ARM processor in an SoC platform. This will bring us another advantage that we can use the same soft-

ware stacks and platforms already developed and established for ARM systems. Our proposed architecture integrates all our hardware supports for accelerating anomalous behavior detection and provides the real-time interface between the CPU and the accelerators. It is widely known that the CPU in a modern smart device is mostly assembled together with other supporting intellectual properties (IPs) in the form of application processors (APs). It is also known that ARM architectures are dominantly deployed as the CPUs in commodity APs today. Therefore in this thesis, our solution is designed as an AP for our smart device, and an ARM processor is opted for as our host CPU that is integrated with the accelerator. The first goal in our design here is how to establish a *hardware-based data transmission channel* that can extract various CPU internal information without incurring performance overhead on the host CPU and send it on the fly to the accelerator located outside the host. To meet this design goal, we exploit an ARM's salient hardware feature, called salient hardware feature, called CoreSight [16], which can perfectly serve our design purpose. Originally for the debug purpose, CoreSight is the debug interface that has a hardware tracing unit that can supply externally a continuous stream of CPU execution traces. Enabling this debug functionality in our AP architecture, we were able to extract from the execution traces all necessary ARM CPU internal information, and delivers it efficiently in hardware to the accelerator. Being supported by the ARM debug interface, our external detector enjoys a full access to the bountiful information transmitted from the ARM CPU in a real time manner, which ultimately enables our solution to quickly detect attack-induced anomalies as soon as each input data set arrives.

This thesis is organized as follows: Chapter 1 introduces and summarizes the thesis. Chapter 2 provides a preliminary research that suggests an idea of exploiting core debug interface. Chapter 3 describes a monitoring solution that detects return-oriented programming attacks uses the control-flow information from the ARM CoreSight debug interface. In Chapter 4, an extended version of CRA detector using CoreSight is explained. In this chapter, we also describe a framework which enables efficient con-

trol flow monitoring in an ARM-based SoC. Finally, Chapter 5 gives the concluding remarks.

Chapter 2

Monitoring Dynamic Information Flow using Control-Flow/Data-Flow Information

2.1 Introduction

DIFT detects a variety of malicious system behaviors that intend to compromise computer systems or leak sensitive information [17]. Generally, DIFT sets up rules to tag (or taint) internal data of interest and keeps track of the taintness of their tags throughout the system [9]. At run time, every data derived from the one with tainted tag has its tag tainted. An alarm will be triggered as soon as any of the tainted data involves in potentially illegal activities, such as pointing inside the prohibited code or being included in a data stream on the output channels. DIFT does not depend on static patterns or signatures of attackers but on their dynamic behaviors at run time. So, it is effective to defend against new attacks whose patterns are not known yet, and to block any unsafe operations on sensitive data even if the data is encrypted [18].

DIFT has been implemented in various forms of either software or hardware. Most software approaches add instrumented code into the original application to track the propagation of tainted data [18, 17]. The key advantage is that they can perform DIFT simply by programming their algorithms. Not surprisingly, however, they show too

large computing overhead to be deployed in practice. Even after much effort [18, 19], the overhead still remains one or two orders of magnitude higher than that of hardware approaches [20, 21] in which extra hardware for DIFT operations is designed and integrated into an existing processor for acceleration. The hardware typically consists of logic blocks that monitor the execution of each instruction in the processor and keep track of tag information flowing from the execution unit at every cycle.

Unfortunately, the remarkable speed of hardware DIFT comes at a cost. To maximize the performance, the hardware has been tightly integrated inside the processor. However, such integration mandates major modifications to processor internal components such as registers and pipeline datapaths, thus substantially increasing the time and cost for re-manufacturing existing processor core architecture [9]. As alternatives to mitigate this problem, there have been more recent studies [9, 14, 15] that propose the techniques aiming to minimize the change to the processor core internal. In their approaches, the host processor can concentrate on the execution of its own code while the time-consuming tag propagation work for DIFT is offloaded to the DIFT hardware device outside the processor. In the literature, they empirically demonstrated that DIFT can be carried out in a great speed by external hardware, relieving significant burden for DIFT computation from the host. However, there still remains a great challenge to overcome for the success of these approaches. The challenge originates from the limited ability of an external device to monitor every internal state change dynamically made by the code running on the host. For precise DIFT, the external monitor should be able to receive from the host virtually all essential runtime information including branch targets, memory addresses and register moves, which will incur a tremendous amount of traffic for communication between the two devices. In [15, 14], they report that the communication overhead may account for up to 30% of the total execution time even after all their optimizations through hardware communication buffers and special instructions. In [9], this overhead issue was treated more aggressively by modifying the host architecture in a way that a customized interface can be embedded into

the processor pipelines. Through this interface, their external device was able to have a special connection for extracting any runtime information for DIFT computation directly from the internal pipelines with very little overhead.

In this chapter, we introduce our recent work on building a hardware DIFT engine. Our approach is similar to those in [15, 14, 9] in that our engine is also connected externally to the host processor. But looking at the details, ours is different from them in several aspects. One main difference is that our approach does not modify internally the host architecture to provide a DIFT-customized interface or connection for the external engine. In our system, the engine is connected to the processor via CDI.

Being plugged into CDI, our DIFT engine has full access to the bountiful information transmitted from CDI. However, as already explained in the previous chapter, the set of CDI signals cannot be simply fed into the DIFT engine, and they must be refined and filtered into what are suitable for DIFT computation. Therefore in our design, between the DIFT engine and CDI, there lies a component, called the *CDI filter*, which, taking the CDI signals as input, filters the signals properly before delivering them to the engine. In Section 2.3, we characterize DIFT computations, and discuss how DIFT works on our computing system with an external engine for DIFT. Then in Section 2.4, we describe in detail the hardware structure of our DIFT engine, and explain how the engine efficiently receives all necessary runtime information through the CDI filter. Experimental results in Section 2.5 show that our engine successfully operates at extremely high speed to provide ample protection against various attacks.

Section 2 : Related Works

2.2 Related Work

Software approaches relied upon either *source-code instrumentation* or *dynamic binary translation* (DBT) [22] to propagate tags for the defense against diverse attacks at execution time [17, 18, 19]. Their main drawback is that they experience excessively

high performance overhead which reaches up to about 40 times the original code execution time in the worst case [17]. Some early hardware approaches [20, 21] tried to improve performance by inserting into the host processor core dedicated hardware modules that accelerate DIFT computations. Although they bring the DIFT computation overhead to around 1% of the original execution time, they usually demand invasive modifications to the processor core. For instance in [20], inside the core, they installed hardware tagging units, called the *flow tracker* and *tag checker*, and widened the widths of registers, internal datapaths and caches, to accommodate tag bits, all of which call for major changes of the processor internal.

In an attempt to minimize the internal architecture changes, the researchers in [14, 15] suggested a DIFT solution in an x86 multi-core environment where one general-purpose core is devoted solely to run a *helper thread* that performs tag propagation for the main code running concurrently on a different core. In [9], they proposed an external DIFT device that processes tags completely outside the host. By dedicating the tag propagation task to a separate core or DIFT hardware, these approaches can manage to enhance the performance drastically. However, as discussed earlier, the fundamental problem of these approaches is that a vast amount of information must be continuously delivered to the external hardware for accurate DIFT operations. To cope with this communication issue, they modified either the x86 architecture to supplement special hardware queues and new instructions [14, 15], or the CPU pipeline datapath to provide a customized channel between the host and the external device [9]. Our work is somewhat similar to the work in [9] since both propose the external hardware optimized for DIFT. But ours is different from theirs in that we exploit the standard interface CDI for communication. Our DIFT hardware has been specially designed to perform tag propagation by interpreting the signals for debugging from CDI.

In fact, the idea of using CDI for other purposes has already been explored in several studies, especially in the field of fault-tolerant computing. Some proposed the systems that can inject faults to the host by accessing internal resources such as regis-

ters and memory via CDI [23]. Others presented an on-line fault detection technique utilizing CDI to retrieve runtime information in a non-intrusive way [24]. The overall concept of these studies exploiting information flow out of CDI without affecting the state or structure of CPU is similar to ours, but none of the above exploit CDI to perform DIFT for ensuring system integrity.

2.3 DIFT Process with an External Hardware Engine

DIFT has been applied to analyze the runtime behaviors of diverse types of attacks, such as *SQL injections*, *buffer overflows* and *data leak prevention* (DLP). In this section, as an example, we explain the DIFT process to guarantee DLP and how it works in our computing framework for DIFT where a DIFT hardware engine is connected to the host processor.

Generally, the first step of DIFT for DLP is *tag initialization* where the input data from confidential sources are tagged as *sensitive*. After tag initialization, follows the *tag propagation* step in which any new data derived from the tagged data is also tagged. Tag propagation continues through code execution. When there is any attempt to extract some data toward outer world, such as sending data over the network or saving it to a storage device, the data is checked whether it is tagged or not. If any tagged data is detected at the *tag check* step, a security exception will be raised. Figure 2.1

Attacker Code for Data Leak	**DIFT for DLP**
1. file_ptr = file_open("Password"); 2. data = read (file_ptr); 3. encrypted_data = encryption (data); 4. data_leak (encrypted_data);	1. tag [file_ptr] = "sensitive" 2. tag [data] = tag [file_ptr]; 3. tag [encrypted_data] = tag [data]; 4. if (tag [encrypted_data] == "sensitive") Exception!!
(a)	(b)

Figure 2.1: Example for DLP using DIFT

presents a code example to illustrate the DIFT process that ensures DLP. Lines 1 and 4 correspond respectively to the tag initialization and tag check stages, and lines 2 and 3 to the tag propagation stage. In our system, the host OS kernel takes responsibility of the first two stages, and our engine of the last one partially because tag propagation is the pivotal and most time-consuming task in DIFT. To denote the tagging in its tag propagation, our engine associates a tag bit with each data location such as registers and memory. When data is tagged, it taints the tag bit by setting the bit on.

We now explain how the attack in the example can be detected by DIFT whether or not the data is encrypted. First, for tag initialization, certain files are to be labeled as sensitive sources. In Figure 2.1, the file *Password* is assumed to be sensitive. In the left column, we see that as the first stage of attack, the adversary code obtains the file pointer after opening the sensitive file, and then reads sensitive data from the file. To detect this trial of attack, the kernel compares a file name to the *list of sensitive files* when the file gets open. To enable this, we have modified system calls for file accesses, such as **open**, so that the kernel can be aware of every access of an application to any file in the system. Since this step requires interaction between the host and the DIFT engine, we have implemented a tag initialization function as a device driver that basically reports to the engine the location (i.e., register number or memory address) of the data that need to be tainted. Then the engine, in return, taints the associated tags for the location in the report delivered from the kernel. In Figure 2.1, the file is sensitive, and so the system call initiates a procedure in which the engine taints the tag of the file pointer by setting the tag bit on.

For tag propagation, any data read from the file is tainted to denote being sensitive because its file pointer tag is on. Even when the data is encrypted, it would be tainted because the outcome of an encrypt function should be tainted if the input is tainted. A tag is propagated in a machine instruction from a source operand to the destination operand based on a set of *tag propagation rules* which are specified at the granularity of basic operations such as arithmetic and logical operations. Figure 2.2 shows a segment

	Original Code	Tag Propagation
1	ldr r9, [r0, #0x40]	tag[r9] = tag[r0] or tag[deref[r0]]
2	add r3, r9, r3	tag[r3] = tag[r9] or tag[r3]
3	orr r9, r9, #0xc0	tag[r9] = tag[r9]
4	sub r2, r9, #0xf	tag[r2] = tag[r9]
5	str r2, [r1]	tag[deref[r1]] = tag[r2]
	(a)	(b)

Figure 2.2: Example of tag propagation rules

of the host code and its associated propagation rules with operands. In the figure, the propagation rule at line 2 in (b) depicts that the **or** operation needs to be performed on the tags of **r9** and **r3** before the result is propagated to the tag of **r3**. In short, two propagation rules, **or** and **=**, are applied when the original code at line 2 is executed. From this example, we learn that for the generation of a tag propagation rule, the DIFT engine must decode the given instruction and identify its opcode and operands.

The tag propagation task on our engine is basically determining whether each tag should be on or off as the host code executes. At the beginning of the execution, the engine allocates one tag bit per CPU register and memory word. Every time the host executes an instruction, the engine also carries out the corresponding propagation rule like those shown in Figure 2.2. Since this rule is generated from each instruction at runtime, the engine first fetches the same instruction from the main memory that the host CPU just did, and tries to resolve the operand values in order to locate every tag operand for its tag operations. However, not all values can be resolved only by decoding the instruction. For instance in Figure 2.2, the load instruction at line 1 uses two operands: register and memory. For correct tag operations, the engine must have the exact register number and memory address. While the former is trivially found (i.e., **r9**) right from the instruction, the latter remains unknown since the value of **r0** is hidden inside the host CPU. In our system, therefore, such hidden information is forced to flow from the host into the DIFT engine in a stream of the data values which

we call *runtime traces*.

At the last stage of attack, the data will be leaked through network. The operations in the right column display a sequence of DIFT actions each corresponding to a statement in the attacker code. At line 4, the data is about to be transferred outside through an output channel. For DLP, the kernel must check the tag of the data given to the channel. For this tag checking step, we installed a function into the system calls involved in data output channels, such as network packet generation. When data is to be carried outside in a network packet through an output channel, this kernel function checks the data tag with the assistance of our DIFT engine. As the first step of this check, the function makes an inquiry to the engine with the location of the data being transferred out. Upon receiving the inquiry, the engine checks the tag value, and notifies the host of the tag checking result. Once the kernel receives the result, it finally checks whether the data is leaked as part of legitimate operations (e.g., bank transactions) or not. If the tainted data is leaked as a result of unauthorized operations, the kernel raises an alarm. Note that deciding the legitimacy of certain operations is in fact beyond the scope of this work as it is irrelevant to the design of our DIFT engine.

2.4 Building a DIFT Engine for CDI

In this section, we describe how our DIFT engine is implemented to fully support the three stages of the DIFT process defined in Section 2.3.

2.4.1 Components of the DIFT Engine

The overall SoC design for our DIFT solution is presented in Figure 2.3 which shows the interconnections between the host CPU core and the DIFT engine. Within our SoC platform, the engine is connected via CDI and a generic shared interconnect to the host CPU along with other hardware modules. It has both the master and slave interfaces so that it cannot only respond to the interconnect transactions from other modules,

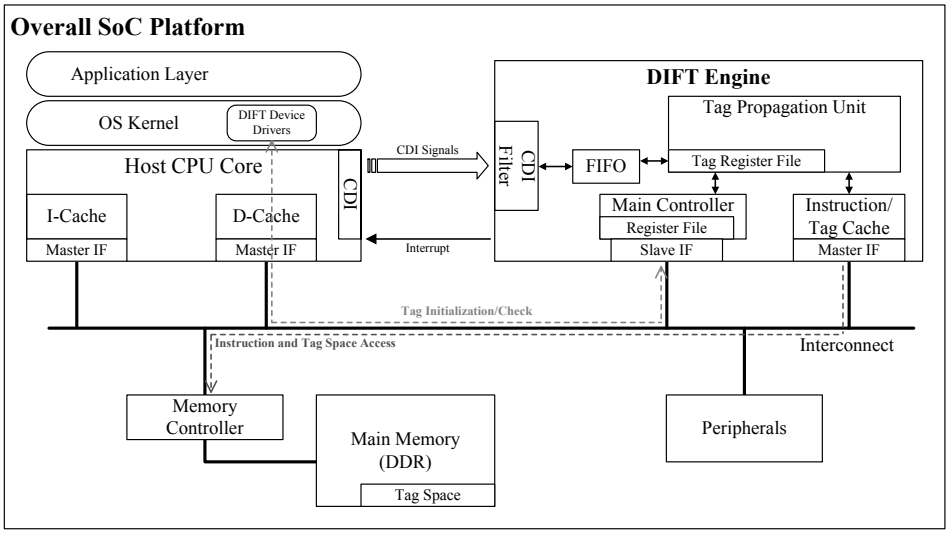


Figure 2.3: Overall SoC platform

but also initiate transactions to access data in the modules. In addition, it can send an interrupt to the host whenever necessary, which would help to reduce the polling overhead incurred during communication between the host CPU and the DIFT engine.

The primary purpose of CDI is to efficiently support the communication for runtime traces flowing from the host to the DIFT engine. A simple way for this without CDI is to use a generic shared interconnect used as the system bus. Of course however, it would consume more bus cycles that normal data transactions could otherwise use. It should also spend extra CPU cycles in executing instructions for the delivery. In this work, to reduce these overheads, we have devised CDI to become a special channel for runtime traces such that a trace can be transmitted from the host into our engine consuming neither CPU nor bus cycles.

As explained earlier, CDI is a CPU side interface built in various modern processors, which helps users verify the functionality and/or analyze the performance of their applications. It is usually connected to the OCD unit that allows the users to watch the control paths that their target processor has taken during code execution, and to examine the values in registers and memory locations. The DIFT engine does not need all

these signals that CDI generates as the runtime traces. Thus, we implemented the CDI filter to drop useless signals before they reach the engine. In our current implementation, the traces emitting from the filter include the current process ID (PID), the address of memory data accessed by a load/store and the program counter (PC) value for the current instruction address. Another important role of the CDI filter is to infer the existence of a branch instruction in the current host execution path. In our implementation, we use a simple heuristic where an abrupt change of the PC values in runtime traces is a sign of the existence. This heuristic is based on a common observation that PC is normally incremented by the instruction word length without (un)conditional branches. As soon as the filter discovers such an abrupt change, it constructs a pair of addresses, the two PC values just before and after the change, each of which stands respectively for the branch address and the target address. These addresses are then delivered to the engine which uses them to grasp the host execution path from outside CPU.

The *main controller* in Figure 2.2 governs the communication between the host and the engine as well as all transactions related to DIFT computation. It is configured by the host to control the DIFT engine. By setting the values of the controller registers, the host can direct the operations of the engine, such as initialization and assignment of the functions for the *tag propagation unit* (TPU). As the central component of our engine, TPU processes all the tags that are associated with data storage in the host. Each entry in the *tag register file* (TRF) represents the tag for an individual register in the host CPU. Borrowing the idea from [25], the engine reserves a special region, called the *tag space*, in memory to stores a long array of bits each of which represents one word of host data in memory. To reduce the memory latency for accessing the tag space, TPU has a small cache, called the *tag cache* [9], for frequently accessed memory tags.

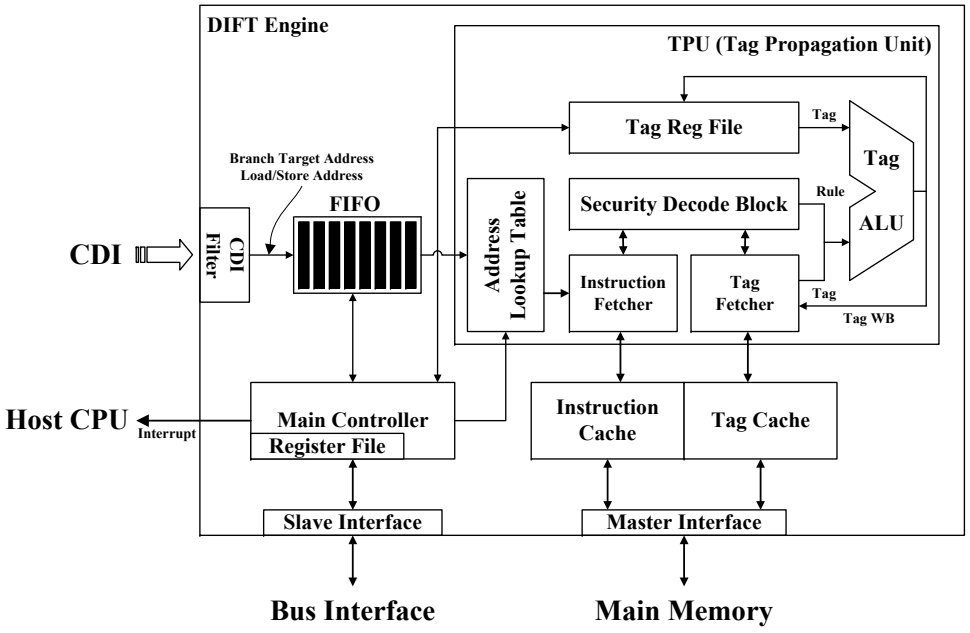


Figure 2.4: Microarchitecture of the proposed DIFT engine

2.4.2 Tag Propagation Unit

Figure 2.4 represents the internal block diagram of our DIFT engine. For unerring tag propagation, it is crucial that TPU correctly fetches host instructions from the main memory, following exactly the same execution path taken by the host CPU. Every fetched instruction enters the *security decode block* (SDB), which derives a tag propagation rule from the opcode and operands of the instruction, as shown in Section 2.3. If the operand is a register, TPU reads from TRF the tag register value corresponding to the operand. If the operand is the memory address for a load or store with register-indirect addressing (see Figure 2.2), TPU acquires the exact address from runtime traces by collecting each trace from the FIFO that temporarily holds all the traces out of the CDI filter. Then, it loads from the tag cache the tag bit representing the memory address. If a tag cache miss is taken place, the *tag fetcher* accesses the tag space allocated in memory to fill the requested line. Once all the tags are ready, TPU performs the tag propagation for the fetched instruction, and writes the result back to the tag bit

for the destination operand in the instruction.

The idea of making TPU follow execution trails of the host brings about a couple of design challenges. One of them is that to follow the trails, TPU relies on the PC values carried in runtime traces, but the values are virtual addresses while TPU uses physical addresses to access the host memory. To resolve the discrepancy in these address spaces, we have the *address lookup table* (ALT) in TPU. An entry of ALT consists of the PID for a process running on the host and the virtual-to-physical address mapping information for the corresponding process. The mapping is determined by the host OS kernel when a new page is allocated for the code section of a process. Therefore, we have slightly modified several system calls related to page allocation in a way that whenever a page is allocated for a process, the mapping information along with its associated PID can be forwarded to TPU for ALT update. Fortunately for our design, a process usually holds only a few entries in ALT. This is because the code section ordinarily occupies a smaller number of pages than the data section. When a process is terminated, its entries are removed from ALT. For this procedure, we have also altered relevant system calls like **exit()**.

Another challenge here is that if TPU should always fetch instructions from memory, it could not catch up with the CPU speed certainly because memory is slow. To tackle this, we have the instruction cache (I-cache) in the DIFT engine. When TPU fetches an instruction, it first tries to load it from I-cache. If a miss occurs, TPU commands the *instruction fetcher* to read the entire cache line containing the instruction from the main memory. As soon as a branch is detected, TPU orders the fetcher to stop and wait until the branch result arrives from the host through runtime traces which continuously carry the PC values. If the branch is taken, the fetcher will load instructions from the address pointed by the new PC.

2.5 Experiment

To evaluate our approach, we have built a full-system FPGA prototype, where the host processor is the SPARC V8 processor, a 32-bit synthesizable core [26] which uses a single-issue, in-order, 7-stage pipeline. It has separate 4K-byte 2-way set associative instruction and data caches. The architecture of our DIFT engine has been implemented as described in Section 2.4. Even though our host core provides their own CDI specification [26], the information that comes out of CDI is quite restricted compared to that of commercial products, such as ARM. Thus, we slightly augmented our core to support the standard CDI signals that resemble those for the ETM of ARM [27]. We implemented the tag cache which is a 512-byte, 2-way set-associative cache with 4-byte cache lines, and the DIFT instruction cache which is a 4K-byte, 2-way set-associative cache with 32-byte cache lines. The bus compliant with AMBA2 AHB protocol [28] is used to interconnect all the modules in our prototype system. Linux 2.6.21.1 is used as our OS kernel and, as mentioned in the previous sections, it has been slightly modified to provide supports for our DIFT engine.

Based on the parameters for the prototype as described above, we synthesized our overall SoC Design onto the prototyping board with a Xilinx XC5VLX330 FPGA and 64MB external SDRAM. Table 2.1 provides the design statistics of our hardware prototype. We quantified the resources necessary for our DIFT engine in terms of lookup tables for logic (LUTs) and block RAMs. The design statistics shows that, compared to the baseline SPARC core, the DIFT engine incurs the resource overhead of 60.0% and 27.98% for BRAMs and LUTs, respectively. To complement the result, we also measured the gate count of the DIFT engine using Synopsys Design Compiler [29]. Synthesized with a commercial 45nm process library, our engine increases 11.98% of overall area over the baseline system. Although it may seem to be a large proportion, the actual gate-count of the engine is 212,051. Considering that recent computing platforms deploy more complex processors like ARM Cortex series compared to the one [26] we used in our experiment, we claim that the area overhead due to our secu-

Category	Component	LUTs	BRAMs
Baseline System	SPARC V8 Core (Host Processor)	4876	18
	Bus components	439	0
	Memory Controller	405	0
	Peripherals (TIMER, UART, and etc.)	963	2
	Total Baseline System	**6683**	**20**
DIFT Engine	Address Lookup Table	670	0
	AHB Master IF	154	0
	CDI Filter	27	0
	FIFO	129	0
	Instruction Cache	293	10
	Instruction/Tag Fetcher	97	0
	Main Controller	176	0
	Security Decode Block (SDB)	35	0
	Tag ALU/Tag Register File	109	0
	Tag Cache	180	2
	Total DIFT Engine	**1870**	**12**
	% DIFT Engine over Baseline System	**27.98%**	**60.00%**

Table 2.1: Synthesis result

rity engine is acceptable in a more realistic hardware. (The gate-count of Cortex-A9 processor with 45 nm process is about 26 M [30].)

To estimate the power consumption of the monitoring engine, we simulated the engine using its synthesized netlists on Modelsim [31]. As a result of the netlist simulation, the switching activity interchange format (SAIF) file is generated. Using the file as an input vector, we run the power estimation tools in Design Compiler with the 45nm process library. The power consumption of the DIFT engine is estimated as 205.4 mW at 1GHz operating clock frequency.

2.5.1 Security Evaluation

To test the security capability of our DIFT engine, we have synthesized the malware that encrypts a sensitive file named as ”secret.txt” and passes it through the network. Our malware, which is similar to the *Dorifel* [32] malware in the wild, has the ability of evading an intrusion detection system or signature-based DLP solution by using the AES encryption algorithm. However, as planned, any attempt to access the file from the malware will be detected by our modified **open** system call in the Linux kernel. When being detected, the kernel invoke the tag initialization function to taint the tag of the file pointer and to configure TPU to be ready for tag propagation. The malware naively proceeds and encrypts the data without knowing the existence of TPU, while our DIFT engine keeps track of the information flow by propagating tags. When the malware tries to leak the derived data, it invokes the **send** system call to transmit a message through the network. Because the system call is also modified to call the tag checking function, the kernel receives tags of the data from TPU as explained in Section 3, and decides whether to allow transfer the data outside or not.

2.5.2 Performance Evaluation

In order to measure the performance of our DIFT solution, we chose eight applications from the *mibench* benchmark suite [33]. The performance of our solution is compared with those of three systems that have different configurations. The first one, called *Native*, stands for a system that executes the original code with DIFT disabled. The *Software-only* solution employs a software-instrumentation technique to augment the host code with instructions that perform DIFT computation on the host. *CDI-DIFT* refers to our DIFT solution that has an external DIFT engine connected to the host side CDI. In addition to these three systems, we added another configuration, named as *Software-DIFT*, that makes use of an external DIFT engine for time consuming tag propagation. The only difference compared to our solution is that the external DIFT engine does not have a connection via CDI to the host. Therefore, for the engine, the

	Native	Software-only	Software-DIFT	CDI-DIFT
dijkstra	1.000	17.785	1.909	1.028
bitcnt	1.000	9.298	1.631	1.011
rinjndael	1.000	47.193	1.799	1.018
sha	1.000	22.556	1.526	1.012
blowfish	1.000	47.147	1.873	1.015
string-search	1.000	17.102	2.247	1.012
patricia	1.000	16.269	1.740	1.016
qsort	1.000	10.503	1.905	1.015
average	1.000	23.482	1.829	1.016

Table 2.2: Comparison table of execution time normalized to Native

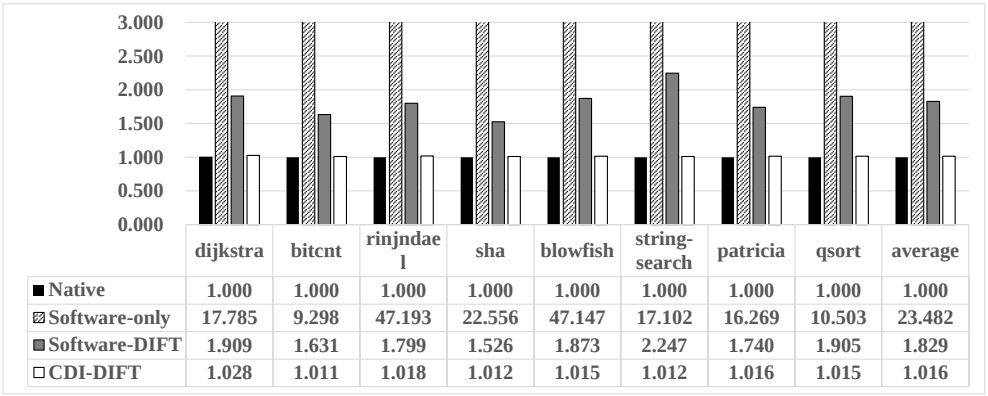
host must execute additional instructions to explicitly transmit runtime traces through the system bus. For this, we instrumented the host code with a set of instructions each of which is inserted after every branch and load/store instruction to send the updated traces to the DIFT engine. We used our in-house tool for code instrumentation.

In Table 2.5, we present the performance comparison of the four configurations. In the table, the host execution time of each configuration is normalized to that of *Native*. The results show that th *Software-only* solution suffers from an excessive performance overhead in that the total runtime is on average 23 times slower than that of *Native*. The overhead of *Software-DIFT* is less devastating than the *Software-only* version: it shows drastically reduced overhead of 82.9% as being compared to that of *Native*. However, it yet runs approximately 1.8 times slower than *Native*. The main cause of such tremendous overhead in both the configurations is the instructions added to the host code for delivering traces. On the other hand, *CDI-DIFT* substantially cuts the overhead down to 1.6% over *Native*. This amazing achievement is mainly due to the fact that, with the supplementary information coming out of CDI, no code instrumentation on the host is needed for our solution. The small amount of performance loss in *CDI-DIFT* is ascribed to the resource competition between the host processor and our

DIFT engine because both are connected to the same interconnect and share the main memory.

2.6 Conclusion

This work presented a dedicated engine for DIFT. Our engine has been implemented and connected to the host processor interface via a standard bus interconnect so that no modifications are made to the processor internal. Nonetheless, being located outside the host system, the engine has limited visibility into the host internal states, which becomes a major stumbling block for successful DIFT computations on the engine. To overcome this limitation, we provide the engine with a separate communication channel through the existing debugging interface, called CDI, of the host. By receiving only the essential information filtered for DIFT out of the original CDI signals, the engine was able to perform its tag propagation task efficiently. Our experiments on FPGA prototype revealed that the engine successfully detects synthetic data leakage attacks with encryption, overcoming the limit of conventional DLP solutions. More importantly, our DIFT engine attains overwhelmingly low overhead, that is less than 2% for a group of mibench applications. The experiments also revealed that the area overhead of the hardware for our DIFT engine is acceptably small even when being compared to the normal sizes of today's mobile processors.



	dijkstra	bitcnt	rinjndael	sha	blowfish	string-search	patricia	qsort	average
Native	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Software-only	17.785	9.298	47.193	22.556	47.147	17.102	16.269	10.503	23.482
Software-DIFT	1.909	1.631	1.799	1.526	1.873	2.247	1.740	1.905	1.829
CDI-DIFT	1.028	1.011	1.018	1.012	1.015	1.012	1.016	1.015	1.016

Figure 2.5: Graph of execution time normalized to Native

Chapter 3

Monitoring Return-Oriented Programming with Control-Flow Information

3.1 Introduction

Since ARM released its first processor architecture in 1985, it has developed a large number of processors which have lower power consumption, yet high performance. Today, ARM processors are undoubtedly deemed as the de-facto standard CPUs for diverse smart mobile devices including smartphones and tablet PCs. As smart mobile devices continue to gain in popularity among the general public for everyday communication and information processing, they are becoming more appealing targets of numerous software-oriented attacks in recent years. The ultimate objective of these attacks is mostly to possess the capabilities which empower them to control the system behavior in almost all aspects so that they can capture various system events and react to the events for their profits.

A popular method to acquire such formidable capabilities has been *code injection*; that is, attackers first inject their own code in the memory and forcefully execute the code after hijacking the normal course of execution [34, 35]. The most effective measure against code injection would be to eradicate the possibility of unau-

thorized code injection and/or injected code execution in the first place. To regulate code injection/execution in the system, modern processors support non-writable and non-executable page permissions with which the OS kernel can enforce the *Writable xor eXecutable* (W$\oplus$X) policy. Although the policy has been proven effective enough to prevent attackers from hijacking control flows of user applications via code injection [36], there has been more recently emerged a new breed of attacks, called *code reuse attacks* (CRAs), in order to neutralize the protection under the policy. Technically, these attacks obey the W$\oplus$X rule since they rely not on injected code but on existing legitimate code in the victim machine. To launch a CRA, the attacker analyzes the target programs and collects a set of code snippets, called *gadgets*, from existing code blocks. By stitching gadgets into a new code sequence, the attacker can perform Turing-complete computation without injecting any additional code. By exploiting common buffer overflows, CRAs can be crafted to target virtually all modern machines like such smart mobile devices as have been targeted by diverse CRA schemes primarily for jailbreaking [37].

With the CRA threat being more significant, the CRA problem has been addressed by various solutions [38, 39, 40, 6, 41, 7], which come in various forms of either software or hardware. The clear advantage of software solutions is that they can be easily adapted to the present machine platform. Their drawback, however, is that they may impose tremendous computational loads upon the host machine mainly because the original program must be augmented with extra code that will be executed periodically to check abnormal control transfers on the host during runtime [38, 39]. Obviously, such a considerable amount of computational overhead can be the biggest obstacle that would impede software solutions from being widely deployed in smart mobile systems that often suffer from severe limited resources. On the other hand, the hardware solutions [6, 7, 41] have demonstrated their strength in performance because they can excel at CRA detection with assistance of special hardware logics customized for this task. To maximize the performance, these solutions all require intrusive modi-

fications to the original CPU internal architecture in a way that their special hardware can be tightly coupled within the host CPU for close monitoring of every control transfer during code execution. Despite their dramatic performance enhancement, fulfilling this requirement however would stymie their direct deployment into existing ARM mobile devices. The reason is that it is contradictory to a common design practice for an ARM device in industry today. In each device lies an *application processor* (AP) [42, 43, 44] as the central computing platform for applications running on the device. To meet ever increasing demands for low design cost, high performance and fast time-to-market, device vendors these days usually build the AP platforms for their products by buying one of ARM cores off the shelf and integrating it together with supporting IPs (intellectual properties) optimized for specific functions. However, if they want to adopt some of those CRA hardware solutions for their products, they cannot follow this usual convention in the hardware design of an AP platform. Instead, they will be compelled to spend their time modifying the ARM core architecture itself including the registers and pipeline datapaths, contrary to the general convention.

Our observation on earlier work inspired us to develop a more practical hardware solution that is to facilitate the acceptance of hardware technologies for the CRA problem in today's smart devices with ARM-based APs. In our solution, all hardware IPs for CRA detection are placed outside the ARM CPU and connected together with this host CPU to build the target AP platform, complying with the conventional design principle. In this paper, we introduce our preliminary architectural design for an ARM-based AP where we have integrated the hardware IPs to detect a representative technique for CRAs, called the *return-oriented programming* (ROP). The ROP attack, as the name implies, chains the gadgets, each of which ends with a return instruction, by manipulating their return addresses on the stack through the exploits of buffer overflow vulnerabilities. Our hardware IPs are basically exerting the same strategy for ROP detection, called the *shadow stack*, that have been employed by earlier work [45, 46]. However, the main difference is of course that our hardware is to monitor ROP attacks

from outside the CPU while theirs were designed to watch the attacks from the inside.

The real challenge here for our approach is how we overcome the limited visibility into the CPU inside so that our monitoring hardware can secure the same quality of information about the host code execution as being readily available to those internal monitors directly from abundant resources within the CPU. To tackle this challenge, we must somehow provide our monitor with a special mechanism that can expose all necessary host code execution information for CRA detection outwardly in a timely manner. Luckily, we have found that ARM is already equipped with a special architecture, called *CoreSight* [16], that corresponds roughly to this mechanism. CoreSight, available in virtually all ARM processors including Cortex-A8, A9 and A15, has been originally developed to supply the outside devices with the information about real-time debugging and tracing of running code in the host. To utilize this architecture, the devices should be attached to the ARM debug interface, such as the *trace port interface unit* (TPIU) and *program trace macrocell* (PTM), from which they can obtain in real time a trace of branch outcomes produced during code execution. In our target AP, for our security purpose, we have attached the ROP monitoring module to the host ARM processor via CoreSight TPIU so that our monitor can see every branch trace of a potential victim program and catch any suspicious call/jump patterns that may indicate CRAs. However, being devoted to its original design purpose, this ARM debug interface carries the minimum information about branch behaviors necessary to keep track of program execution flows for debugging. For instance from CoreSight, we only obtain two kinds of branch outcomes: the target address of an indirect branch and the direction (taken/not-taken) of a direct branch. Sadly for our purpose, these are not sufficient enough; for its accurate monitoring task, our hardware needs to distinguish the differences among various branch types such as direct/indirect calls, returns and other direct/indirect jumps. To supplement this lacking information for our modules, we perform the offline binary analysis for each program and generate the *meta-data* that will direct at runtime the external modules how to obtain the exact type for every

branch in the traces from CoreSight TPIU.

The rest of the paper is organized as follows. Section 4.2 describes the previous studies related to ours and also the threat model with our assumptions. After Section 3.3 presents the overall hardware architecture of our ROP monitoring module, Section 3.4 explains in detail how ROP attacks are efficiently detected with help of additional code analysis in our approach. Then, Section 3.5 discusses the experimental setup and results. For the setup, we have used an ARM-based Zynq FPGA board and prototyped our hardware modules to build a full AP system on the board. The results show that our prototype system offers a feasible security solution for protecting ARM-based APs against ROP attacks with high speed and low area overhead. Finally in Section 3.6, we give some concluding remarks.

3.2 Related Work and Assumptions

In this section, we first relate our work in more detail with others. We then define the threat model assumed in this work.

3.2.1 Related Work

In [45], as one of the hardware solutions for CRA detection, they propose a hardware solution called SmashGuard which protects the host system against ROP attacks. In their approach, on each function call, return addresses are saved in a hardware stack added to the CPU. A return instruction pops the most recent return address from the top of the hardware stack, and then the popped address is compared to the return address of the program at runtime. If there is a mismatch between the two addresses, it is highly likely that the return addresses are maliciously manipulated by attackers. More recently, the branch regulation technique to detect CRAs is introduced in [6] on the ground of a simple invariant ruling the normal behaviors of branches in a programming language. The invariant rule says that the target of a branch instruction should

point to either the address of a function entry or an address within the same function that the instruction belongs to. To enforce this rule, they first rewrite the original binary to annotate each function entry in the victim code with the information delimiting the function boundaries. Then during code execution, their special hardware checks if any branch violates the rule. Being installed within the host CPU pipelines, the augmented hardware was able to efficiently monitor every branch behavior in a timely fashion. SCRAP [7] is another noticeable hardware-assisted technique aiming to detect CRAs based on unique signatures that characterize the execution patterns of instructions commonly encountered when the CPU is under the attacks. As in the case of branch regulation, their hardware is implanted inside the CPU, more specifically at the commit stage of the pipeline. As another related work, the researchers in [41] use their hardware to confine indirect calls and returns only to target the valid addresses, and apply software heuristics to pinpoint CRAs by analyzing their execution patterns. Especially for a function return, they use *active labels* to ensure that the function returns to another function that is currently active, that is, not returned yet after being invoked. These hardware studies for the CRA problem empirically suggest that capitalizing on hardware techniques should be an excellent way to conquer the performance issues, inherent to this problem, from which pure software solutions often suffer significantly. Unlike ours, however, their techniques have difficulty in being directly deployed in most modern smart devices with ARM-based APs, as discussed earlier.

Similar to our effort of extracting accurate runtime information inside the target system, others also have made attempts to use the debug interface like ARM CoreSight. In [47], the On-Chip Debug infrastructures were used to test the fault-tolerance of the target system. Through the interface, they access internal resources including registers and memory so as to inject faults into the resources of interest and analyze the system response in a non-intrusive manner. In [48], researchers proposed an on-line fault detection technology by reusing available debugging features of existing processors like LEON3 and ARM7TDMI. Although all these studies and ours both are com-

monly exploiting the built-in debug interfaces, they use the interfaces for just the fault detection techniques, not the security-enhanced system that we do for.

Very recently, to the best of our knowledge, there is the first approach [49] that utilizes the debug interface in an effort to thwart security threats by developing a kernel integrity monitor. To detect any attempt to compromise the kernel in the target system, the monitor incessantly snoops the memory traffic to track all alterations made to the memory regions for critical kernel data. In the original design, the monitor was to watch the memory bus for all write transactions issued from the host CPU to the main memory so that it could capture malicious transactions towards the kernel regions as soon as they appear on the bus. However in reality, exploiting the memory hierarchy, attackers may deliberately hold their altered data in the on-chip cache before flushing it onto the bus, long enough to hide their attacks from the monitor. Via the debug interface, the monitor came to successfully extract the cache resident information and consequently uncover some of the attacks.

3.2.2 Threat Model and Assumptions

We make the same assumptions on CRA that had appeared in previous studies [40, 38, 7, 46]. We first assume that by enforcing the W$\oplus$X security protection rule [50, 51], the OS and CPU cooperate to forbid a memory page from being both writable and executable at the same time, and subsequently that adversaries cannot execute their injected code. Under this assumption, to disable the defense mechanism and do additional attacks, the adversaries must gain sufficient privileges for the first time. We hereby assume that there are no other attack vectors or security holes which can directly escalate adversary's privilege.

As another assumption, adversaries might exploit memory corruption vulnerabilities like buffer overflows to hijack the control flow of the victim software by overwriting control data in the stack or heap. After they gain the control flow, they have to execute complex operations (e.g., privilege escalation, executing files, and reverse con-

nection) using CRA techniques. Also, we do not consider the adversary who intends other arbitrary attacks such as denial of service. All underlying system operations that are performed by the OS kernel and hardware are always secure until they are thwarted by the sufficient privilege of an adversary obtained through CRAs.

We also assume that an adversary knows all implementation details of the target application and locates the exact address of gadgets. Although *address space layout randomization* (ASLR) [52] can prevent the adversary from locating the address of gadgets, it is here assumed that the adversary still can bypass ASLR and reveal the memory layout by exploiting memory leak bugs like a format string bug [53]. In addition, we assume that the application is not compiled by attackers so that a number of useful gadgets cannot be contained within a small code base. Lastly, the self-modifying code is not considered in our assumptions because it conflicts with the W$\oplus$X security protection.

3.3 Architecture for ROP Detection

As clearly stated in Section 3.1, the ultimate goal of our research is to build a practical hardware solution for CRAs that is deployable to modern ARM-based AP platforms. As the first step towards this goal, we have implemented and integrated monitoring modules for ROP detection as a subsystem, called the *ROP monitor*, in an AP platform with an ARM CPU. Figure 3.1 displays the overall design of our hardware built on top of the platform where as an off-core hardware IP, the monitor is assembled together with the host CPU as well as other IPs. As shown in this figure, we expect that our security subsystem can be readily incorporated into modern mobile products with ARM-based AP platforms. In our platform, the CPU is an ARM Cortex-A9 processor [54] which has been installed in a large number of commercial devices these days [55, 56]. Also, observing the convention of AP design, the host CPU and our hardware are connected with the shared main memory via the standard AMBA3 AXI

interconnect like ARM NIC-301 [57]. It is noteworthy that we have strived to design all our modules to comply with the standard protocols and specifications of commercial ARM-based AP platforms in the present. Specifically, the modules communicate with the ARM CPU, conforming to the AMBA3 bus protocol and the CoreSight debug interface specification. In accordance to our proposed approach, the branch traces should be emitted from the host, and transmitted to the ROP monitor via the ARM CoreSight PTM and TPIU.

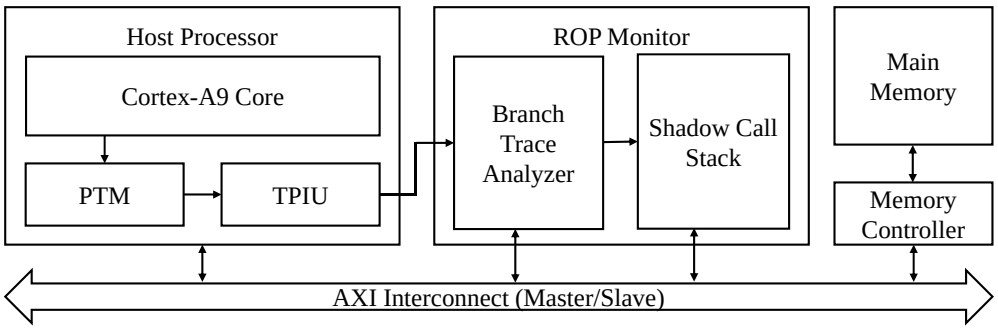


Figure 3.1: Overall architecture of our AP design

As depicted in Figure 3.1, the ROP monitor is largely divided into two modules: the *shadow call stack* (SCS) and *branch trace analyzer* (BTA). In our AP design, SCS plays the pivotal role of monitoring ROP. Upon receiving all branch execution patterns possibly relevant to ROP attacks, SCS analyzes the patterns and judges whether or not the patterns indeed result from the attacks. BTA is an additional module that connects SCS to the TPIU debug interface of the host CPU. Its central role is to decode the debug information from TPIU and to properly refine the information for analysis before the delivery to SCS. Below we will give the detailed descriptions of aforementioned hardware modules of our proposed design.

3.3.1 Branch Trace Analyzer

As discussed before, ARM CoreSight has been used by many developers to debug or evaluate their software on the devices. PTM is the key module of CoreSight that

captures diverse debug information of the ARM CPU, such as branch target addresses, exceptions, current process IDs and instruction set mode changes (ARM/THUMB). It also produces the generic form of the tracing data and compresses the data according to the CoreSight program flow trace architecture specification [58]. After compression, the generated PTM traces are routed to TPIU, and finally forwarded to the off-chip pins to provide the external modules with the runtime information of host programs.

In the current implementation (Figure 3.1), the output signals of TPIU are directly routed to the on-chip ports of the ROP monitor instead of the off-chip pins so that we can utilize the CoreSight modules within our AP prototype without leaking any internal information to outside the AP. We have also built a device driver running on the Linux kernel to control the functionalities of PTM and TPIU. As soon as the CoreSight modules are activated, the PTM traces are generated and transferred to BTA in our ROP monitor via TPIU. In our prototype, PTM is configured to generate a trace for every branch that indicates whether the branch is taken or not. If the branch is indirect[1], then the trace also includes the branch target address.

Figure 3.2 depicts the internal structure of BTA. A main submodule in BTA is the *trace analyzer* that decodes the PTM traces to extract the branch type[2] and target address of each branch instruction executed on the host. The concern here is that the host CPU generally operates faster than other supportive IPs including our monitor, and as a result, the trace analyzer may not process instantly all the traces arriving from the host. To resolve this, we provide an asynchronous memory queue, called the *branch trace FIFO*, in order to temporarily store the incoming traces for the analyzer.

In most cases, the traces alone do not give sufficient information for the trace analyzer to correctly interpret the current branch behaviors in the host CPU, which is an indispensable step to reveal the existence of a ROP attack in our system. For such accurate trace analysis, our SCS in principle needs to know three pieces of the information regarding the behaviors: the branch target addresses and branch types and

[1] that is, an indirect call, indirect jump or return

[2] There are five types: direct jump/call, indirect jump/call and return

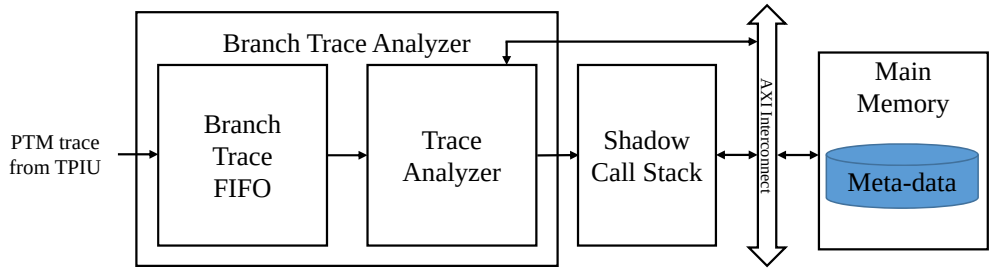


Figure 3.2: Hardware architecture of BTA

branch outcomes (taken/not-taken). Unfortunately, the traces coming through TPIU do not disclose the branch types. To supplement this information, we perform the offline binary analysis before running a program, during which we generate the set of meta-data for the type of every branch and store them in the main memory. Combining the meta-data set with the traces coming from TPIU will constitute the complete information about branch behaviors within the CPU that is necessary for our monitor to analyze the presence of ROP attacks in the system. During code execution, as soon as identifying the type of each instruction, the trace analyzer determines which information should be extracted from the incoming traces. The extracted information is then sequentially fed as a trace of CPU execution into SCS which keeps track of all occurrences of branches in the trace for ROP detection.

3.3.2 Shadow Call Stack

When attackers try to compromise a system with ROP attacks, they manipulate the return addresses stored in the stack in a way to assign them arbitrary values different from those initially set by the callers. In [45, 46], the idea of shadow stacks has been proposed to detect ROP attacks by capitalizing on their distinctive characteristic. Adopting this idea, we have designed SCS to make a copy of the return address on every call instruction and to match the copies with the return addresses coming from BTA.

Figure 3.3 shows the hardware architecture of SCS. Note that SCS have three input signals flowing from BTA; one is `addr_in` for the return address of a branch instruction, and the other two are `call` and `return` for branch types. Using `call` and `return`, the *queue controller* composes the three signals `push`, `pop` and `counter`, and forwards them to the *address queue*, whose job is to maintain a shadow copy of the call stack on the host side. When a function is called, the controller sets `push` on and stores the value of `addr_in` into its queue. Then at the next cycle, it increases the counter value by one. When the function returns, the controller sets `pop` on, outputs the front queue value through the `addr_out_queue` line, and decreases the counter value by one.

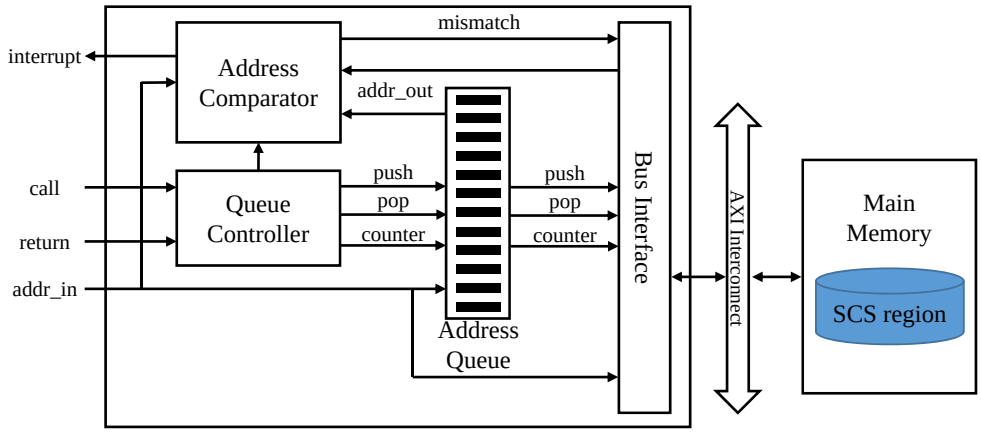


Figure 3.3: Hardware architecture of SCS

Note that the address queue has a finite number of entries, eight in our work. Due to this limited number of its entries, the queue will be overflowed if the victim code contains more than eight times nested calls. To cope with this exceptional case, we reserve a memory region, called the *SCS region*, in the main memory. When the queue is full, its controller saves into the SCS region the values of `addr_in` for all subsequent function calls. Later when one of these functions returns to its caller, the controller fetches the saved return address from the SCS region and outputs it via the `addr_out_mem` line.

Figure 3.3 shows a multiplexer that is connected to two input lines, `addr_out_queue` and `addr_out_mem`, respectively from the different sources for saved return addresses. The multiplexer chooses from which source a return address is transferred to the output line `addr_out`. The choice depends on the value of `counter`. For the value $\leq$ 8, the queue should be the source. Otherwise, the other source will be chosen. The destination of `addr_out` is the *address comparator* that is in charge of making a final decision about the existence of an ROP attack. For this, the comparator takes two input values from `addr_in` and `addr_out`. The former value represents the actual address that the current function is about to return, and the latter the original return address saved when the function was called. If these values do not match, the comparator interrupts the host CPU to notify of an ROP attack. To summarize, in our design, the value of `addr_in` is pushed into either the address queue or SCS region at a call instruction, and compared at a return instruction with the value of `addr_out` from the address queue.

3.4 Meta-data Construction

As discussed in the previous section, the meta-data plays a pivotal role for the success of our ROP detection mechanism built on top of the monitoring hardware platform. In this section, we first describe how the data is generated from a target program, and then the mechanism that makes its use to detect ROP attacks.

Figure 3.4 illustrates the ROP detection process using our ROP monitor. It basically consists of two phases; the static binary analysis and the runtime detection process. In the analysis phase, users generate all the static information of their target programs necessary for ROP detection schemes. For this, they are provided with the *binary analyzer* that is to find the information regarding branch behaviors in the program binaries. As explained earlier, BTA and SCS both are particularly interested in the information related to indirect branch instructions such as returns, calls, indirect

jumps and functions' boundaries. With the help of the analyzer, the users can extract this information from their programs. After the binary analysis, the resulting information is summarized in the form of meta-data whose objective is helping the ROP monitor to better understand the execution behaviors of the target program running on the host and to check if there is any behavior possibly related to ROP attacks. At runtime, when the program binary is loaded by the OS kernel into the host, the associated meta-data for the detection process is also downloaded to the pre-defined region in the main memory. The data can be accessed by BTA to serve its needs. In the following, we will describe the meta-data structure and how the data are used by BTA to detect ROP attacks.

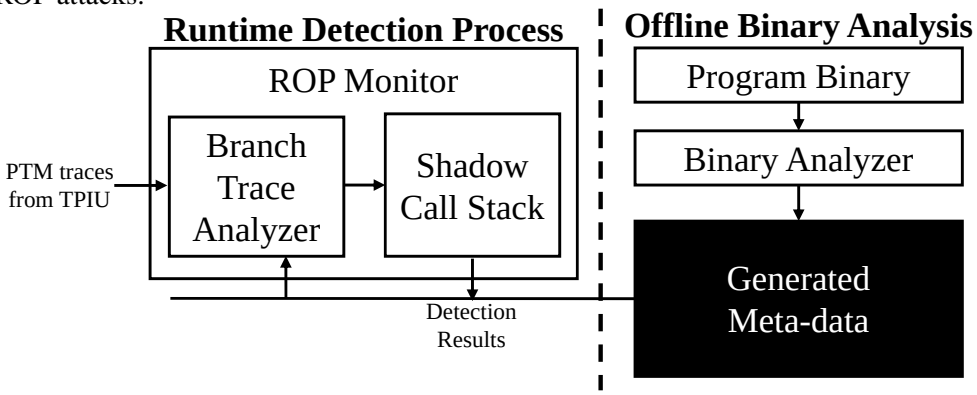


Figure 3.4: ROP detection process

3.4.1 Meta-data Structure

To understand how the ROP monitor works with meta-data, we briefly explain the relevant aspects of the ARM processor architecture. The 32-bit ARM processor is provided with 16 general-purpose registers (`r0-r15`). A key difference between the x86 and ARM architectures is that all ARM registers can be accessed directly by ordinary instructions. Even the program counter (PC) is aliased to `r15`, and thus can be modified by various types of instructions including moves, arithmetics or loads/stores if it is their destination register. As a result, the control flow of a program can be changed

by not only branch/jump instructions but also other types of instructions.

The function calling convention is described in the document for the ARM architecture procedure call standard [59]. According to the document, function calls are implemented by `bl` (branch with link) or `blx` (branch with link and exchange) instructions. Both instructions perform a branch with the link operation that changes PC while the return address is saved to the link register (LR), which is aliased to `r14`.

In the ARM architecture, any instruction that may change PC can be used as a return. The most common method is to use a `bx` (branch exchange) instruction with LR. The instruction `bx lr` replaces PC with the saved return address in LR (`r14`). Another way is to use the `ldm` (load multiple) or `pop` instructions that take PC as the destination operand. In this case, the return address which was pushed on the stack is restored to PC. In the case of indirect branches, the `bx` instruction is executed with the register operand storing the target address.

Following the convention for calls, returns and indirect branches, the binary analyzer extracts the branch type for each branch instruction of the program binary. The extracted branch types are stored in the main memory as meta-data. In addition, to support SCS, BTA should deliver the return and target addresses for branch instructions. Recalling that only the target address of an indirect branch and the direction of a direct branch can be acquired from the traces coming through TPIU, BTA has to generate the target addresses of direct branches and the return addresses for all types of branch instructions. For this purpose, we keep the types and the source and target addresses of branch instructions in the form of meta-data.

As the first step of binary analysis, the analyzer divides the application code into multiple code regions. For this, it scans the entire code from top to bottom. First, it creates a new code region starting from the entry of the application. Now, scanning down the code, it constructs the region by including in order every instruction following the starting point until it hits a control transfer instruction that is either a branch (direct/indirect) or a return. This control transfer instruction will be the last instruction

to be added to the current code region. Upon completing the construction of one code region, the analyzer initializes a new region beginning with the instruction immediately next to the last region, and repeats the above procedure for this region with the instructions in the remaining application code.

In Figure 3.5, we show an example of code regions created by the binary analyzer. Given the resulting code regions, the analyzer generates two types of meta-data for the application code: the *region info set* and *region info jump table*. The former contains the source and target addresses of every control transfer instruction, which is positioned at the end of a code region. The region info jump table is composed of entries each of which maps a code region onto the region info set. By using this table, we can access the information about code regions in the set. The table is located in the main memory with a predefined offset relative to the address where the target application is loaded. The region info jump table has an additional field, *B-type*. The B-type tells the branch type of the last instruction in the current code region.

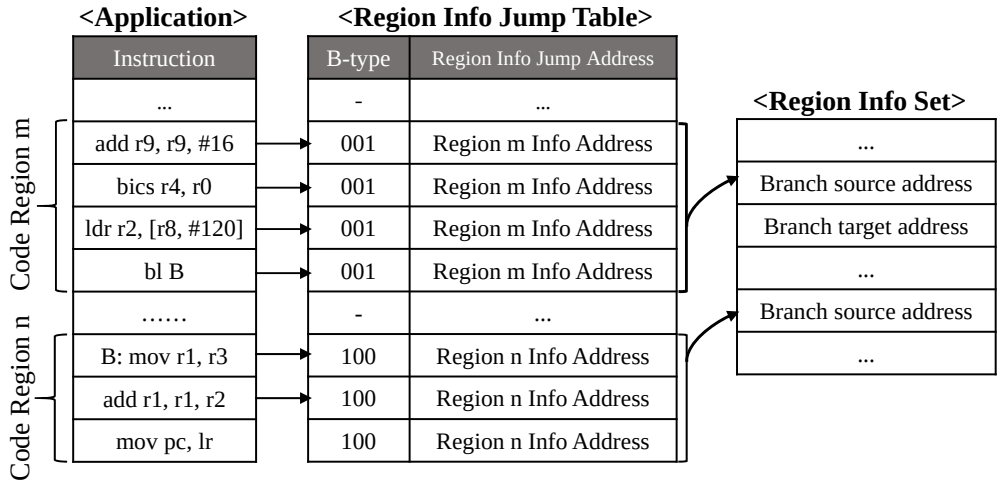


Figure 3.5: Meta-data layout for the ROP monitor

Table 3.1 shows the contents of the region info set where we can see that the set contains different information for different branches depending on their types. For instance, for a direct call instruction, we need both its source and target addresses

in the meta-data. Recall that the traces from the TPIU interface do not carry the target addresses of branch instructions. Therefore, the ROP monitor should obtain the address information from the meta-data. The source address is also necessary to calculate the return address (= source address + 0x4) which then will be delivered to SCS. On the other hand, for an indirect call, the monitor only needs the source address because it can glean the target address from the incoming traces from TPIU. In the following subsection, we will present an example of meta-data and show how each submodule of our ROP monitor obtains the necessary information from the data.

Value	Branch Type	Branch Information
000	direct jump	source/target address
001	direct call	source/target address
010	indirect jump	source address
011	indirect call	source address
100	return	source address

Table 3.1: Information for different branch types

3.4.2 Using Meta-data for ROP Monitoring

An example of meta-data is depicted in Figure 3.6. Let us assume that the control flow has reached the address 0x8040 in the example. The instruction at this address is within the code region 0 whose last instruction is a direct call, "`bl func_3`". As a direct call requires both the source and target addresses, BTA reads these addresses from the address `A` which points to the information of the code region 0 in the region info set. In the example, the return address of the direct call is 0x8048 (= 0x8044 + 0x4). BTA sends the calculated return address to SCS and sets the `call` signal on at the same time. When the `call` signal is on, SCS pushes the address coming from BTA into the shadow stack. The function `func_3` has its return instruction "`mov pc, lr`" at 0x8074. When this instruction is conducted, BTA delivers the target address coming from TPIU, which is the return address, to SCS and simultaneously sets the `return` signal on. When the `return` signal comes to be on, SCS is notified to pop from its shadow stack. Then, the stack top value is compared with the delivered return address

to determine the legitimacy of the return instruction. Let's assume `r2` points to the entry of an arbitrary function such as `func_4` when the control reaches the instruction at 0x8050. Because the instruction is an indirect call, only the branch source address is provided as meta-data for this instruction. But as the return address of a call should always be `source address` + 0x4, integrity checking procedure remains the same as that of a direct call. Following the above procedure, the return address for every call is pushed into the shadow stack to be compared later with the actual return address when the control reaches a succeeding return.

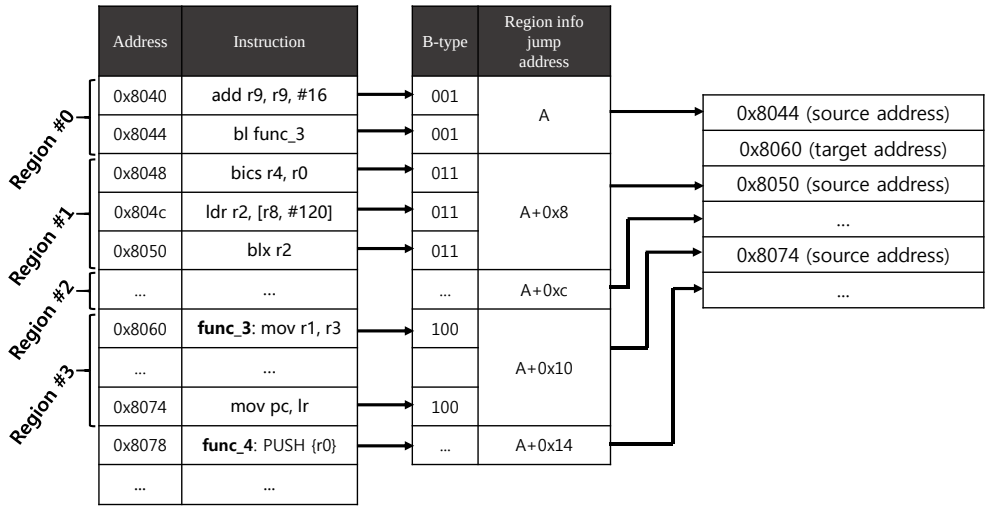


Figure 3.6: Example of meta-data

3.5 Experimental Result

To evaluate our approach, we have implemented a full-system SoC prototype on the Xilinx ZC 702 evaluation board. This development board is composed of the Zynq-7000 XC7Z020 platform which is equipped with a dual-core ARM Cortex-A9 processor, ARM NIC-301 AXI interconnect, an FPGA chip with 1.3 million gates, 1GB DDR3 SDRAM and other peripherals. We have built the host system with the A9 processor and deployed Xilinx ARM Linux kernel version 3.8 as the host OS. Also, two

CoreSight modules, PTM and TPIU, in the Cortex-A9 processor are enabled so that we can extract branch traces from the host CPU. The operating frequencies for our ROP monitor and the host CPU are 90 MHz and 200MHz, respectively. To support the asynchronous relation between the clocks, there is an asynchronous bridge in PTM. The ROP monitor also contains the branch trace FIFO of 16 entries in BTA. Based on the parameters for the prototype mentioned above, we have synthesized the ROP monitor onto the FPGA chip in the evaluation board and quantified the logics necessary for the hardware modules of the ROP monitor including BTA and SCS in terms of lookup tables for logic (LUTs) and memory elements (BRAMs). The synthesis result shows that our ROP monitor occupies 13.8% (7,362/53,200) of total LUTs and 3.1% (539/17,400) of total memory elements. To complement the result, we also measured the gate count of our ROP monitor using Synopsys Design Compiler. With a commercial 45nm process library, the gate-count of the proposed monitor is 86,714. Considering that the ARM Cortex-A9 dual core requires about 26 million gates [30], the area overhead for adopting the ROP monitor into the emerging AP platforms seems to be acceptably small.

To measure the performance overhead of our ROP monitor, we chose ten applications from the mibench benchmark suite [33]. We compared the running time for the applications using two configurations. The first one is *Base* which acts as the control group where the execution of the original code runs on the host processor with the ROP monitor disabled, thus being exposed to ROP attacks. *Ours* refers to the same code execution with the monitor enabled. We show the performance numbers of the two configurations in Figure 3.7 where the execution time of each configuration is normalized to that of *Base*. The results evince that our ROP monitor only incurs 2.39% running time overhead on average over *Base*. This performance overhead is caused by resource conflicts between the host CPU and our monitor because they share the same memory as explained in Section 3.3.

To evaluate the detection capability of our ROP monitor, we have implemented

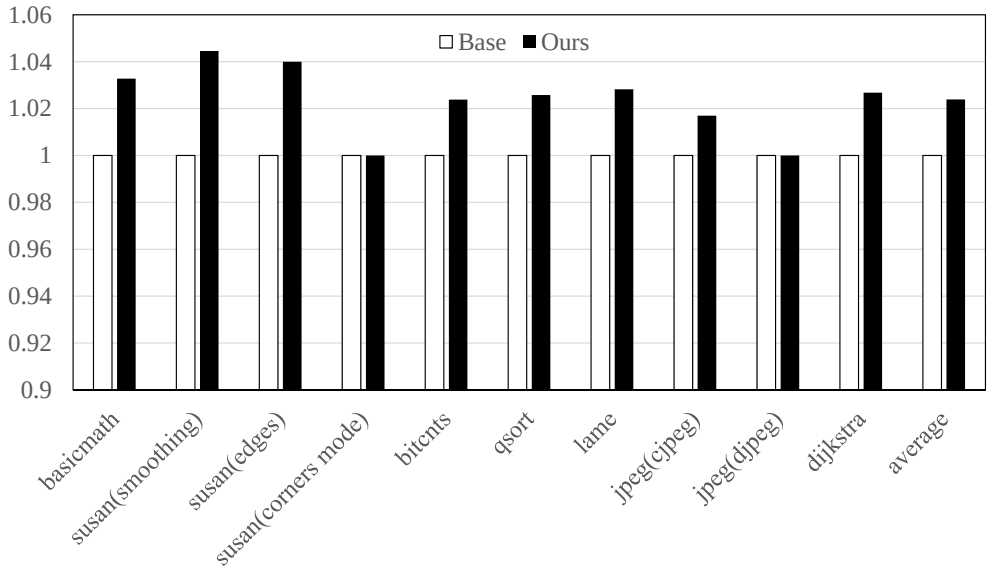


Figure 3.7: Comparison of the execution time normalized to the Base configuration

three types of ROP attacks based on the Shell-storm shellcode [60], as shown in Table 4.2. A1 and A2 are crafted to open a new shell, thus enabling attackers to enter arbitrary commands. A3 changes the attribute of the memory page where the attacker code is located with the `mprotect` system call. Among them, A2 contains a long-gadget which enables the attack to bypass the signature-based CRA defense mechanisms [7, 38, 40], which use short gadget lengths as a distinctive feature of CRAs. To gather necessary gadgets, we have leveraged three general libraries (libwebcore in Android 4.2.2, libc-2.13 in Xilinx-linux and libc-2.15 in Ubuntu) as our code base.

With the implemented attacks, we have tested the effectiveness of our monitor in terms of security. As we expected, all the implemented ROP instances are detected by SCS. Since ROP attacks violate the general convention of the function invocation, their malicious behaviors are always observed by our detection scheme even when the advanced skills like long-gadgets are employed. Based on this result, we assert that our ROP monitor can protect the target system from any type of ROP attacks.

Attack No.	Goal	Advanced Skill	Detection
A1	Open a shell	-	√
A2	Open a shell	Long-gadget	√
A3	Invoke a mprotect system call	-	√

Table 3.2: Description of implemented ROP attacks and detection results of the attacks

3.6 Conclusion and Future Extension

This paper introduces a hardware-based ROP monitor to detect code reuse attacks on commodity smart mobile devices. The monitor provides a negligible performance overhead for runtime detection of ROP attacks. Moreover, the monitor has been implemented without any modifications in the processor internal, and the hardware modules are integrated with a widely available processor core, observing the conventional AP design rules so that our solution can be easily implanted to commercial mobile APs. Our experiments on the FPGA prototype revealed that our current implementation successfully detects synthetic ROP attacks. The experimental results also reveal that the area overhead of the hardware for our monitor is acceptably small even when being compared to the normal sizes of today's mobile processors. All in all, we hope that our proposed architecture would become an attractive CRA defense solution to production-quality ARM-based mobile AP platforms.

As an extension to our implementation, we are now planning to augment our hardware with additional modules for detecting CRAs with a more advanced attack scheme, called the *jump-oriented programming* (JOP). For JOP, instead of simply altering return values saved in the stack, attackers manipulate register operands of indirect branches to retarget the branch destinations to their gadgets. Therefore, JOP attacks would be able to bypass our ROP monitor, which is designed only to watch changes to the return addresses. To detect these attacks, our monitor should have a capability to keep track of all the behaviors of indirect branches. To add this capability to our hard-

ware, we will implement in the monitor a new module that can check the legitimacy of indirect branch behaviors to find whether or not each branch is maliciously exploited. Fortunately, we have found that many of existing submodules like BTA in our existing monitoring module for ROP can be reused as they are. The original meta-data layout for ROP can also be reused but with minor modification. Consequently, we believe that the current design of our monitor architecture can be incrementally extended to detect various types of CRAs, even those that may appear in the future, from outside the host CPU without modifying the CPU architecture itself.

Chapter 4

Implementing Host-based Control-Flow Monitoring Framework using the ARM PTM Interface

4.1 INTRODUCTION

ARM processors have been the de-facto standard central computing engines in mobile SoCs for various embedded systems. As the architecture of ARM processors are getting sophisticated, so does the software running on the processors. This complication urged architects to design advanced debugging features that can help developers see the way how software interacts with others or with the underlying hardware. As an answer to this desire, ARM has deployed in recent ARM processors the on-chip real-time debugging resources, called *CoreSight* [16]. Among the hardware IPs defined in CoreSight, *Program Trace Macrocell* (PTM) is the core module that provides the on-chip hooks to implement an on-chip instruction trace facility. PTM supports full-speed tracing for the instruction trace by monitoring PC changes and branch addresses. Having rich information, the traces emitted from PTM can be used for different tasks other than debugging. One such an example is to monitor the execution flow (or control flow) of the program running on the host from the outside the host CPU using a hardware IP. This motivated us to develop an external security monitor that can detect the security

breach caused by the well-known *code reuse attack* (CRA), which performs malicious activity by executing chains of small code sequences (called *gadgets*), normally collected from the existing benign code blocks in the system memory.

As the CRA threats continue to escalate, a variety of *control flow integrity* (CFI) techniques based on the original work of Abadi et al. [61] are regarded as the most promising measures against CRAs. These solutions have come in various forms of either software or hardware. The clear advantage of software solutions is that they can be easily adapted to an existing machine platform. However, a drawback is that they may impose tremendous computational loads on a host machine. This is primarily because the original program must be augmented with additional code that will check abnormal control transfers on the host during runtime [38, 39]. Obviously, such considerable computational overhead can be the greatest obstacle that prevents software solutions from being widely deployed in commercial computing devices, particularly those in embedded and mobile systems with extremely limited resources. To mitigate this performance concern, some recent software-based approaches [62, 63] have capitalized on high-level information in the source code. In [62], a compiler-based approach was proposed to enforce CFI for indirect jumps and calls. In [63], proposed was a new CFI technique called per-input CFI (PICFI), which utilizes the class constructor information to rewrite a target program's source code. This is necessary to augment the edge-adding codes, the main role of which is to activate the reachable target addresses in the *control flow graph* (CFG). At runtime, the augmented codes dynamically activate target addresses (or add edges to the CFG) lazily before the addresses are required later to execute indirect branches. Although both studies demonstrated that the number of exploitable indirect branch edges for attackers are effectively reduced with low performance overhead, their techniques cannot be applied directly to binary code that does not contain the high-level language features that can be utilized.

On the other hand, hardware solutions [6, 7, 8, 64, 1] tend to exhibit high performance because they can accelerate the CRA detection process with the assistance of

special hardware logics customized for the task. Specifically, authors in [6, 7, 8] proposed solutions in which the hardware logics are tightly coupled with the host CPU to monitor closely every control transfer during code execution. Despite considerable performance improvement, these solutions require the redesign of the existing processor architecture, which would stymie the direct deployment of these solutions into commercial smart mobile devices. This is because such a drastic modification to the internal core contradicts the common design practice for smart mobile devices. As the central computing platform for applications running on the device, an *application processor* (AP) in the form of SoC is used in each device. To meet ever-increasing demands for low design cost, high performance and fast time-to-market, the general design rule of SoC is to integrate commodity processor cores and supporting intellectual properties (IPs) for specific functions. Therefore, if AP vendors adopt some of these hardware solutions for their products, they will be compelled to spend their time restructuring the CPU core architectures, which is contrary to the general convention. In addition, such a drastic change to the host CPU, coupled with verification, will result in enormous cost.

To facilitate acceptance of hardware solutions for the CRA detection in today's smart mobile devices, some recent approaches [64, 1] have attempted to comply with the design rule of SoC. We claim that their systems are practical in the sense that they do not require any internal change to the host architecture; their hardware IPs are simply connected to the host via simple connections in order to build an SoC. In particular, they exploit the built-in debugging features to obtain correct control flow information related to the applications running within the host processor.

Although these approaches can achieve high performance in CRA detection, they face another challenging problem. In principle, in a debug environment that uses a hardware interface such as ARM CoreSight, the debugger is assumed to have the same binary code running on the host. Thus, to reduce the quantity of traces delivered to the debugger, the interface generally does not provide information that can be inferred

or simply extracted from the binary code. Unfortunately, omitted pieces of information such as branch types or source addresses for branch instructions are indispensable for accurate CRA monitoring. To overcome this lack of information, in a previous study [1], auxiliary information called *meta-data* was stored in the main memory region. In addition, the hardware IPs were given the task of reading the data at runtime. Despite negligible performance overhead, this solution suffers severely from substantial storage overhead as a result of the additional space necessary for the meta-data. Their experiments revealed that the size of the required storage for meta-data may be double that of the original application. Another limitation is that their study could detect only the *return-oriented programming* (ROP) attacks, which corrupt return addresses stored in a stack in order to chain gadgets. Although ROP attacks are representative examples of CRAs, there is another breed of CRAs, called *jump-oriented programming* (JOP) attacks. Their objective is to alter the target addresses of indirect calls or jumps. Therefore, to defend the system against CRAs successfully, the CRA monitoring hardware should be implemented with mechanisms that can detect not only ROP attacks but also JOP ones.

Based on our review of these previous studies, we propose two external hardware-based CRA solutions that can simultaneously monitor both ROP and JOP attacks on a system. The first solution, which we hereafter call the *hardware-based* solution, implements a *unified* ROP/JOP monitor entirely in hardware. To be applied to existing smart devices, we build the solution in a manner that the monitor can be integrated as IP modules into an ARM-based SoC. As in the previous study [1], the monitor is placed outside and connected to the ARM CPU through the CoreSight interface and system bus to monitor the host execution traces in a timely manner. In addition, we also try to avoid substantial storage overhead because of the vast amount of the meta-data. For this, we analyze the program binary with the help of compiler analysis techniques and instrument the binary in a way that missing essential information for CRA monitoring can be efficiently delivered on the fly from the host CPU through the debug

interface. This eliminates the need to store meta-data a priori for our monitor.

Although the hardware-based CRA solution can defend the system against most CRAs, fully implementing the necessary hardware functions may require non-negligible area overhead. In some cases in which deployed devices have stringent area constraints, this solution may not be a viable option. Therefore, to provide a balanced solution with respect to the level of security and the required system resources, we also suggest a software/hardware-combined CRA detection scheme on which various CRA detection algorithms [38, 40, 39, 65] can be deployed on the host CPU as a software module. For the sake of brevity, we refer to this solution as the *mixed* CRA solution throughout this paper.

Note that the mixed CRA solutions may incur substantial performance overhead onto the host system. This is because some necessary computations that detect CRAs will be delegated to the underlying processor. To address this performance issue, in this study, we divide the CRA detection process into two stages: *monitoring* and *inspection*. This decision is based on the observation that exercising exhaustive CRA inspection is wasteful because in-depth investigations for most branch operations are actually benigh and not under attack. Therefore, we attempt to avoid this wasteful computation by confining in-depth inspection only to the small fraction of branches that show anomalous behaviors. For monitoring as the preprocessing stage, we attempt to catch the first indication of attacks by incessantly monitoring all branch traces and discriminating suspected malignant branch operations from most benign ones. For example, CRAs generally form a chain of short code sequences each of which ends with an indirect branch instruction. In this study, this unusual operation is looked up during the program execution to identify an attack [38, 40]. Upon detecting such an operation, our first stage monitor interrupts the victim application that runs on the host CPU and invokes the OS to inspect the application‘s behavior in a more in-depth manner in order to determine whether the detected operation is indeed a true indication of CRA intrusion or simply a false alarm.

We believe that the success of this approach with respect to both the attainable performance and level of security depends considerably on the number of the inspection stage invocations. Obviously, it also depends on the type of algorithms that are to be deployed during the second stage. However, if the second stage rarely occurs, the administrators may have more freedom to determine the complexity of the detection algorithms employed when this stage does in fact occur, thereby increasing the security level of the system. To reduce the number of invocations, the branch monitoring stage must filter out most benign branch traces before transferring them to the inspection stage. In this study, this filtering task is conducted based on a set of rules dubbed the *alarming condition*. This condition defines the behaviors of suspectedly malicious branch traces that most CRAs commonly exhibit. The condition is constituted from a small set of primitive parameters that individually characterize branch behaviors such as the length or frequency of branch operations having specific types. For a more thorough description, we defer to the explanation of the parameters given in Section 4.5.

Regardless of how simple the alarming condition is, the monitoring task is still time-consuming. This is because in order to screen every branch behavior, it is necessary to keep track of all the branch outcomes by the host CPU. Therefore, to relieve the burden of the host from performing this computation, we designed the first stage monitor as a tiny hardware module, which is then plugged into the host CPU externally as a peripheral device. When an application of interest is initiated and scheduled as an active process, the OS kernel configures the alarming condition of our monitor and activates it. Once the monitor is activated for a new process, it operates independently of the kernel and keeps track of the branch traces emitted from the host CPU through the debug interface. When the monitor detects suspicious branch operations, it notifies the CRA inspection module in the kernel of the events.

This paper is organized as follows. In Section 4.2, we present the assumptions that underlie this study. Section 4.3 gives an overall architecture of our system. Then, through Section 4.4 and 4.5, we describe how we realize the security monitoring tech-

niques on our framework. Section 4.6 discusses the experimental setup and results. For the setup, we have used an ARM-based Zynq FPGA board [66] and prototyped our hardware modules to build a full SoC platform on the board. The results show that our prototype system offers a feasible security solution for protecting ARM-based SoCs against CRAs with high speed and low storage overhead.

4.2 ASSUMPTIONS

Throughout this paper, we make the same assumptions as used by previous studies on CRA defenses [40, 38, 7, 46]. Usually, modern OSes and the underlying processors cooperate to enforce the W$\oplus$X security protection rule [50, 51] which forbids a memory page to be both writable and executable at the same time. Also in this work, we hereby assume that the system is under the W$\oplus$X protection, and subsequently that the attackers cannot execute their injected code. Under this assumption, to disable the defense mechanism and do additional attacks, the adversary must gain sufficient privileges for the first time. We also assume that there are no other attack vectors or security holes that can directly escalate the attacker's privilege. Also, we suppose that there are memory corruption vulnerabilities such as buffer overflow ones in the victim application. By exploiting the vulnerabilities, the attackers can overwrite the control data (e.g. return addresses) stored in application's stack or heap in order to craft a CRA for arbitrary malicious actions such as privilege escalation, executing files and reverse connection. We do not consider the denial-of-service attacks in this paper. Furthermore, all underlying system operations performed by the kernel and hardware are assumed to be trusted.

We assume that an adversary knows all implementation details of the target application and locates the exact addresses of gadgets. Although ASLR [52] can prevent an adversary from locating the addresses of gadgets, we could assume that ASLR would not be enabled on the victim machine or the memory layout is revealed by memory

leak bugs like format string bugs even when ASLR is enabled. Also, we assume that the application is not compiled by attackers so that a number of useful gadgets cannot be contained within a small code base. Lastly, the self-modifying code is not considered in our assumptions because it conflicts with the W$\oplus$X security protection.

4.3 OVERALL SYSTEM ARCHITECTURE

In this section, we first give an overview of our SoC prototype architecture, and then give the description of the two stages (i.e., the branch monitoring and the inspection) of our CFI enforcement mechanism.

4.3.1 SoC Prototype Overview

Fig. 4.1 depicts our overall SoC design. As seen in the figure, our unified ROP/JOP detection module, called the *CRA monitor*, is externally placed and integrated into a SoC platform in which an ARM Cortex-A9 processor is used as the host core. The host CPU and our hardware module are connected via the standard AMBA3 AXI interconnect. In addition, to obtain the results of branch operations performed on the host without imposing any performance overhead on the host, we utilize the built-in hardware modules of the ARM CoreSight debug architecture, PTM (explained in Section 4.1) and the *trace port interface unit* (TPIU). In our implementation, the output signals of TPIU are directly routed to the on-chip ports of CRA monitor instead of the off-chip debug pins. The two Coresight modules are configured by the OS kernel via a device driver. Upon activation of the CoreSight modules, PTM packets are generated and transferred to the CRA monitor via the TPIU interface. It is noteworthy that, in terms of hardware design, the goal of our work is to build a practical and deployable hardware solution for CRAs on ARM-based SoCs. To achieve the goal, we adhere to the design convention of the commercial SoC platforms, where off-the-shelf ARM processors and newly designed hardware modules are integrated and connected only

through the existing communication channels, such as the system interconnect and the debug interface.

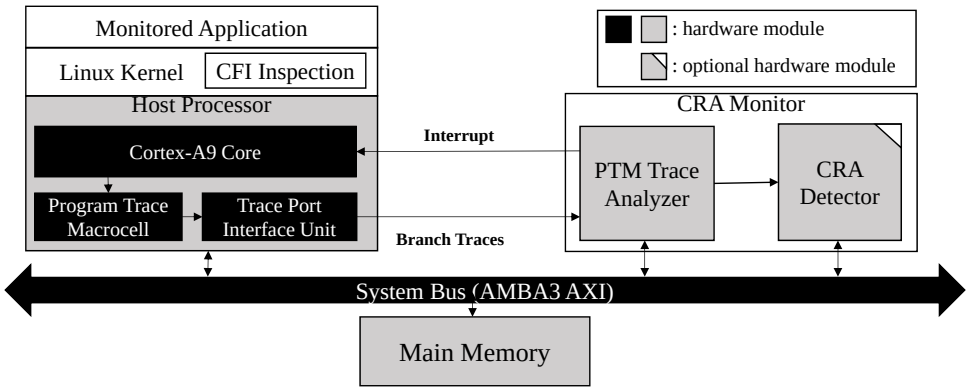


Figure 4.1: Overall framework of our SoC prototype

As mentioned in Section 4.1, we divide our CRA detection process into two stages: the branch monitoring and the inspection. Accordingly, the design of the CRA monitor reflects this grouping. Note that in Fig. 4.1, the CRA monitor has two sub modules, the *PTM trace analyzer* (PTA) and the *CRA detector*. To incessantly, yet efficiently, handle every branch trace coming from TPIU, PTA is implemented as a customized hardware module and included in the design regardless of whether it is a lightweight solution or not. The CRA detector, on the other hand, is implemented in hardware only if it is a hardware-based solution. When a lightweight one is being considered, at design time, the CRA detector is removed from the SoC, and the inspection stage is implemented, if necessary, on the host kernel in the form of a loadable kernel module. More details are given later in Section 4.4 and 4.5, each of which discusses (1) the hardware-based unified ROP/JOP monitor and (2) the lightweight mixed solution, respectively.

4.3.2 CRA Detection Process

Fig. 4.2 depicts the overall process of CRA detection in our approach. As stated in Section 4.1, our CRA monitor detects both ROP and JOP attacks. The necessary in-

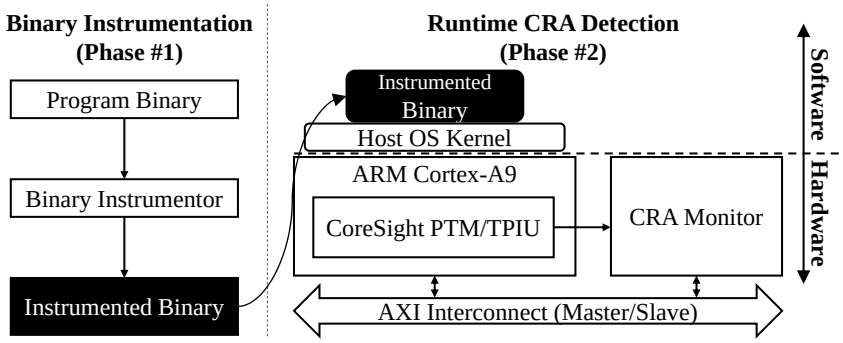


Figure 4.2: CRA detection process with the CRA monitor

formation to detect both types of attacks varies widely according to the required level of security and the type of detection algorithm one wants to deploy on the system; for instance, if one wants to employ a detection algorithm introduced in [38], the CRA monitor requires only the recent 8 or 16 indirect branch traces. In our work, especially for the hardware-based solution, we have realized the detection algorithms based on those proposed in [45] and [6], respectively, and therefore, for the rest of this subsection, we will explicate the detection process of our solutions based on these algorithms to ease explanation.

To detect ROP-based attacks, we copy the return address of every call instruction in a special stack buffer called the *shadow stack* and check the target address of each return instruction with the value retrieved from the top of the shadow stack. Therefore, the necessary information to implement the shadow stack into our system are (1) the target address of return instructions and (2) the source address of call instructions to calculate the address to be returned later. Unlike ROP, JOP usually creates a code sequence by linking gadgets together with indirect jumps or calls. Hence, to launch JOP attacks, instead of altering return values stored in the stack, attackers try to corrupt code pointers such as function pointers, which will be used as the target addresses of indirect calls or jumps to point to their gadgets. The JOP detection algorithm is on the ground of a simple invariant ruling the normal behaviors of branches in a pro-

gramming language. The invariant rule says that, in a normal program execution, the target address of a call instruction should point to the address of a function entry, and that of each indirect jump should always point to an address within the same function that the instruction belongs to [6]. To check this legitimacy to detect JOP attacks, the CRA monitor has to obtain the information about (1) the target address of call instructions, (2) the target address of indirect jump and (3) function boundaries which contain the entry and end addresses of functions. To summarize, the essential information to simultaneously check the existences of ROP/JOP attacks from outside the CPU is categorized into four classifications:

(1) Target address of indirect branches (i.e., indirect calls, indirect jumps and returns)

(2) Source address of call instructions

(3) Function boundaries

(4) Branch type to classify the branch instructions

Recall that, to reduce the quantity of generated traces, the ARM debug interface generally does not provide the information which can be directly derived from the binary code. In fact, only the target address of an indirect branch and the direction (taken/not taken) of a direct branch can be acquired from the traces coming through the debug interface. Gathering the target addresses of indirect branches are quite straightforward in our solution as the ARM debug interface is designed to provide such information. However, the other classes of information cannot be directly acquired from the debug interface, and therefore we have devised a special mechanism where we instrument the original binary to supply the lacking information. For this purpose, we built an in-house tool called the *binary instrumentor* that can statically instrument the target binary (phase 1). It basically analyzes and generates binary code in a way that all lacking pieces of the information for CRA detection will be explicitly delivered to the

CRA monitor, either through the ARM debug interface or the system bus. When the program binary is downloaded by the OS kernel into the local storage such as a disk or a flash memory, the instrumentor generates the instrumented version of the binary and stores it into the storage. More detailed explanation will be given in Section 4.4. After the instrumented code is loaded, the CRA monitor performs its task of constantly watching the runtime traces gathered from both TPIU and the system bus and checking if there is any behavior possibly related to CRAs (phase 2).

4.4 FULL HARDWARE IMPLEMENTATION

In this section, we introduce the CRA detection scheme which is fully implemented in hardware. We first give the detailed description of our instrumentation scheme. And then, the hardware architecture of our CRA monitor will be provided.

4.4.1 Binary Instrumentation

As briefly discussed in Section 4.3, we propose a binary instrumentation technique that enables us to derive from the branch traces of the host system more information that includes not only the target addresses of indirect branches but also the branch types and the source addresses of call instructions. As the first step of the instrumentation, the binary instrumentor scans the entire code to find all function call instructions. These are executed by either a `bl` (branch with link) or a `blx` (branch with link and exchange) instruction in the ARM architecture. To deliver the information associated with the call instructions, we introduce a new code section called the *trampoline*.The trampoline section is added to the end of the original binary. Each call instruction in the original binary is moved to an associated location in the trampoline and the original instruction is replaced with an indirect jump which targets the associated place; specifically, for each direct call (`bl` or `blx`) with an immediate offset moved to the trampoline, the offset is recalculated so that it can target the same address as the origi-

nal instruction pointing to. At the same time, to maintain the current offsets for direct branches and the function addresses, we use another intermediate code to set a register and execute an indirect jump with the register. For each call in the trampoline, there is a unique stub which contains a direct jump to the next address of the original call. This stub is the target of the subsequent return instruction executed in the callee function. In Fig. 4.3, we present an example of the instrumented binary code for the original one. Note here that the newly added parts are written in boldface.

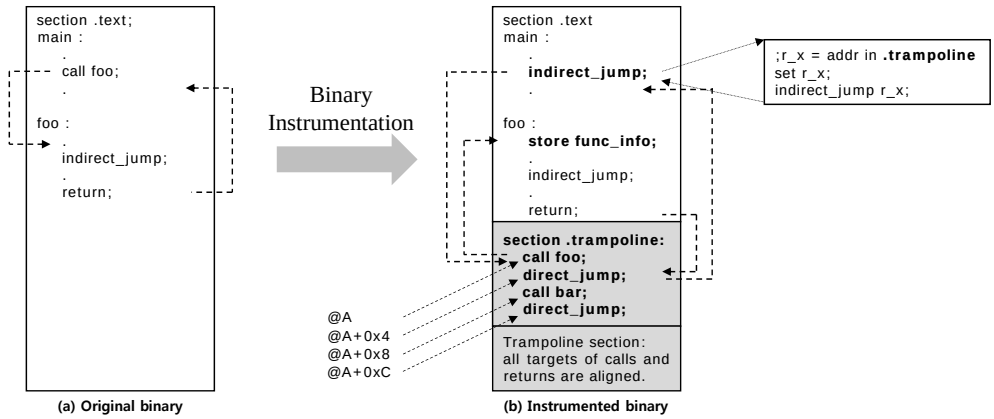


Figure 4.3: Original vs. instrumented binary

As shown in Fig. 4.3, when the address of the trampoline entry is A, every call instruction is aligned at addresses $A + 8 * n$, while the targets of return instructions are aligned at $A + 8 * n + 4$ (n is an integer and $0 \leq n \leq total\ number\ of\ calls$) . Using these aligned data, the types of the executed branch instructions can be classified by simply checking the target address coming from TPIU. Especially for a call instruction, an indirect jump to the trampoline is followed either by a target address or by a direction (taken/not taken) in the branch traces from TPIU. When a target address follows the indirect jump, the branch type is considered to be an indirect call. Otherwise, it is decoded as a direct call. Note that all the calls are pointed to by indirect jumps as a result of the instrumentation. This means that the source address of each call can then be obtained from TPIU because the target address of the indirect jump

pointing to $A + 8 * n$ is now the source address of the call. This allows our monitor to calculate the legitimate destinations of return instructions which are necessary to maintain the shadow stack for ROP detection.

Moreover, to detect JOP attacks, the function boundary information is indispensable in checking whether the target address of an indirect jump falls inside the function body where the current PC resides. Thus, in our instrumentation scheme, each function is transformed in a way that it can start with an annotation code (`store func_info;` in Fig. 4.3(b)) that writes the entry address and size of the function to the memory-mapped addresses of our hardware modules through the system bus. The binary instrumentor can identify the entry address and size of each function by referring to the symbol tables of executable formats such as the executable and linkable format (ELF).

4.4.2 Hardware Architectures

Fig. 4.4 shows the hardware structure of our CRA monitor including PTA. In the current implementation, the output signals of TPIU are routed directly to the on-chip ports of our CRA monitor. Because the host CPU generally operates considerably faster than other hardware IPs such as our CRA monitor, we implemented in PTA an asynchronous buffer called the *branch trace first-in-first-out (FIFO)*, which temporarily stores the traces coming from TPIU. Another submodule in PTA called the *trace decoder* then analyzes these stored traces to obtain the target addresses of indirect branch instructions and the direction (taken/not taken) of the direct branches. With this information, the decoder further extracts the branch types and source addresses of calls as discussed previously. Finally, for each branch instruction, its type and associated information (i.e., source addresses for calls and target addresses for indirect branches) are conveyed to the CRA detector in order to monitor CRAs.

Fig. 4.5 shows the hardware architecture of our CRA detector which monitors the host execution traces in order to detect simultaneously both ROP and JOP attacks. To

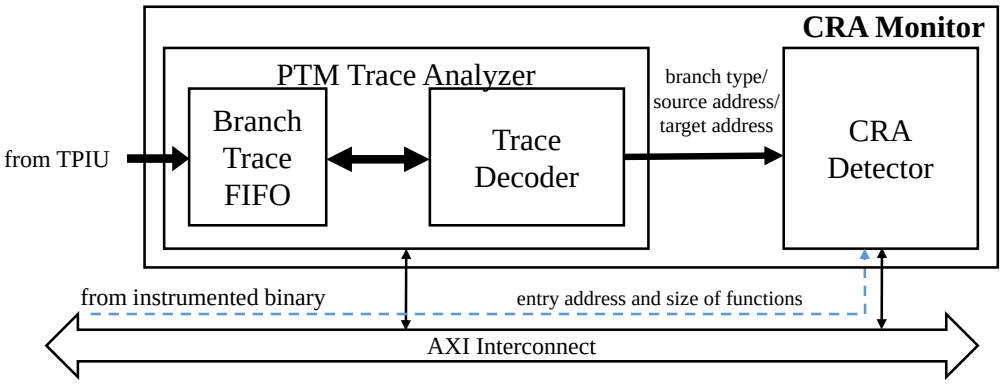


Figure 4.4: CRA monitor hardware architecture

determine whether CRAs are present, our detector relies on the aforementioned branch information fed by PTA as well as the entry addresses and sizes of functions coming through the system bus.

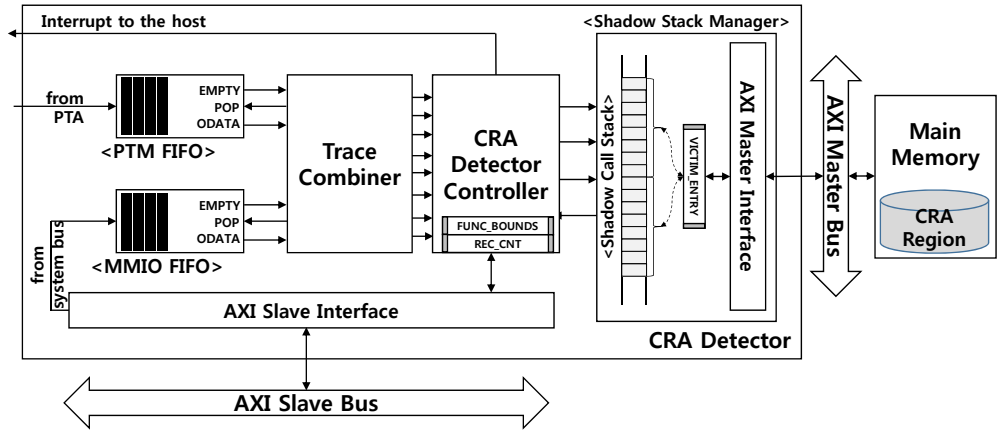


Figure 4.5: Hardware architecture of the CRA detector

Note that the information from different sources (i.e., TPIU and the system bus) have no ordering restrictions. For this reason, the CRA detector must combine and rearrange the information from the two sources to correctly keep track of the original program sequence of the application. To perform this task, the detector has two separate FIFO buffers, called the *PTM FIFO* and the *MMIO FIFO*, to store temporarily

the information received from PTA and the bus, respectively. The output signals of both the FIFOs are given as input to the *trace combiner* (TC), which is responsible for combining the information from the two FIFOs and extracting the original program execution behaviors. We present an example of the information flow from the two sources and the manner in which they are combined by TC in Fig. 4.6

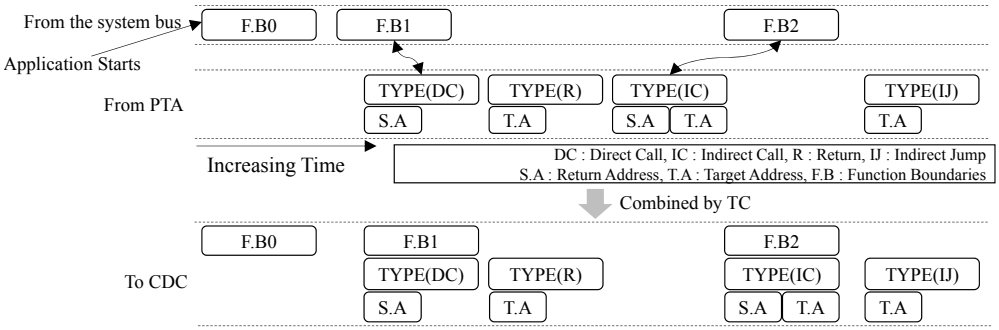


Figure 4.6: Information flow from PTA and the system bus to the CRA detector

As exemplified in the figure, when the application begins, the program flow encounters the initially invoked function, main(), for the first time. TC is notified of this special event through the MMIO FIFO so that TC can begin operation (see as F.B0 in Fig. 4.6). At runtime, when the program runs into a call instruction, the instrumented code at a function prologue is executed right after the call instruction. It thus delivers the branch information (branch type and the associated information) as well as the function boundary information through the PTA and the system bus, respectively. When any of these events arrives and is stored in either the MMIO FIFO or the PTM FIFO, TC reads it and determines an appropriate action. If the function boundary information is written to the MMIO FIFO, TC waits for an event to arrive from the PTM FIFO. Once the PTM FIFO obtains an entry, TC checks the branch type, and if the type is either a direct or an indirect call, TC combines the pieces of information (i.e., the branch type, the function boundary information and the source address for a direct call as depicted in the figure) from both FIFOs. Otherwise, only the information from the PTM FIFO is selected. This information is then delivered to the *CRA detector con-*

troller (CDC) whose purpose is to make the final decision regarding the existence of CRAs.

After the application starts, CDC first expects to receive information about the initially invoked function fed by TC before anything else. Upon receiving the information, CDC calculates the entry and end address (= entry address + size) of the callee function and stores them into a register called `FUNC_BOUNDS`. Later when the branch type coming from TC is a call, CDC also obtains the entry address and size of the callee function from TC. Especially for an indirect call, if its target address is not matched with the incoming function entry address, it means that the call jumped to an unknown address, which is a typical behavior exhibited by a JOP attack. If the call instruction is a direct one or verified to be benign, CDC pushes the concatenated value of the return address (= source address + 0x4) and `FUNC_BOUNDS` onto the *shadow call stack*, whose job is to maintain a shadow copy of the call stack on the host. The reason why `FUNC_BOUNDS` is saved into the stack is that its value should be restored when the callee function returns later. At the same time, CDC overwrites `FUNC_BOUNDS` with the newly calculated entry and end addresses of the callee function. When a function returns, CDC pops the top entry of the stack and compares the saved return address against the target address coming from TC. If there is a mismatch, it means that the return address in the host stack is maliciously manipulated by ROP attacks, and consequently CDC issues an interrupt. Otherwise, CDC overwrites `FUNC_BOUNDS` again with the saved function boundaries. When an indirect jump is made in the host, CDC will check whether or not its target address falls between the entry and end addresses of the currently running function by referring to `FUNC_BOUNDS`. If the address points to outside the function boundaries, CDC deems that this is the act of a JOP attack, and immediately notifies the host of this attack by setting the `interrupt` signal on. Note that, in the current implementation, the target addresses of indirect jumps can be within the trampoline section due to the instrumentation process. This exceptional case is not signaled as an error case by our hardware and is considered to be a legal

state transition.

Note that the shadow call stack has a finite number of entries, specifically, 16 in this study. Due to this limited stack depth, it would be overflown if the target application has more than 16 times nested function calls. To cope with this limitation, we implemented a special stack management module called the *shadow stack manager* (SSM). When the shadow call stack fills up with deeply nested calls, SSM copies the oldest 8 entries to the pre-defined region, called the *CRA region*, in the main memory through the *AXI Master Interface*. In addition, we implemented a register called `VICTIM_ENTRY`, which plays the role of victim cache storage to store temporarily the most recently evicted 8 entries. Moreover, an exceptional case exists in which the host program calls the same function recursively. In response to this, CDC employs a counter register, which we refer to as `REC_CNT`, to store the number of recursive calls. When the same function is called consecutively, CDC increases the counter value by one without pushing any value onto the stack. When the function returns and `REC_CNT` has a non-zero value, CDC decreases the counter value by one instead of reading the top stack value.

4.5 LIGHTWEIGHT MIXED CRA DETECTION SOLUTION

As explained in Section 4.1, the main idea of the mixed approach is to confine the CRA inspection only to the small fraction of branches showing anomalous behaviors that are suspected to be involved in an attack. In our solution, the inspection stage is implemented as software, whereas the branch monitoring stage is exercised by the customized hardware. Thus various CRA detection schemes [61, 67, 46, 6, 68] can be selected by administrators for use with the system by incorporating them as software modules into our inspection code.

4.5.1 Hardware Architectures

Fig. 4.7 shows the hardware structure of our CRA monitor subsystem. The subsystem is divided into three components: PTA, the *branch behavior analyzer* (BBA) and the *branch history buffer* (BHB).

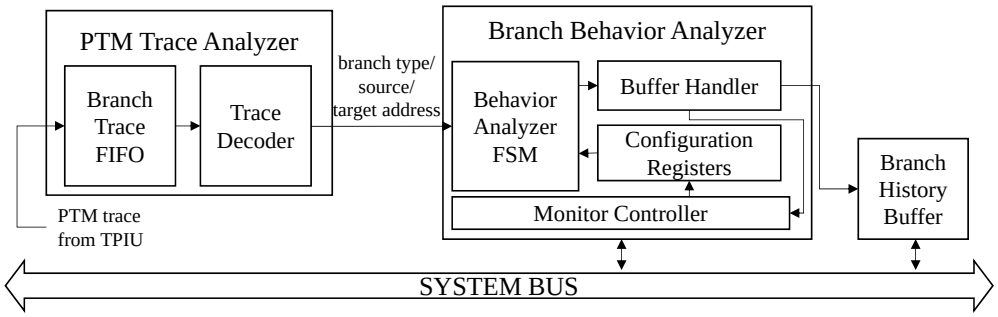


Figure 4.7: Overall hardware structure of the CRA monitor subsystem

The role of BBA is to diagnose a symptom of CRAs from the branch traces of victim processes at an early stage and to notify the host of seemingly malicious events to initiate further inspection. For this task, BBA examines the input traces to determine if any shows abnormal behavior based on alarming conditions possibly resulting from malicious acts of a CRA. Therefore, the success of our approach hinges heavily on the accuracy of the alarming conditions. If the conditions are loose, the host machine will suffer a performance problem because the kernel will receive too many branch traces as input for CRA inspection. On the contrary, if they are too tight, the false-negative detection problem may arise in which we fail to detect a CRA because we ignore the symptom of the attack indicated by the branch traces.

In an attempt to identify optimal alarming conditions for our CRA mechanism, we have studied and categorized diverse known attacks, which can largely be divided into ROP [69, 70, 71] and JOP attacks [72, 73, 74]. In addition to the basic attack behaviors, there are also variations allied with some tricks to deceive existing detection schemes by adding mutations to the gadget structure [7, 74, 75] (Such gadgets include

long gadget, call-preceded gadget and entry-point gadget). Although there are many variations of CRAs with distinctive features, there has been known rules that are applicable to both the current and possibly the future CRAs. One of the rules adopted as our alarming condition is that a CRA utilizes indirect branch instructions to chain their gadgets. For example, ROP constructs an attack by using return instructions which can change the control flow to the addresses saved in the link register (LR) or the stack. The same rule can be applied to JOP attacks except that indirect calls or jumps substitute for the returns in ROP attacks. Even with advanced attack schemes such as the JOP with long-gadgets [73], the rule remains valid.

To summarize, the aforementioned rule should work for almost every, if not all, CRA. This is because malicious code for a CRA by nature involves several indirect branches consecutively executed. Therefore, by thoroughly examining the branch traces of a process, we can diagnose early whether process is presumably under attack. In other words, if this type of branch behavioral pattern is identified, we may articulate a reasonable suspicion, though not being conclusive, that a CRA may currently be present in the system as this pattern is hardly be observed in a healthy system with normal processes. Based on this reasoning, we can formally state our rule that when encountering consecutive occurrences of indirect branches in a trace, determining the existence of a CRA in the system is possible. From this rule, we establish the following alarming condition for our solution;

- *More than γ indirect branches appear consecutively one after another in the incoming traces.*

Here, the threshold γ is a parameter that can be configured by the kernel or system administrator.

Although many previous solutions [73, 61, 7] have shown that this rule is valid for a majority of CRAs, it remains possible for adversaries to compose attacks in a manner that nullifies the rule. One such means is to insert direct branches into their

gadget chains to pretend to be innocent. This new type of attack, called a *gadget gluing attack* [40], constructs a special gadget that consists of two short code sequences glued together by a direct branch instruction. To the best of our knowledge, the gadget gluing attack has not been witnessed. However, if it exists, it can bypass the aforementioned alarming condition whereby it only looks for consecutive indirect branch instructions in the traces. To address this potential threat, we add to our alarming conditions another threshold parameter δ, which represents the maximum number of direct branches that can come between two indirect ones. Thus, to reduce the possibility of false-negative detection, we loosen the original alarming condition as follows;

- *More than γ indirect branches appear in the incoming traces, and between two adjacent indirect branches in their appearance order, at most δ direct branches may arbitrarily come.*

As depicted in Fig. 4.7, BBA has a set of registers, named as the *configuration registers* that store alarming conditions. The content of registers can be configured by the kernel using the *monitor controller* logic, which is connected to the slave interface of the system bus. Each of the registers is assigned a unique memory address so that the host can write any data to the address assigned to the register. Based on the parameters stored in the registers, BBA checks whether the branch traces from the host exhibit any anomalous behaviors. For this task, BBA contains a tiny finite state machine, called the *branch analyzer FSM* (BAF). Whenever PTA sends a set of decoded branch signals, BAF first identifies the type of branch. If it identifies an indirect branch, it increments its internal counter for γ. Whenever it observes a sequence of consecutive indirect branches, it repeatedly increments the counter accordingly. Finally, when it reaches the pre-defined threshold value γ, BAF triggers an alarm to inform the host of the potential CRA attack. To handle the threat of a gadget-gluing attack [40], the developer can also configure BAF to allow direct branches to come between indirect ones. When this option is enabled, a specific number of direct branches between two indirect ones are allowed by configuring δ. BAF does not reset the counter in such cases.

When BAF finds the possibility of an attack, the associated information must be transferred to the host OS kernel for detailed inspection. In BBA, this mission is performed primarily by a submodule, called the *buffer handler*. Every time the target address of an indirect branch is routed from PTA, the handler copies it to BHB in the order it was received. Therefore, when BAF identifies a suspicious branch behavior, BHB will contain the branch history of the victim process. When this event occurs, BBA stores the current index of BHB in the special register inside the monitor controller and issues an interrupt signal through the monitor controller (see Fig. 4.7) to notify the host kernel of a possible CRA occurrence in the host system. Then using the system bus, the kernel reads the BHB entries to obtain the branch traces.

4.5.2 Implementing CRA Inspection Software on Our Framework

Fig. 4.8 presents an example of CRA inspection code (running in kernel mode) that can be implemented on the host CPU. On the left side, a list of supposedly malicious branch traces that are stored in BHB await further analysis to determine the true nature of their malignancy. Our inspection code has a subroutine code, called the *interface*, whose purpose is to communicate with the CRA monitor.

The list contains the target addresses of indirect branches and, if the administrator chooses to include them, the direct branch results (taken/not-taken). As mentioned in Section 4.4, the traces alone may not provide sufficient information for our inspection code to interpret current branch behaviors. Recall that, apart from target addresses of indirect branches, several attributes of an indirect branch exist that may be useful for branch behavior analysis.

The required attributes for CRA detection depend on an algorithm that is to be deployed on the host system. In this paper, as an example, we implement well-known heuristic algorithms introduced in [38, 40] for both JOP and ROP attacks. These algorithms define a few rules that most benign programs comply with; the rules indicate that a return instruction always points to the call-preceding instructions, an indirect

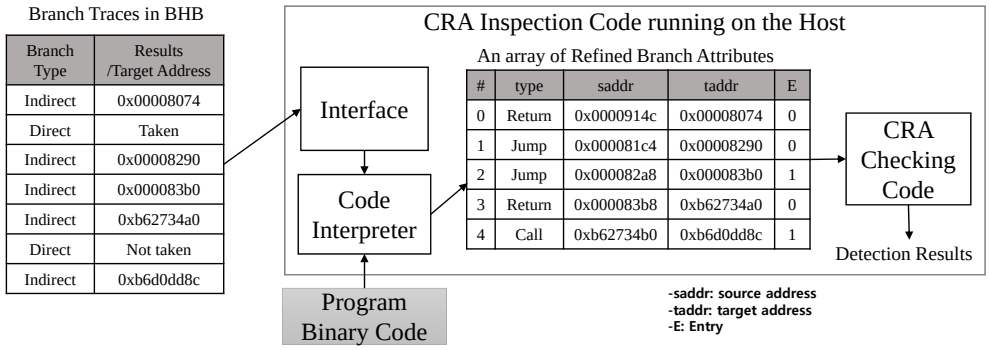


Figure 4.8: Structure of the CRA inspection code

call should transfer the control flow only to a function entry, and the number of instructions between two adjacent indirect branches tends to be large. To extract the attributes required by these algorithms, we implement another subroutine code, called the *code interpreter*, which mainly performs code analysis on the monitored program binary. Given a list of the traces, the code interpreter accesses their associated code sections in the monitored program and extracts necessary information for the heuristic algorithms. In our example, for each branch retrieved from BHB, the code interpreter accesses the binary code to identify the corresponding source address and determine whether the target address points to the function entry. With these additional branch attributes as an input, the *CRA checking code* finally determines whether the branch behaviors comply with the rules we described previously and whether the seemingly malicious branch behaviors truly indicate CRAs.

4.6 EXPERIMENTAL RESULTS

4.6.1 Experimental Setup and Synthesis Results

To evaluate our approach, we implemented a full-system SoC prototype on a Xilinx ZC 702 evaluation board [66]. This development board, which is based on the Zynq-7000 XC7Z020-1CLG484C platform, is equipped with a dual-core ARM Cortex-A9

processor, ARM NIC-301 AXI network interconnect, an FPGA chip that can accommodate ASIC of up to 1.3 million gates, 1GB DDR3 SDRAM and other peripherals. The host system is based on the A9 processor, and Xilinx ARM Linux Kernel 3.8 is used as the OS. We enabled the two CoreSight modules in the Coretex-A9 processor, PTM and TPIU, to extract branch traces for our experiment. All the hardware modules we described in the previous sections were implemented in Verilog HDL and mapped on the FPGA. Mainly due to the speed limit of FPGA, our CRA monitor subsystem was configured to operate at 60 MHz and the host clock was lowered to 150 MHz to conform to the performance ratio between the host and the coprocessors in most AP platforms [56].

Based on these parameters, we synthesized our prototype onto the FPGA chip in the evaluation board and quantified the logics necessary for the hardware modules in terms of lookup tables for logic (LUTs) and block RAMs (BRAMs). We provide the synthesis results for two cases, full hardware-based and mixed implementations. The synthesis results, shown in Table 4.1, reveal that our hardware-based CRA monitor occupies 5.96% (3,271/53,200) of all LUTs and 0.02% (3/17,400) of all memory elements. The lightweight CRA monitor occupies 3.3% (1,759/53,200) of all LUTs and 0.02% (2/17,400) of all BRAMs. Considering that the ARM Cortex-A9 dual core processor requires approximately 26 million gates [30], we claim that the area overhead that was induced to the design as a result of our security monitors was acceptable.

4.6.2 Analysis of Full Hardware Implementation

To evaluate the detection capability of our CRA monitor, we have implemented five types of CRAs based on the Shell-storm shellcode [60], as shown in Table II. Four attacks (A1, A2, A4 and A5) are crafted to open a new shell, thus enabling attackers to enter arbitrary commands. A3 modifies the memory page attributes (writable/executable) where the attacker's own code is placed with the `mprotect` system call. To glean necessary gadgets, we have leveraged as our code base three general libraries found in

Category	Component	Logic LUTs	BRAMs
Hardware-based Approach	Branch Trace FIFO	96	1
	Trace Decoder	1209	0
	MIMO FIFO	265	1
	PTM FIFO	63	1
	Trace Combiner	195	0
	CRA Detector Controller	206	0
	Shadow stack manager	1138	0
	Total	**3172**	**3**
Hardware/Software Mixed Approach	Branch Trace FIFO	96	1
	Trace Decoder	1209	0
	Behavior Analyzer FSM	115	0
	Buffer Handler	42	0
	Configuration Registers	38	0
	Monitor Controller	175	0
	Branch History Buffer	120	1
	Total	**1795**	**2**

Table 4.1: Synthesis result

libwebcore in Android 4.2.2, libc-2.13 in Xilinx-linux and libc-2.15 in Ubuntu.

By launching the implemented attacks on our system, we tested the effectiveness of our monitor in terms of security. As we expected, all the ROP attacks (A1-A3) are detected by the hardware-based CRA monitor, mainly thanks to the shadow stack. Because the attacks violate the general function calling convention, their suspicious behaviors are always captured by our monitor even when the adversaries employ advanced skills like long-gadgets. As to the JOP samples (A4-A5), they are crafted by using `blx` (indirect call) or `bx` (indirect jump) instructions to weave their gadgets. In these attacks, every used `blx` instruction does not target an entry of a function. Similarly, all the target addresses of `bx` instructions are always beyond the current function bounds. Consequently, all their illegal behaviors are detected by our CRA monitor.

It is worth noting here that our monitor may not be able to detect a few advanced

Attack No.	Type	Goal	Advanced Skill	Detection
A1	ROP	Open a shell	-	√
A2	ROP	Open a shell	Long-gadget	√
A3	ROP	Invoke a mprotect system call	-	√
A4	JOP	Open a shell	-	√
A5	JOP	Open a shell	Long-gadget	√

Table 4.2: The description of implemented CRAs and detection results of the attacks

attacks recently introduced in [75, 76]; they present special gadgets that can build attacks that do not violate any aforementioned rules (legitimate control flows of the target application). However, to the best of our knowledge, no prior work has proven the existence of gadgets that can achieve Turing completeness. Moreover, as pointed out in [76], a majority of such advanced attacks is still defeated by defense mechanisms with shadow stacks. Therefore, we claim that employing our monitor, which is fortified with the shadow stack, may substantially raises the bar to attackers, because the attackers are forced to use only a limited set of gadgets that adhere to the basic function invocation rules under the guard of our monitor.

To assess the performance overhead of our hardware-based CRA monitor, we chose eight applications from the SPEC CPU2006 benchmark suite [77]. For this experiment, we provide two different configurations first of which is called the *Base* and the other the *wCRA*. The former acts as the control group where the original code runs on the host CPU with our monitor disabled. The latter refers to the same host settings, but with our CRA monitor enabled. We present the test results in Fig. 4.9 where the runtime of each application with *wCRA* is normalized to that of *Base*. The empirical results show that the performance overhead is 4.51%/10.68% (average/max) over *Base*.

As already explained, although the hardware-based solution necessitates the target binary being instrumented a priori, it eliminates the needs of meta-data. To evince

Benchmark	Original size (a)	Ours		[8]	
		increased code (b)	(b)/(a)	meta-data (c)	(c)/(a)
bzip2	503,664	88,020	0.1748	797,144	1.5827
mcf	464,775	82,836	0.1782	725,992	1.5620
milc	588,408	114,564	0.1947	1,169,487	1.9875
gobmk	3,973,190	286,032	0.0720	1,856,400	0.4672
hmmer	764,042	156,472	0.2048	1,147,512	1.5019
libquantum	561,254	96,804	0.1725	863,652	1.5388
h264ref	1,000,235	143,916	0.1439	1,474,460	1.4741
astar	579,187	107,604	0.1858	885,456	1.5288
average			**0.1658**		**1.4554**

Table 4.3: Comparison of binary sizes between ours and [1]

this, we compared the storage overhead due to our instrumentation for hardware-based approach with the overhead incurred by the meta-data proposed in [1]. Even though the meta-data has been introduced to accelerate the overall detection process, it induces substantial storage overhead proportional to the code size of the target application. Although our approach also requires the binary code running on the host CPU to be instrumented with additional instructions, we argue that the amount of additional code is rather small compared to the previous approaches. To support this argument, we measured the amount of memory required for the CRA detection suggested in [1] and ours, as presented in Table III. As seen in the figure, our approach needs slightly more memory than the original, uninstrumented code, but requires far less memory (on average 16.58%) than that of the technique proposed in [1] (on average 145.54%).

The above results clearly show the advantage of our approach over the previous work [1] in terms of memory usage. The removal of the meta-data also gives us another advantage that our hardware IPs no longer need to read a large quantity of data from the main memory at runtime. Although the experiment in [1] reported that their performance overhead is about 3% due mainly to memory contention between the host and their monitor, which is slightly better than ours, we have discovered that their approach relying on massive memory accesses for meta-data inherently entails a serious

ARM_CLK:IP_CLK	1:1	2:1	3:1	4:1	5:1
Ours	o	o	o	o	o
[1]	o	x	x	x	x

Table 4.4: Frequency gap tolerance of ours and [1]
(IP_CLK is for both the monitor and the DDR memory)

flaw. In their work, the latency to the main memory such as DDR has to be paid for processing each branch trace coming from the debug interface. Since it requires the reference to the meta-data, the processing capability of the monitoring hardware is severely limited. This trend gets more obvious when the user wants to increase the CPU frequency for the higher host performance or decreases the DDR frequency for the less power consumption. In these conditions, branch traces are more likely to be dropped without being analyzed.

To put forward evidence to support the hypothesis, we measured the operable frequency gap between the host CPU and our CRA monitor. For this experiment, we implemented the ROP monitor in [1] and checks how slow their monitor can operate while correctly performing the CRA detection. Then, we compared the result with that of ours in Table 4.4. Both of them are configured to have the same depth of the input buffers (32 in this experiment) to temporarily store the incoming traces from TPIU. As we expected, ours tolerates up to the 5:1 frequency gap without overflowing the buffer. On the other hand, the work in [1] cannot stand even the 2:1 frequency gap. This result indicates that their solution does not function correctly for more realistic SoC architecture models where the host CPU is much faster than external devices like our monitor. In this sense, we believe that our approach is more acceptable in real-world systems such as APs of modern smartphones [56] whose frequency gap between the host CPU and other auxiliary IPs are typically configured up to 5:1.

4.6.3 Analysis of Mixed Hardware/Software Implementation

Before the monitoring process starts in our mixed solution, we must define the alarming conditions by configuring the parameters γ and δ, which describe the suspicious branch behaviors of interest. Recall that, we define our alarming condition as follows; more than γ indirect branches appear in the incoming traces, and between two adjacent indirect branches in their appearance order, at most δ direct branches may arbitrary come.

Based on the values for the parameters, a trade-off relation exists between the detection accuracy and the performance overhead. As γ decreases, it becomes increasingly difficult for attackers to construct CRAs without being detected. In addition, the probability that benign branch behaviors are identified as abnormal will increase. This in turn increases the computational burden of the checking engine. By contrast, the higher γ is, the greater is the number of false negatives that will be generated. This is because the number of gadgets allowed for attackers also increases. However, the performance overhead reduces dramatically.

To explore this trade-off, we conducted an experiment to evaluate the *inspection engagement rates*, which are defined as (the number of indirect branches delivered to the inspection code) / (the number of total branches). Fig. 4.10 shows the inspection engagement rates for various alarming conditions as we changed γ and δ for the execution of eight applications from the SPEC CPU2006 benchmark suite [77]. As expected, the inspection engagement rate substantially decreased as γ increased. For example, if γ was set to 4 when δ was set to 0, only 0.000389% of all branches were delivered to the inspection code. Thus, if we set γ to a high number, frequent invocations of the inspection code running on the kernel could be avoided.

However, a high threshold would provide attackers with more freedom in crafting their gadget chains. As long as the number of consecutive indirect branches remains below the threshold, the monitor inadvertently filters such events (and incurs false positives), thus being unable to engage the inspection code running on the ker-

nel. Regarding security, the configuration having a small γ value produces fewer false positives. However, if we set the monitor to a γ value that, as a triggering parameter, is too small, the inspection code on the kernel will most likely be overwhelmed by many benign consecutive indirect branches. Each of these branches, as well as the associated gadgets, must be inspected for possible CRAs.

The second parameter of the alarming condition is δ, which is the number of direct branches allowed between indirect branches. This parameter is necessary for the monitor to handle potential gadget-gluing attacks [40], that is, when this theoretical CRA attack becomes practically feasible in the future. In Fig. reffigure:EngagementRate, the configurations, "δ = 1" and "δ = 2", refer to possible inspection engagement rates. As shown in Fig. reffigure:EngagementRate, the rates of both configurations are greater than that of "δ = 0". For example, when γ is 4, the inspection engagement rate of "δ = 2" reaches 0.625%, which is much higher than that of "δ = 0", or 0.000389%. Nevertheless, if γ is 5 or greater, the inspection engagement rate of "δ = 2" remains less than 0.002%. This shows that a sufficiently large parameter for the number of consecutive indirect branches can lower the inspection engagement rate. Fig. 4.10 also shows that the lower γ is, the more frequently is the processor interrupted by our hardware. Consequently, the interrupt handling time (for CRA detection algorithm) is expected to dominate the performance overhead. To support this statement, we measured the performance overhead for each case and depict the results in Fig. 4.11. As shown, the performance overhead is nearly proportional to the engagement rate (i.e., the number of triggered interrupts that the processor should handle).

To test the detection capability of our solution, we used the same attacks described in the previous subsection. In this experiment, we set γ to the value used in the previous study [40], which is 10, because this latter study targeted a Linux environment just as we did. The value of δ was set to 2 in order to mitigate the gadget-gluing attack. However, this did not affect detection accuracy because none of our attack samples used the attack technique. Recall that the purpose of this solution is to help the sys-

tem perform the CRA enforcement with configurable parameters efficiently. As shown by the experiment listed in Table 4.2, our monitoring solution can detect all the ROP attacks, which violate the rule of a return instruction (i.e., that it must point to call-preceding instructions). The implemented JOP samples used `blx` (indirect call) or `bx` (indirect jump) instructions to link their gadgets. In these attack samples, every used `blx` instruction did not target an entry of a function. Consequently, all their illegal behaviors were detected by our monitor. Note that, for the mixed CRA detection scheme, the area overhead remained the same because we configured only the parameter values (γ and δ) using the same underlying hardware.

Note that, for the mixed CRA detection scheme, the area overhead remained the same because we configured only the parameter values (γ and δ) using the same underlying hardware. Regarding security, the leftmost case (γ = 1) in Fig. 4.10 yields the least false positives, and conducts the most frequent and thorough investigation of the system. By contrast, the lower γ is, the greater the number of interrupts triggered to the processor, and the greater the amount of performance overhead incurred.

Again, our current implementation may be defeated by advanced CRAs proposed in [75, 76] as the heuristic algorithms employed on our platform are relatively simple. However, the main goal of using such algorithms on our framework is to demonstrate a means by which our monitoring solution can work with the host system, not to provide a conclusive CRA detection solutions. Therefore, we believe that deploying more sophisticated algorithms on the host system would improve security to a level considerably greater than that achieved with the current example.

4.7 RELATED WORK AND DISCUSSION

4.7.1 RELATED WORK

Such high overhead inherent in CFI technique has led researchers to find more viable solutions for CRA defense [67, 78]. Their basic strategy to reduce the overhead is

relaxing CFI rules so that the computation cost required for the brute-force CFI enforcement on every executed branch can be substantially curtailed. For instance, the work in [78] uses approximated CFI rules that allow control transfer of returns to an instruction that follows a function call and indirect calls/jumps to the entry of functions. CCFIR [67] collects all control flow targets for indirect branches and allocates them in a specially aligned memory region, called the *springboard* section. At runtime, to enforce CFI, it checks whether each indirect control transfer goes through the springboard or not, by using one-bit test instruction thanks to the aligned springboard section.

Another line of research focused on finding the distinguishing attack signatures of CRAs exposed in runtime execution flow [38, 40, 39]. For example, DROP [39] uses as signatures the number of consecutive returns and instruction count residing in between the two adjacent returns. Even though their security analysis shows that DROP can detect various attacks in the wild, it suffers from high performance overhead, which ranges roughly from 2 to 20 times, due mostly to its validity checking code placed at every control transfer instruction. To lessen the performance overhead, more recent heuristic-based approaches [38, 40] focused on reducing the number of indirect branches for inspection. More specifically, they suggested applying detection heuristics only on the branch traces bound to several crucial kernel events that CRAs are likely to exploit. ROPecker [40] presents a similar mechanism but with a new concept, called the *sliding window*. By configuring only a set of recently accessed code regions (window) as executable while setting others to be non-executable, they tried to trigger the CRA inspection routine whenever a page fault, as a kernel event, occurs, for the purpose of utilizing the natural tendency of CRAs to employ gadgets across multiple pages.

Despite the performance gains, these relaxed CFI enforcement or event-driven detection heuristics may be inaccurate, since being deceived by advanced CRAs [79] that use gadgets not bound to any kernel event. In [71, 70, 74], they present possible attack

scenarios where the attackers employ a special kind of gadgets that can either bypass the relaxed CFI rules or erase the attack trails just before a kernel event. Contrary to these techniques, our hardware-based solution neither relies on any kernel event for performance gain as it incessantly monitors every branch trace emitted from PTI, nor employs relaxed CFI rules as it adopts in hardware the shadow stack, which is regarded as the most stringent way of enforcing CFI rules regarding the function invocation. As to our heuristic-based approach, although we instituted simple heuristic algorithms on our solution, our approach is not bound to a specific control flow monitoring scheme. Rather, our proposed framework can be coupled with other types of CRA solutions as the inspection phase is implemented as software. By delegating time consuming branch monitoring to our hardware logics, administrators will be able to implement more complicated algorithms empowered by delicate analysis techniques that can help determine whether the system is under CRAs with reasonable performance overhead.

There has been a number of research in [6, 41, 80] that proposes low-overhead hardware CRA solutions. All these solutions in common have installed special hardware logics inside the CPU for the shadow stack so as to provide a perfect protection against ROP attacks. In [6, 41], they have made further steps to handle JOP attacks by customizing their hardware for the additional logics to inspect the indirect calls and jumps that might be exploited by JOP. The problem of these hardware implementations is that, because the detection logic is implemented in hardware and fully integrated into the internal CPU architecture, their functionalities for CRA inspection are permanently fixed or at least limited unless they redesign their hardware along with their hosting CPU internals. In contrast, since our CRA monitor is a hardware device externally connected to the host CPU as one of the peripherals in the system, it in principle can be developed separately and attached to a commodity CPU like an ARM processor.

Recently, more elaborated attacks [75, 81, 82, 76] are incessantly crafted to outwit the existing defense techniques. The *call oriented programming* attack proposed

in [75] is one of such examples, which exploits gadgets that start with function entries, and therefore does not violate the CFI rule that the target address of a call should land on a function entry. Consequently, such an advanced attack can bypass the branch regulation scheme. In [81, 82], another type of advanced CRA, which we refer to as the *function reuse attack*, is introduced. Since this attack is invoked only using the existing C++ virtual functions in a program, the simple rule-based CFI defenses such as branch regulation cannot capture the attack. In another notable work [76], the authors propose the *control bending attack*, which *bends* a control edge in a way that the modified one is still found within the CFG. Although it can be detected by the shadow stack in our implementation, the function call invocations in the control bending attack are not regarded as illegal ones by the rules defined in the branch regulation schemes.

4.7.2 DISCUSSION

As previously discussed, the CFI techniques used in this study (especially the current implementation of a mixed CRA solution) may be bypassed by recently emerging novel attacks. In fact, the main contribution of this study is to implement a branch tracing framework, which can serve as a basis for CRA solutions to monitor continuously inside the target CPU without any modification to the CPU. This can be accomplished by utilizing the existing debug interface such as that of ARM PTM. Although currently implemented CFI techniques are somewhat simple, we believe that our framework is generic enough to implement more sophisticated CFI solutions that require the runtime monitoring of branch traces (that is, the control flow). As an example of such advanced CFI technique, PathArmor [65] has recently been proposed. This elaborated CFI techniques against CRAs try to enforce the integrity of both backward and forward edges on Intel x86 processors. During the execution of the victim application, PathArmor determines whether the control flow of the victim application can be identified within the pre-computed CFG. Our framework can provide every functionality required to implement this technique. The functionalities in PathArmor can be largely

divided into path monitoring and path verification. In our framework, the path verification part can be implemented in software to realize the algorithm used in PathArmor. This is because the software can access the code execution history stored in the BHB module. The path monitoring part can be mapped to the hardware function (i.e.,BBA). With the help of the customized hardware, the algorithm efficiently track control transfers at runtime. Note that PathArmor exploited the Intel's *last branch record* (LBR) to minimize the performance burden because of continuous control path tracking. However, Intel's LBR has the constraint of recording only a limited number (typically 16) of branch traces. The authors of PathArmor also admitted that their algorithm may suffer from the detection incapability because of an insufficient number of branch traces supported by LBR. On the contrary, ARM PTM can trace a nearly unlimited number of branch executions with negligible performance overhead. Thanks to the tracing feature of ARM PTM, our hardware module (i.e., BBA and BHB) can record more number of branch executions than LBR. Consequently, we believe the software part of our mixed CRA defense mechanism can not only just perform the CRA inspection proposed in PathArmor, but also be used to improve the technique a little (enforcing a more fine-grained CFI) by taking advantage of the fast branch tracing hardware.

In our mixed CRA detection approach, the most essential part is the BBA hardware. This is because implemented external CFI solutions must at least be provided with a mechanism to monitor the control flow of the currently running application. With the BBA module as a basic and the most primitive building block, the remaining parts can be partitioned in other ways. In other words, one can add different hardware logics (or modules) to the design so that all the hardware modules collectively operate in a way that can attain higher level of security. For example, a special hardware module such as a shadow stack can be placed to enforce CFI for the backward edges more securely and efficiently. In this case, users or software developers can deploy a more sophisticated software algorithm that focuses on forward edges with fewer performance penalties. However, when the CFI enforcement on backward edges

fully relies on the hardwired logic, the developers will be forced to use the employed algorithm in the hardware logic.

All in all, the more hardware logics are deployed for our CFI solution, the faster the entire system will be, but the less freedom is left to the developers for implementing their own algorithms. Hence the design choice should be made in advance in order to meet the requirements of the designed system.

4.8 CONCLUSION

We have discussed how our hardware-based CRA solution has been implemented and integrated into an ARM-based SoC. Our solution incurs reasonably low performance overhead for runtime detection of CRAs by implementing the unified hardware monitor. As our solution mainly monitors the execution traces coming out of the debug interface of the host CPU, it does not require any modification to the host processor internals and thus may be able to be easily integrated with a commodity ARM processor core, still complying with the conventional SoC design rules. Moreover, our solution can drastically reduce the storage requirement compared to that of the previous work. The experiments revealed that our current implementation successfully detects synthetic ROP/JOP attacks, and that the storage overhead incurred by our solution is acceptably small when being compared to the previous work.

Additionally, we propose a hardware/software mixed scheme. This approach attempts to catch the first indication of attacks by continuously monitoring all branch traces and discriminating (suspectedly) malignant branch operations from most benign ones. To efficiently perform the monitoring task without performance loss, we implemented in hardware a simple branch monitoring logic, and plugged it into the host CPU externally as a peripheral device. Upon finding any suspicious branch behaviors, our monitor notifies the OS kernel of the event so that the second main stage task of performing the in-depth control flow inspection can be performed on the ker-

nel. The evaluation performed on a real ARM-based FPGA system evinces that our solution was able to detect synthetic CRA attacks we deployed on the system. We also presented that our monitor can successfully identify and filter out most benign branch behaviors from the incoming traces, hence leaving to the inspection stage only a little faction of the branch information originating from the host. According to our evaluation with the default parameter settings for the alarming conditions, less than 0.01% of the information was transferred to the host kernel for in-depth inspection, which is low enough to substantially reduce performance overhead on the host machine.

While this work only covers the two cases of CRA defenses, our approach is not restricted to the use for specific CFI techniques. Rather, our proposed framework utilizing the existing debug interface can be applied to implement the other types of CFI enforcement techniques. Moreover, even though we are focusing on the branch tracing capability of ARM debug interfaces in this work, it also provides many other piece of information related to processor state transitions. Motivated by this feature, we plan to explore more applications where the information extract from ARM PTM can be fully utilized.

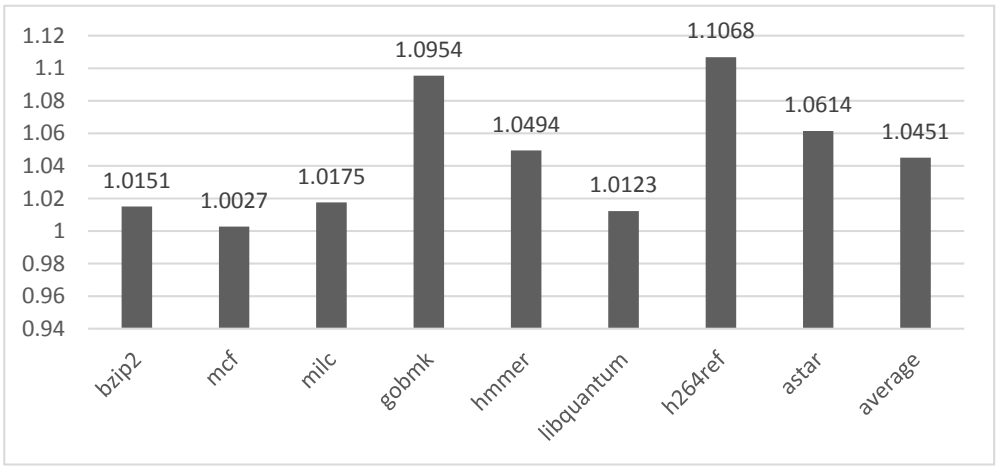


Figure 4.9: Benchmark execution time when the CRA monitor is enabled

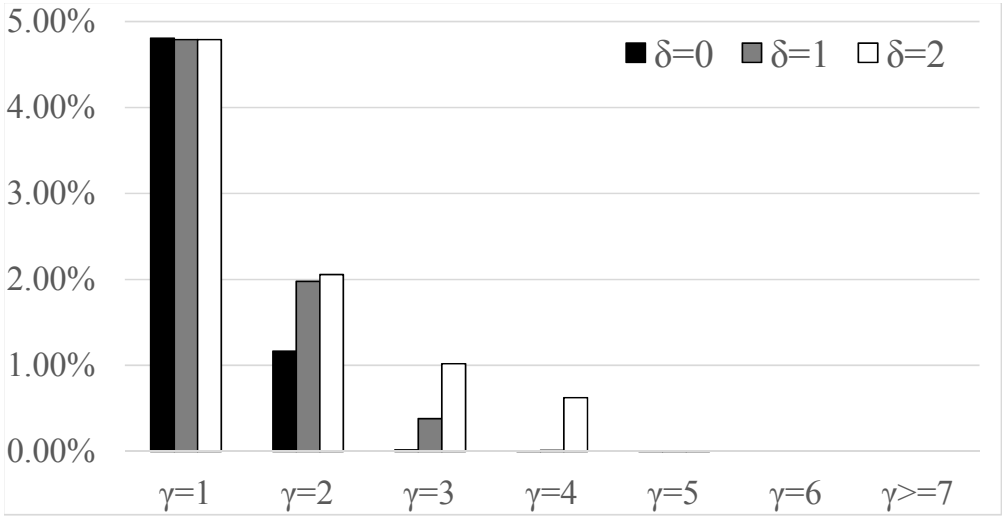


Figure 4.10: Inspection engagement rates as γ and δ change. When more than γ indirect branches arrive in the incoming traces and at most δ direct branches appear between two adjacent indirect branches, the traces are transferred to the inspection stage for further analysis. (The y axis indicates the percentage of branches that are transferred to the inspection stage for in-depth analysis, whereas the x axis refers to the configuration of the γ and δ values.)

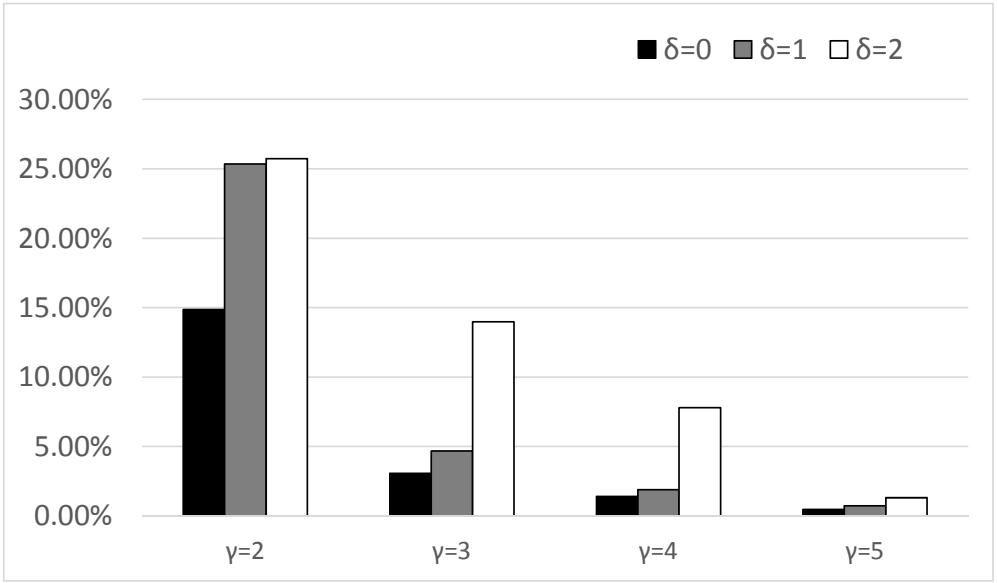


Figure 4.11: Performance overheads as parameters change (The y axis indicates the performance overhead for each configuration incurred on the underlying processor, whereas the x axis refers to the configuration of the γ and δ values.)

Chapter 5

Conclusion

This thesis presents a hardware-assisted SoC-level mechanism to enhance the security of ARM-based SoC. In our approach, an anomalous behavior detector is connected to the ARM host processor via a ARM CoreSight debug interface so that the detector can get the handful information of the host CPU. Our solution inherits the advantage of recently proposed hardware-based solutions in that the computation-intensive tasks are offloaded to the specialized external hardware engine, thus reducing the burden of the host processor introduced by the detection tasks. In addition, our solution provides the mechanism through which the detector can get the CPU execution information such as the control-flow and/or data-flow of the target program running on the host in a real-time manner. This functionality plays as a key factor to design various security applications that can attain both high level of security and low performance overhead, without necessity of changing the ARM processor's internal microarchitectures.

The experiment results in each section evince that, implemented and verified on a FPGA prototyping board, the solutions can successfully catch various attacks deployed on the system. Moreover, the performance estimation with a group of benchmark applications shows that our solutions do not suffer from severe performance loss. The experiment results also reveal that the area overhead of the hardware is acceptably small when compared to the normal sizes of today's ARM processors. We believe

that our solution can be applied to establish more wide range of security monitoring schemes, which need to observe the processor internal states to detect anomalous behaviors induced by attacks.

Bibliography

[1] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a practical solution to detect code reuse attacks on arm mobile devices," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2015, p. 3.

[2] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 2005, pp. 178–183.

[3] A. K. Kanuparthi, M. Zahran, and R. Karri, "Architecture support for dynamic integrity checking," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 321–332, 2012.

[4] S. Das, W. Zhang, and Y. Liu, "Reconfigurable dynamic trusted platform module for control flow checking," in *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*. IEEE, 2014, pp. 166–171.

[5] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Integration of rop/jop monitoring ips in an arm-based soc," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE '16. San Jose, CA, USA: EDA Consortium, 2016, pp. 331–336. [Online]. Available: http://dl.acm.org/citation.cfm?id=2971808.2971884

[6] M. Kayaalp *et al.*, "Branch regulation: Low-overhead protection from code reuse attacks," in *Computer Architecture (ISCA), International Symposium on*, June 2012, pp. 94–105.

[7] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," in *High Performance Computer Architecture, 2013 IEEE 19th International Symposium on*, Feb 2013, pp. 258–269.

[8] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-assisted flow integrity extension," in *Proceedings of the The 52nd Annual Design Automation Conference on Design Automation Conference*, June 2015, pp. 1–6.

[9] H. Kannan *et al.*, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *DSN '09.*, 2009.

[10] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object," in *USENIX conference on Security*. USENIX Association, 2013.

[11] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 28–37.

[12] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "Cpu transparent protection of os kernel and hypervisor integrity with programmable dram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 392–403.

[13] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor." in *USENIX Security Symposium*, 2004, pp. 179–194.

[14] S. Chen *et al.*, "Flexible hardware acceleration for instruction-grain program monitoring," ser. ISCA '08, 2008.

[15] V. Nagarajan *et al.*, "Dynamic information flow tracking on multicores," 2008.

[16] ARM co., LTD, "ARM CoreSight Architecture Specification v2.0," 2013.

[17] J. Newsome *et al.*, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[18] W. Cheng *et al.*, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *ISCC '06*, 2006.

[19] F. Qin *et al.*, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *2006. MICRO-39.*, 2006.

[20] G. E. Suh *et al.*, "Secure program execution via dynamic information flow tracking," ser. ASPLOS XI. ACM, 2004.

[21] M. Dalton *et al.*, "Raksha: a flexible information flow architecture for software security," ser. ISCA '07. ACM, 2007.

[22] N. Nethercote *et al.*, "Valgrind: a framework for heavyweight dynamic binary instrumentation," ser. PLDI '07. ACM, 2007.

[23] A. V. Fidalgo *et al.*, "Real-time fault injection using enhanced on-chip debug infrastructures," *Microprocessors and Microsystems*, vol. 35, no. 4, pp. 441–452, 2011.

[24] M. Portela-García *et al.*, "On the use of embedded debug features for permanent and transient fault resilience in microprocessors," *Microprocessors and Microsystems*, vol. 36, no. 5, pp. 334–343, 2012.

[25] G. Venkataramani *et al.*, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *HPCA*, 2008.

[26] "Leon3 processor user's manual, gaisler research," 2004.

[27] *Embedded Trace Macrocell Architecture Specification*, ARM, 2011.

[28] *AMBA Specification*, ARM, 1999.

[29] D. C. U. Guide, "Version c-2009.06," *Synopsys.(a)(b)(c)*, 2009.

[30] S.-W. Olle *et al.*, "Evaluation of the energy efficiency of arm based processors for cloud infrastructure," *Turku Centre for Computer Science*, 2010.

[31] M. Graphics, "Modelsim," 2007.

[32] K. Lab, "Dorifel Malware Encrypts Files, Steals Financial Data, May Be Related to Zeus or Citadel," 2012. [Online]. Available: http://threatpost.com/

[33] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, 2001.

[34] S. Vogl *et al.*, "Dynamic hooks: hiding control flow changes within non-control data," in *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX, 2014, pp. 813–828.

[35] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 103–115.

[36] L. Szekeres *et al.*, "Sok: Eternal war in memory," in *Secury and Privacy (SP), 2013 IEEE Symposium on*. IEEE, June 2013, pp. 48–62.

[37] E. SOGETI, "R&D Lab," *Analysis of the jailbreakme v3 font exploit*, 2012.

[38] V. Pappas *et al.*, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22Nd USENIX Conference on Security*. USENIX Association, August 2013, pp. 447–462.

[39] P. Chen *et al.*, "DROP: Detecting return-oriented programming malicious code," in *Information Systems Security*. Springer, 2009, pp. 163–177.

[40] Y. Cheng *et al.*, "ROPecker: A generic and practical approach for defending against rop attacks," in *Symposium on Network and Distributed System Security (NDSS)*, Feb 2014.

[41] L. Davi *et al.*, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, June 2014, pp. 1–6.

[42] Samsung Electronics co., LTD, "Exynos," 2015. [Online]. Available: http://www.samsung.com/global/business/semiconductor/product/application/overview

[43] Qualcomm Technologies, Inc., "Snapdragon," 2015. [Online]. Available: https://www.qualcomm.com/products/snapdragon

[44] NVIDIA Corporation, "Tegra," 2015. [Online]. Available: http://www.nvidia.com/object/tegra.html

[45] H. Özdoganoglu, T. Vijaykumar, C. E. Brodley, B. Kuperman, A. Jalote *et al.*, "Smashguard: A hardware solution to prevent security attacks on the function return address," *Computers, IEEE Transactions on*, vol. 55, no. 10, pp. 1271–1285, 2006.

[46] L. Davi *et al.*, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, March 2011, pp. 40–51.

[47] A. V. Fidalgo, M. G. Gericota, G. R. Alves, and J. M. Ferreira, "Real-time fault injection using enhanced on-chip debug infrastructures," *Microprocessors and Microsystems*, vol. 35, no. 4, pp. 441–452, 2011.

[48] M. Portela-García, M. Grosso, M. Gallardo-Campos, M. S. Reorda, L. Entrena, M. García-Valderas, and C. López-Ongil, "On the use of embedded debug features for permanent and transient fault resilience in microprocessors," *Microprocessors and Microsystems*, vol. 36, no. 5, pp. 334–343, 2012.

[49] J. Lee *et al.*, "Extrax: security extension to extract cache resident information for snoop-based external monitors," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, March 2015, pp. 151–156.

[50] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[51] ARM co., LTD, "ARM System Memory Management Unit Architecture Specification," 2013.

[52] PaX Team, "Address Space Layout Randomization," 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[53] T. Newsham, "Format string attacks," 2000.

[54] ARM co., LTD, "Cortex-A9 Processor," 2014. [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php

[55] V. Gaikar *et al.*, "IPhone 4S officially announced by Apple," 2011.

[56] Samsung Electronics co., LTD, "Exynos," 2015. [Online]. Available: http://www.samsung.com/global/business/semiconductor/product/

[57] ARM co., LTD, "AMBA Network Interconnect (NIC-301) Technical Reference Manual," 2014. [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php

[58] ——, "CoreSight Program Flow Trace Architecture Specification," 2011.

[59] ——, "Procedure call standard for the arm architecture," 2012. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042e/index.html

[60] The shell storm linux shellcode repository, 2014. [Online]. Available: http://www.shell-storm.org

[61] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[62] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm." in *USENIX Security Symposium*, 2014, pp. 941–955.

[63] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[64] Z. Guo, R. Bhakta, and I. G. Harris, "Control-flow checking for intrusion detection via a real-time debug interface," in *Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on*. IEEE, 2014, pp. 87–92.

[65] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 927–940.

[66] Features, ZC702 Board, "ZC702 evaluation board features," *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 Extensible Processing Platform*, 2012.

[67] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[68] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Mocfi: A framework to mitigate control-flow attacks on smartphones." in *NDSS*, 2012.

[69] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[70] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," 2014.

[71] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of USENIX Security*, 2014.

[72] T. Bletsch *et al.*, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, March 2011, pp. 30–40.

[73] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[74] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. of the 23rd USENIX Security Symposium (August 2014)*, 2014.

[75] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 575–589.

[76] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity." in *USENIX Security Symposium*, 2015, pp. 161–176.

[77] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[78] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *USENIX Security*, 2013, pp. 337–352.

[79] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in *USENIX Security Symposium*, 2011.

[80] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient dynamic information flow tracking on a processor with core debug interface," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 79.

[81] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 745–762.

[82] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained con-

trol flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 901–913.

초 록

최근 스마트 임베디드 기기의 보안 및 개인 정보 보호가 점차 중요한 문제로 부각되고 있다. 공격자는 여러 기기들에서 실행되는 프로그램의 노출된 취약점을 악용하여 시스템 동작을 제어 할 수있는 권한을 획득하려고 시도하는 것이 일반적이다. 결과적으로 타깃이 되는 프로그램은 제어 흐름의 조작과 같은 비정상적인 행위를 보이게 된다. 현재 실행중인 프로그램의 비정상적인 동작을 탐지하는 일반적인 방법은 런타임 실행 흐름을 모니터링하고 모니터링 된 흐름이 사전 정의 된 규칙과 비교했을 때 합법적인지 여부를 확인하는 것이다. 따라서 대상 기기를 비정상적으로 조작하려는 순간에 즉시 공격을 탐지하기 위해서는 프로그램 동작을 나타내는 엄청난 양의 CPU 실행 정보가 필요하게 된다. 이러한 이유로 인해 실행 시간에 CPU 실행 정보를 모으고 실행 된 프로그램에 대한 공격을 탐지하기위한 입력으로 수집 된 정보를 탐지 알고리즘에 신속하게 전달하는 메커니즘을 제공해야 할 필요성이 발생한다. 기존 많은 연구자들은 시스템에 끼치는 성능 오버 헤드를 최소화하면서도 높은 수준의 보안을 달성 할 수있는 보안 솔루션을 제안하여 문제를 해결하기 위해 노력해 왔다. 하지만 이러한 메커니즘이 거의 시장에 받아 들여지지 않은 실정이다. 예를 들어 이 메커니즘이 소프트웨어로 구현된다면, 분명히 호스트 CPU에 너무 많은 성능 부담을 부과하게 되고, 존재하는 하드웨어 솔루션 또한 호스트 아키텍처 내부의 무시할 수없는 수정을 초래하므로 설계 시간 및 제조 비용을 크게 증가시키는 요소가 되며, ARM 혹은 인텔 같은 상용 프로세서는 내부 수정이 불가능한 경우가 대부분이다.

이 논문은 앞서 언급한 문제들을 해결하기 위한 스마트 기기 상에서의 효율적인

비정상 행동 탐지 기법을 제안한다. ARM은 수년간 모바일 CPU 시장에서 사실상의 표준의 위치에 있었기 때문에 ARM CPU를 호스트 CPU로 선택하였다. CPU 실행 정보를 수집하기 위해 Real-time 디버깅 및 소프트웨어 추적을 위해 최근 프로세서에 널리 배포 된 ARM CoreSight 디버그 인터페이스를 활용한다. 디버그 인터페이스를 사용하면 CPU 내부 구조의 변경이 없이도, 성능 제약이있는 장치에서도 허용 가능한 낮은 성능 부하로 공격 탐지 작업을 수행할 수 있는 하드웨어 지원 SoC 수준 메커니즘의 설계가 가능하다. 우리의 접근법의 타당성을 보여주고 비정상적인 동작 탐지를 위해 ARM 디버그 인터페이스의 사용처를 탐색하기 위해 먼저 잘 알려진 보안 문제, 즉 데이터 유출 및 코드 재사용 공격을 처리하는 보안 모니터링 시스템을 제시한다. 또한 사용자가 ARM 디버그 인터페이스를 사용하여 자체적인 방어기법을 코어 상에 설계할 수 있는 HW / SW 혼합 접근 방식을 제시한다. 실험 결과에 따르면 오늘날 모바일 프로세서의 일반적인 크기와 비교할 때 하드웨어의 영역 오버 헤드가 허용 가능한 수준으로, 효율적인 비정상 행위 탐지가 가능함을 볼 수 있었다.

주요어: 정보 보안, 하드웨어 기반 이상행위 탐지, 디버그 인터페이스, ARM, CoreSight

학번: 2013-30249