Integration of ROP/JOP Monitoring IPs in an ARM-based SoC

Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang and Yunheung Paek
Department of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC)
Seoul National University, Seoul, Korea
Email:{yjlee, jylee, igheo, dihwang, ypaek}@sor.snu.ac.kr

***Abstract*—Code reuse attack (CRA) is a powerful technique that allows attackers to perform arbitrary computation by reusing the existing code fragments. To defend from CRAs while complying with the conventional ARM-based SoC design principles, the previous hardware solution suggests the use of the ARM debug interface to acquire the control flow information of an application running on the host. However, it requires tremendous storage space to store the complementary data necessary to trace the execution flow. In this paper, we propose a new hardware CRA monitor which gives both low storage overhead and high performance. For this, we have used an instrumentation technique which transforms the original ARM binary code into a form which will ease the CRA monitor to efficiently extract through the debug interface all crucial pieces of runtime information from the trace outcomes. In addition, while the previous solution was only built to detect one type of CRAs, called return-oriented programming (ROP), ours has been designed to unify the detection logics for ROP and another important type of CRAs, called jump-oriented programming (JOP). Empirical results show that our solution dramatically reduces the storage overhead for CRA detection, yet successfully detecting both ROP and JOP attacks simultaneously with negligibly low runtime overhead and moderate area overhead.**

I. Introduction and Previous Work

As smart mobile devices become our main devices for everyday communication, they are becoming more appealing targets of numerous software-oriented attacks. Among them, the *code reuse attack* (CRA) is a recently introduced technique that collects from the existing code blocks a set of small code sequences called *gadgets*, and chains them to perform malicious actions. Doing so empowers an adversary to perform Turing-complete computation without any attacker injected code [1], thus successfully defeating the well-known and widely adopted technique, generally called the W$\oplus$X (Write XOR eXecute) protection [2].

As the CRA threat is continuously escalating, many solutions have been proposed [2]–[6]. 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 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 [3], [4]. On the other hand, hardware solutions [2], [5]–[8] tend to exhibit high performance by accelerating the CRA detection process with the assistance of customized hardware logics for this task. In specific, authors in [2], [5], [6] proposed solutions where the hardware logics are tightly coupled with the host CPU for close monitoring of every control transfer during code execution. Despite their dramatic performance enhancement, the main drawback of these approaches is that they require the redesign of the existing processor architecture, which would stymie the direct deployment of these solutions into commercial smart mobile devices. The reason is that such modification to the core internal is contradictory to the common design practice for a smart mobile device in industry today. As the central computing platform for applications running on the device, an *application processor* (AP) in the form of SoC lies 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 now to integrate commodity processors and supporting IPs (intellectual properties) for specific functions together. Thus, if the AP vendors adopt some of these hardware solutions for their products, they will be compelled to restructure the CPU core architectures, contrary to the general convention, thus resulting in tremendous cost for design and verification.

To facilitate the acceptance of hardware solutions for the CRA detection in todays smart mobile devices, some latest approaches [7], [8] endeavor to comply with the design rule of SoC. Their security hardware IPs are practical solutions for CRAs in a sense that they do not require any internal modifications to current host architectures but simply external connections with the host processor to build an SoC. The biggest challenge of the approaches however is that, being located outside the host processor, their hardware IPs are usually difficult to acquire the correct control flow information of the applications running inside the host, which is essential to monitor the existence of CRAs. In order to tackle this challenge, they exploit the built-in debug features to reveal the runtime information of the host to the outside of the core. Especially, the work by Lee et al. [8] implements a CRA monitoring hardware using the debug features supported in commercial ARM processors, which are the de-facto standard CPUs for mobile SoCs today. To provide the efficient and convenient debug/trace environment to software developers, virtually all ARM processors including Cortex-A8, A9 and A15 embed the ARM CoreSight debug architecture [9]. The CoreSight architecture provides features for continuous collection of the processor execution traces using the hardware trace unit. Utilizing this unit, the hardware IPs proposed in the work can obtain the real-time traces of branch outcomes produced during code execution.

Although these approaches using the tracing hardware could achieve high performance in CRA detection, they are facing another challenging problem. In principle, in the debug environment using the hardware interface like CoreSight, it is assumed that the debugger has 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 the information which could be inferred or simply extracted from the binary code. However, unfortunately, these omitted pieces of information such as branch types or source addresses for branch instructions are indispensable for accurate CRA monitoring. To supplement the lacking information, in previous work, they store in the main memory region the auxiliary information, called the *meta-data*, that is necessary for CRA detection, and make the hardware IPs to read the data at runtime when the detection scheme needs to reference the data. In spite of the negligible performance overhead, they severely suffer from the substantial storage overhead due to the additional space for their meta-data. According to their experiments, the size of the required storage for meta-data can even be double that of the original application. Another limitation of their hardware implementations is that they are only capable of detecting the *return-oriented programming* (ROP) attacks which are to corrupt return addresses stored in a stack to chain gadgets. Although ROP attacks are representative examples of CRAs, there is another breed of CRAs, called *jump-oriented programming* (JOP) attacks, whose objective is to alter the target addresses of indirect calls or jumps. To successfully defend the system against CRAs, therefore, the CRA monitoring hardware should be implemented with mechanisms that can detect not only ROP but also JOP attacks.

In this paper, we present a hardware-based CRA solution that can simultaneously monitor both ROP and JOP attacks on the system. For applicability of our solution to existing smart devices, we have built a *unified* ROP/JOP monitor that is integrated as IPs into an ARM-based SoC. As in previous work [8], the monitor is connected with the ARM CPU via the CoreSight interface and system bus to keep track of the host execution traces from outside in a timely fashion. In addition, for efficient monitoring, we have also made an effort to avoid substantial storage overhead due to meta-data in the previous work. 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 via the debug interface, thereby eliminating the need to store meta-data a priori for our monitor. However, a problem with this approach is that the two independent interfaces (i.e., the debug interface and the system bus) through which our external monitoring IPs receive host's runtime information are not perfectly synchronized; that is, when at some point in the code the CPU executes an instruction, proper pieces of the information for that execution will be generated and eventually transferred to our monitor through each interface, but not necessarily at the same time. Obviously, our monitor must correctly puzzle together these information pieces asynchronously arriving from two different sources to grasp the exact execution behaviors on the host for CRA detection. To resolve this issue, we added a special hardware logic to synchronize the incoming information from the two sources.

II. Assumptions and Threat Model

We use the same assumptions on CRA taken by previous studies [2], [3]. We first assume that the target mobile device is under the protection of the W$\oplus$X policy and the OS is trusted. Considering that the modern OSes and processors usually cooperate to enforce the W$\oplus$X security protection rule [10], we believe this assumption is reasonable. Under this assumption, to circumvent the defense mechanism, the adversaries must gain sufficient privileges for the first time. We assume that, other than CRAs, there are no other attack vectors or security holes which can directly escalate adversarys privilege. As another assumption, adversaries might have full control over the stack or heap to exploit common memory vulnerabilities like buffer overflows and therefore can initiate a code-reuse attack. Also, the OS kernel and hardware are trusted until the underlying system is compromised through CRAs. We also assume that adversaries know all implementation details of the target application, thus being able to locate the exact address of available gadgets. This means that the adversary can bypass any code randomization techniques such as *address space layout randomization* (ASLR) [11]. Lastly, the self-modifying code is not considered in our assumptions because it conflicts with the W$\oplus$X security protection.

III. Overall System Architecture

A. SoC Prototype Overview

Figure 1 depicts our overall SoC design. The monitoring modules for ROP/JOP detection were designed and implemented as a subsystem, called the *CRA monitor*, which is then integrated in a SoC platform with an ARM CPU. In our platform, the host CPU is an ARM Cortex-A9 processor, which has been installed in a large number of commercial devices these days. The host CPU and our monitor are connected via the standard AMBA3 AXI interconnect. To obtain the results of branch operations performed on the host, we utilize the built-in hardware modules of the ARM CoreSight debug architecture, which are the *program trace macrocell* (PTM) and the *trace port interface unit* (TPIU). Being tightly coupled with the host core, the two modules deliver the branch traces generated from the host to the CRA monitor. 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 smart computing devices. 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. As shown in the figure, our CRA monitor is divided into two modules: the *PTM Trace Analyzer* (PTA) and the *CRA detector*. To reduce the amount of transferred data, TPIU basically provides runtime traces in a highly compressed form. Thus, PTA analyzes and decompresses the incoming information from TPIU, and delivers to the CRA detector the refined branch traces which are necessary for the CRA detection. Upon receiving all the traces from PTA, the detector determines whether or not the traces exhibit any symptom of CRAs. In Section IV, more details of the hardware modules will be explained.

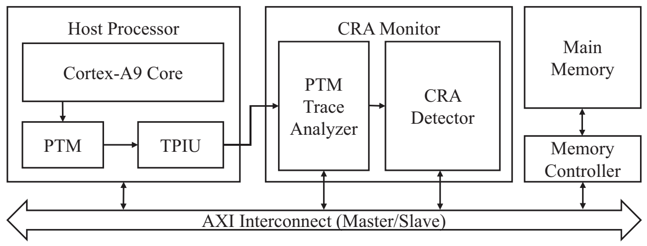

Fig. 1. Overall architecture of our SoC design

B. CRA Detection Process

As stated in Section I, our CRA monitor detects both ROP and JOP attacks. To determine an attack from outside the host CPU, it must be provided with the necessary runtime information inside the CPU. In our work, to detect both types of attacks, we have realized in hardware the detection algorithms based on those proposed in [12] and [5], respectively.

For ROP, 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 programming 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. 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 jumps 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 IV. 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).

IV. IMPLEMENTATION DETAILS

A. Binary Instrumentation

As briefly discussed in Section III, we propose a binary instrumentation technique that enables us to derive from the branch traces of the host system more information including 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, which are executed by either a `bl` (branch with link) or a `blx` (branch with link and exchange) instruction in the ARM architecture. In order to deliver the information associated with the call instructions, we introduce a new code section called the *trampoline*. Each call instruction in the original code 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, each direct call (`bl` or `blx` with an immediate offset) moved to the trampoline is manipulated by the instrumentor so that it can target the same address as the original instruction pointing to. In addition, 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 Figure 2, we present an example to explain our instrumentation technique.

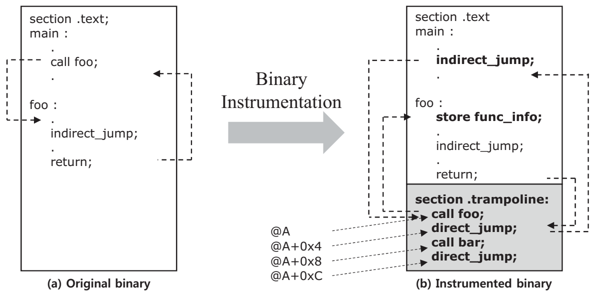

Fig. 2. Original vs. instrumented binary (newly added parts are written in boldface)

As shown in the Figure, 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 $0 \leq n < total\ number\ of\ calls$) . 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, when 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. It means that the source address of each call can now be obtained from TPIU because the target address of the indirect jump pointing to $A + 8 * n$ becomes the source address of the call, allowing our monitor to calculate the legitimate destinations of return instructions which are necessary to maintain the shadow stack for ROP detection. Also, to detect JOP attacks, the function boundary information is indispensable to check if 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 Figure 2(b)) which 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 executable and linkable format (ELF).

B. Hardware Architectures

Figure 3 shows the hardware structure of our CRA monitor including PTA. In our SoC prototype implementation, the output signals of TPIU are directly routed to the on-chip ports of our CRA monitor so that we can utilize the CoreSight modules. As the host CPU generally operates far faster than other hardware IPs such as our CRA monitor. Therefore, to transfer the PTM traces from the host to the monitor, we implement an asynchronous buffer, called the *branch trace FIFO*, which temporarily stores the traces coming from TPIU. When the traces are stored in the FIFO, another submodule in PTA called the *trace decoder* analyzes the saved 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 mentioned in the previous subsection. Finally, for each branch instruction, its type and associate information (i.e., source addresses for calls and target addresses for indirect branches) are conveyed to the CRA detector for monitoring CRAs.

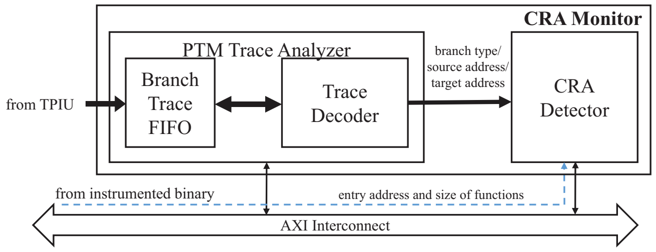

Fig. 3. CRA monitor hardware architecture

Figure 4 shows the unified hardware architecture of our CRA detector which keeps track of the host execution traces to simultaneously detect both ROP and JOP attacks. To find the existence of CRAs, our detector relies on the aforementioned branch information fed by PTA and the entry address and the size of functions coming through the system bus.

Recalling that the information from different sources (i.e., TPIU and the system bus) have no ordering restrictions, the CRA detector has to combine and rearrange the information from the two sources to keep track of the original program sequence of the application. In order to perform this task, the detector has two separate First-In-First-Out (FIFO) buffers, called the *PTM FIFO* and the *MMIO FIFO*, to temporarily store the information received respectively from PTA and the bus. The output signals of the FIFOs are given as input to the *trace combiner* (TC), which is in charge of combining the information from the two FIFOs and extracting the original program execution behaviors. We present the example of the information flow from the two sources and how they are combined by TC in Figure 5.

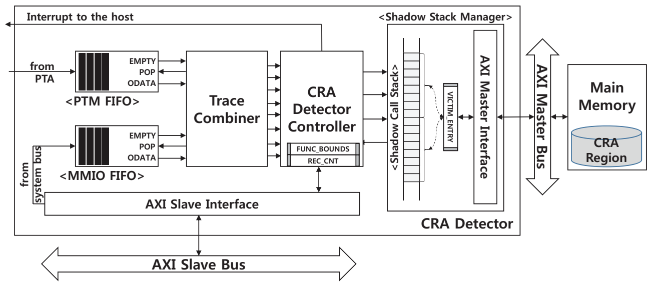

Fig. 4. Hardware architecture of the CRA detector

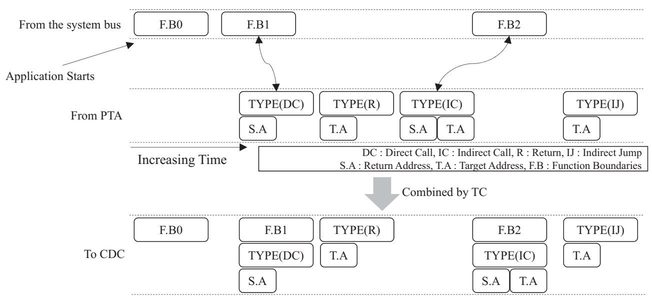

Fig. 5. Information flow diagram processed by the Trace Combiner

As exemplified in the figure, when the application begins, the program flow encounters the initially invoked function (i.e., main()) for the first time. This special event is notified to TC via the MMIO FIFO so that TC can start operation (presented in Figure 5 as F.B0). At runtime, when the program runs into a call instruction, the instrumented code at a function prologue is executed right after the call instruction, thus delivering the branch information (branch type and the associated information) and the function boundary information via 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 to take an appropriate action. If the function boundary information is written to the MMIO FIFO, TC waits for an event to come from the PTM FIFO. Once the PTM FIFO gets an entry, TC checks the branch type, and if it is either a direct or an indirect call, TC combines the pieces of information from the both FIFOs (i.e., the branch type, the function boundary information and the source address for a direct call as shown in Figure 5); otherwise, only the information from the PTM FIFO is selected. The information is then delivered to the *CRA detector controller* (CDC) whose mission is to make the final decision about the existence of CRAs.

After the application starts, CDC expects the information of 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 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 spontaneously notifies the host of this attack by setting the `interrupt` signal on.

Note that the shadow call stack has a finite number of entries, 16 in this work. Therefore, 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 SSM. Also, we implemented a register called `VICTIM_ENTRY` which plays a role as a victim cache storage to temporarily store the most recently evicted 8 entries. Moreover, there is an exceptional case that the host program calls the same function recursively. For handling this case, CDC has a counter register, which we refer to as `REC_CNT`, to store the number of recursive calls. When the same function is called in a row, CDC increase 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.

V. EXPERIMENTAL RESULTS

To evaluate our approach, we implemented a full SoC prototype on the Xilinx Zynq-7000 XC7Z020 evaluation board, which is equipped with a dual-core ARM Cortex-A9 processor, AMBA3 AXI interconnect, 1GB DDR3 SDRAM, an FPGA chip and other peripherals. We used Linaro Ubuntu Linux version 3.8.0 as our host kernel. Also, we enabled the CoreSight modules (i.e., PTM and TPIU) in the host processor and controlled them with the device driver which is extended according to our purpose. Our CRA monitor and the host CPU commonly operate at 60 MHz. Based on the above design parameters for the prototype, we synthesized the CRA monitor onto the FPGA chip and measured the required logic count in terms of lookup tables for logic (LUTs) and memory elements. The synthesis result shows that our CRA monitor occupies 10.12% (5,387/53,200) of total LUTs and 0.13% (24/17,400) of total memory elements.

To measure the detection capability of our monitor, we implemented five CRA instances based on the Shell-storm shellcode [13] as shown in Table I. Especially, A2 and A5 contain long-gadgets to bypass the signature-based CRA solutions proposed in [2], [3], which use the small number of instructions in a gadget as the distinctive feature of CRAs.

With the implemented attacks, we tested the detection capability of our monitor. As expected, all the ROP samples (A1-A3) are detected by our CRA monitor. Since they violate the general convention of the function invocation, their malicious behaviors are detected by our CRA monitor even when the attacks contain long-gadgets which is an advanced skill for circumventing the state-of-the-art CRA detection schemes. The JOP samples (A4-A5) are crafted by using `blx` (indirect call) or `bx` (indirect jump) instructions of ARM ISA to link their gadgets. In these attacks, every `blx` instruction used to link gadgets 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. Based on this result, we assert that our CRA monitor can protect the target system from any type of CRAs.

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 I. THE DESCRIPTION OF IMPLEMENTED CRAS AND DETECTION RESULTS OF THE ATTACKS

To measure the performance overhead of our CRA monitor, we chose eight applications from the SPEC CPU2006 benchmark suite [14]. 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 CRA monitor disabled, thus being exposed to CRA attacks. The other is *wCRA* that refers to the same code execution with the CRA monitor enabled. We show the performance numbers of *wCRA* in Figure 6 where the execution time of each application with *wCRA* is normalized to that of *Base*. The empirical results show the running time overhead of 4.51%/10.68% (average/max) over *Base*.

Also, we compared the storage overhead due to our instrumentation with the overhead incurred by the meta-data proposed in [8]. 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 [8] and ours, as presented in Table II. 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 [8] (on average 145.54%).

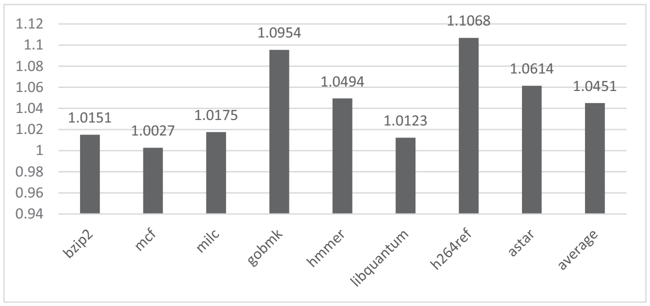

Fig. 6. Benchmark execution time when the CRA monitor is enabled

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 II. COMPARISON OF BINARY SIZES BETWEEN OURS AND [8]

The above results clearly show the advantage of our approach over the previous work [8] 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 [8] 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 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 [8] 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 III. 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 [8] 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 [15] whose frequency gap between the host CPU and other auxiliary IPs are typically configured up to 5:1.

VI. CONCLUSION

We have discussed how our hardware solution has been integrated into an ARM-based SoC to defend the system against ROP and JOP attacks at the same time. The solution incurs very low performance overhead for runtime detection of CRAs by implementing the unified hardware IPs to efficiently detect both types of attacks. Our solution does not require any modification in the host ARM processor internal. Therefore, our hardware modules can be easily integrated with a commodity ARM processor core, observing the conventional SoC design rules so that our solution can be easily implanted to commercial mobile SoCs. Moreover, our key contribution is that ours reduces the storage overhead dramatically compared to the previous work. To achieve this, we propose an instrumentation technique which enables us to make the most use of the existing debug interface. 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.

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

TABLE III. FREQUENCY GAP TOLERANCE OF OURS AND [8] (IP_CLK IS FOR BOTH THE MONITOR AND THE DDR MEMORY)

ACKNOWLEDGMENT

This work was partly supported by IITP grant funded by the MSIP (No. R0190-15-2010), the Brain Korea 21 Plus Project in 2015, the MSIP under the ITRC support program (IITP-2015-R0992-15-1006) supervised by the IITP, and the National Research Foundation of Korea(NRF) grant funded by the MSIP (No. 2014R1A2A1A10051792). The ICT at Seoul National University provides research facilities for this study.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

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

[12] 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.

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

[14] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

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