IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 14, 2019, accepted December 11, 2019, date of publication December 23, 2019,
date of current version December 31, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2961416

ActiMon: Unified JOP and ROP Detection With Active Function Lists on an SoC FPGA

HYUNYOUNG OH[1,2], MYONGHOON YANG[1,2], YEONGPIL CHO[3], AND YUNHEUNG PAEK[1,2], (Member, IEEE)

[1]Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, South Korea
[2]ISRC, Seoul National University, Seoul 08826, South Korea
[3]School of Software, Soongsil University, Seoul 06978, South Korea

Corresponding authors: Yeongpil Cho (ypcho@ssu.ac.kr) and Yunheung Paek (ypaek@snu.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evalution (IITP) Grant Funded by the Korean Government (MSIT) under Grant 2018-0-00230 (Development on Autonomous Trust Enhancement Technology of IoT Device and Study on Adaptive IoT Security Open Architecture based on Global Standardization [TrusThingz Project]) and Grant 2017-0-00213 (Development of Cyber Self Mutation Technologies for Proactive Cyber Defense), in part by the National Research Foundation of Korea (NRF) Grant Funded by the Korean Government (MSIT) under Grant NRF-2017R1A2A1A17069478 and Grant NRF-2018R1D1A1B07049870, in part by the BK21 Plus Project in 2019, and in part by the EDA tool from the IC Design Education Center (IDEC), South Korea.

ABSTRACT Field programmable gate arrays (FPGAs) have been increasingly mounted on commodity systems. As a matter of fact, such an emerging adoption of FPGAs in the commodity systems is attributed to their versatility came from the programmable property. Accordingly many industrial and academic attempts have been performed to exploit FPGAs in a variety of applications. In this paper, we note that FPGAs also can be used to protect the host CPU from a nasty security threat, called code reuse attacks (CRAs). Code reuse attack (CRA) is a powerful technique that allows attackers to execute arbitrary code. Control-flow integrity (CFI) has been popularly employed to mitigate CRAs. CFI entails CRA monitoring that checks if a program runs as directed by its control-flow graph. However, as monitoring naturally incurs non-negligible runtime overhead to the host CPU, many studies proposed hardware techniques to lessen the monitoring overhead. To facilitate the immediate deployment of a hardware-based solution, we propose a CRA monitor, called *ActiMon*, that can be implemented on an SoC FPGA where the host CPU and FPGA are manufactured together in a single platform. However, implementing the CRA monitor operating on FPGA arouses a new challenge that has never been addressed in the previous solutions: the operating clock of FPGA is many times slower than the CPU. By overcoming this speed difference, we ultimately purpose to evince the feasibility of FPGA as a computing device in the field of CRA defense. For this purpose, we have developed a highly efficient algorithm designed to run on FPGA whose goal is to monitor the existence of CRAs on the host CPU residing in the same SoC FPGA platform. Empirical results show that ActiMon runs on our target SoC FPGA platform efficiently enough to catch up to the speed of host code execution and promptly detects two important types of CRAs, JOP (Jump-Oriented Programming) and ROP (Return-Oriented Programming), as soon as they occurred in the host system. We assert that such results are encouraging thanks to our unified, lightweight ROP/JOP detection mechanism based on a list of *active functions*, and also to additional optimizations to leverage the inherent capabilities of FPGA for parallel computation.

INDEX TERMS Code reuse attacks (CRAs), control-flow integrity (CFI), external monitor, field programmable gate arrays (FPGAs), hardware-based security.

I. INTRODUCTION

Field programmable gate arrays (FPGAs) have been increasingly mounted on commodity systems. For example, Amazon has established FPGA-builtin servers to allow customers to use FPGAs in *Amazon Web Services* (AWS) [1]. In another example, Intel, a prominent commercial CPU vendor, has recently developed a *programmable acceleration card* (PAC) [2] which runs various operations on an FPGA connected via PCIe to the host CPU. It is noteworthy that FPGAs become more tightly coupled by being integrated on the same die with various CPUs such as Intel Xeon [3]

The associate editor coordinating the review of this manuscript and approving it for publication was Mohamed Elhoseny.

ARM [4], [5] and RISC-V [6]. As a matter of fact, such an emerging adoption of FPGAs in the commodity systems (i.e., hybrid CPU-FPGA architectures) is attributed to their versatility came from the programmable property. Accordingly, many industrial and academic attempts have been performed to exploit FPGAs in a variety of applications—data analytics [7], media processing [8], artificial intelligence [9], network security and monitoring [10], financial [11] and genomics [12].

In this paper, we note that FPGAs also can be used to protect the host CPU from a nasty security threat, called code reuse attacks (CRAs). Attackers frequently launch CRAs to hijack control-flow of the program execution so that they can force the victim program to execute the existing code snippets (i.e., gadgets) in a maliciously crafted order to seize sensitive information or cause unexpected actions like program interruption or data tampering.

Many techniques for mitigating CRAs have been devised over the past decade. Among them, *control-flow integrity* (CFI) [13] has been the most popular, whose goal is to ensure that a program is performed according to the control-flow intended by software developers. Unfortunately, enforcing CFI typically entails *CRA monitoring* [14], [15] whose runtime overhead should grow to a great extent. To lessen the monitoring overhead in enforcing CFI, security researchers have developed a solution that uses hardware support, particularly FPGAs. As a concrete example, the authors in [16] proposed an FPGA-based monitor to detect CRAs which have been launched inside the host CPU. To be brief, the solution extracts the runtime information of a program running on the host CPU via ARM's built-in external interface, and then delivers to the monitor implemented on the FPGA to check if the CFI is violated. Note that these FPGA-based solutions including that of [16] have built up on an assumption that monitors on the FPGA can run virtually as fast as the host CPU. However, such an assumption goes against the fact that in real systems there is a big gap between the CPU and FPGA in operating speed—according to our observations FPGAs merely run about eight times slower than the host CPU—due to the intrinsic nature of FPGA that has to sacrifice performance for programmability [17]. According to our preliminary experiment, in real systems where the CPU and FPGA shows significantly different operating speed, the implementation of [16] ran too slowly to process all crucial runtime information emitted from the CPU for CRA detection in a timely fashion, resultantly raising many false alarms and even failing to recognize the existence of several CRAs that were all detected in [16].

In this paper, we propose a new FPGA-based CRA monitor, *ActiMon*, that is deployable to real systems despite of the certain differential operating speed between the host CPU and FPGA. ActiMon aims to defeat two major CRA schemes, JOP [18] and ROP [19]. One way for ActiMon to tackle these types of attacks is to capture all branch traces taken in the host CPU and employ the well-known CFI enforcement techniques such as *branch regulation* (BR) [20] and *shadow stack* (SS) [21]. However, due to the intrinsic performance limitation of FPGA, the monitor on FPGA was not able to process all branch traces promptly as they are generated and transmitted from the CPU, and thus soon overwhelmed by the speed and volume of incoming branch traces.

Therefore, in order to avoid such a problem, we have engineered a CRA monitoring algorithm for ActiMon such that it can fulfill its monitoring task even on low-speed FPGA fast enough to process all branch traces emitted from the high-speed CPU. At the center of our algorithm, there is a notion, called the *active function*, which allows ActiMon to read a much less amount of runtime information from the CPU with little loss of CRA detection accuracy. We here say that a function is active if it was invoked but is not yet returned. In our algorithm, active functions are recorded in the *active function list* (AFL), which is updated every time functions are invoked and returned during program execution. ActiMon detects ROP attacks by enforcing the rule that a return is only allowed to target one of the active functions. To add JOP detection capability hereby, we have combined our ROP detection rule with the BR rule that the control jumps across a function boundary is only allowed to target the entry of another function. More specifically, in our algorithm with AFL, we classify such cross-function jumps that target the middle of functions into two classes depending on whether or not their target locations belong to active functions. Of the first class are the jumps that transfer the control flow into the middle of an inactive function. As regulated by BR, we declare that they are making illegal control transfers induced by CRAs. On the other hand, a jump that causes the control transfer to the middle of active functions can be either legal or illegal depending on the instruction type. If the jump is made not by a return instruction but by an ordinary indirect branch instruction, it will be judged illegal by our algorithm. However, we rule that the jump made indeed by a return is a legal, normal control transfer from the callee function to the caller.

As can be seen from our description just above, our JOP/ROP detection algorithm is deemed *unified* in a sense that it tackles JOP and ROP simultaneously with a single unified mechanism based on AFL. More accurately, it is basically the ROP defense scheme added with JOP detection capability where the AFL management of ActiMon is involved in the detection of both JOP and ROP. As evidenced experimentally, our unified, lighter AFL-based detection solution runs more efficiently, thus attaining higher performance in JOP/ROP detection than [16], heavier solution carrying out multiple schemes separately for JOP and ROP detection. As another optimization to maximize the inherent capabilities of FPGA specialized for parallel processing, we parallelize the AFL management by processing multiple cross-function jumps concurrently whenever possible.

In principle, as far as two jumps have all different call objects, return targets and return objects, the transactions to manage AFL for these jumps can be parallelized because no order dependency exists among the transactions in our scheme. The most typical, commonly-occurring case where they must be processed sequentially is when either the pair of call and return or that of return and return relates to the same function. In this case, each member of the pair must be processed in order one after the other by first updating AFL for the one jump and then verifying the other. To reason this, suppose that a call/return pair in the above case is processed in parallel. Then in our scheme, the return jump may raise a false attack alarm because the target function entry of AFL is not yet activated by the prior call jump. At runtime, upon receiving a group of cross-function jumps from the host CPU, ActiMon first performs the pair-wise comparison between them to determine if the AFL transactions for these jumps are parallelizable. We then divide the group of jumps into subgroups in each of which the AFL transactions for its members are carried out concurrently. Once the AFL transactions are complete for one subgroup, the next subgroup is chosen in order, obeying the dependence, and all its jumps belonging to the same subgroup are processed in parallel again. Fortunately, our observations have revealed that performance degradation due to such serialization of AFL transactions among subgroups is negligible because a majority of jumps encountered in reality are parallelizable.

To summarize, the AFL-based CRA defense mechanism has been an enabler of our unified ROP/JOP detection algorithm for ActiMon that is lightweight enough to successfully run on an FPGA platform today. Further performance optimizations on the platform were made possible by parallelizing AFL transactions for cross-functions jumps. Despite all these strengths in terms of performance, we also admit that ActiMon has a relative weakness in terms of security as compared with previous solutions based on separate detection algorithms specialized for ROP and JOP respectively. For instance, the algorithm based on SS can regulate the strictly ordered matching between a return address and its call site, whereas our AFL-based algorithm is more lenient or coarse-grained in a sense that it allows a return to target one of the executing functions. However, our experiment revealed that the monitoring algorithm of ActiMon is effective enough to detect CRAs by demonstrating all the real attack samples as implemented in previous work [16] are successfully detected. At the same time, the experiment with the SPEC CPU2006 [22] demonstrates that ActiMon is able to run on Zynq-7000 SoC FPGA with an acceptable performance overhead of 9.77% on average in comparison with native host code execution without CRA detection. This result we assert is encouraging in a sense that it exhibits a potential of immediate deployment of today's SoC FPGA (possibly including many other types of devices integrated with FPGA inside) armed with security functions for real uses in the field.

Our contributions are as follows:

- **Readily deployable FPGA-based CRA monitor.** ActiMon is a CRA monitor that can be actually deployed to the FPGA. Unlike previous work [16] not considering the significant clock speed difference between FPGA and CPU, ActiMon is workable even at up to an eight times slower clock.
- **Optimizations for an efficient CRA monitor.** To overcome the slower clock speed of FPGA, we invented the CRA monitoring algorithm based on AFL to read a much less amount of runtime information. In addition, we parallelized the AFL management processes to maximize the performance.
- **Low performance and area overhead.** In our experiments, ActiMon only incurs 9.77% performance overhead to the host CPU, while having a small binary size increment of 18.7% and 100k LUTs which is 45.75% usage in our Zynq-7000 SoC FPGA.

II. RELATED WORK

There have been diverse attempts to mitigate CRAs by enforcing CFI. Abadi et al. [13] introduced a software-based CFI solution to protect against CRA attacks that change the original control flow graph (CFG). The CFG is typically generated by statically analyzing the source code or the binary of a program. According to the CFG, a unique label is assigned between the indirect branch and the basic block that the control can move to. Then the instrumented code checks whether the label matches between the branch and the target basic block in runtime. While being a promising method in defending gains JOP and ROP, representative attacks of CRA, it still requires an expensive static analysis and incurs non-trivial performance overhead due to a software-based runtime check.

Coarse-grained CFI is a policy which does not strictly enforce the full CFG in order to improve the performance [14], [15], [23], [24] while minimizing the loss of security measure. They set a few equivalence classes of target addresses and verify that the target of the indirect branch matches one of the equivalence classes. For example, binCFI [24] allows a return to go to one of all the possible targets, and PICFI [15] allows a return to go to one of the functions called so far. In addition, regardless of CFI technique, DROP [14] monitors ROP attacks by intercepting a return and examining the length of its sequence based on the knowledge that the ROP often connects gadgets of short length repeatedly.

As another line of study, there are various CFI solutions augmented with hardware supports to enhance their performance [16], [25]–[38]. Their hardware modules are integrated into the CPU pipeline stage [25]–[31] or just placed within the CPU without significant changes in the CPU architecture [32]–[34] or combined with the outside of CPU in the form of SoCs [16], [35], [37], [38] to extract and analyze information needed to monitor CRA attacks. Since their hardware modules execute monitoring algorithms in parallel with the host CPU, the performance overhead is greatly decreased. Our ActiMon also operates in a parallel fashion similar to the previous hardware-based solutions. The first group that integrates the monitor inside of the CPU has a disadvantage where invasive modifications to the processor internals (e.g., registers and pipeline datapaths) are required. In fact, modern microprocessor development may take several years and hundreds of engineers from initial design to production [39]. Therefore, the substantial costs of development to integrate the customized logic would hamper processor vendors to adopt them, unless the necessity is clearly established. Although the second group that implements their monitor outside of the CPU can eliminate the redesign cost of the CPU, they still result in the tremendous cost of the semiconductor manufacturing for the integrated custom SoC. However, our solution is designed to be operated on commercial off-the-shelf (COTS) SoC FPGAs that can easily be acquired from the market. On top of this, ActiMon makes its notable difference in that, unlike previous hardware-based solutions, it does not require either high non-recurring cost or long manufacturing time.

ActiMon, we would say, is the direct descendant of the CRA monitor developed by Lee *et al.* [16] among the previous hardware-based solutions. Both ours and theirs are detecting CRAs from the devices located outside of the ARM host processor and connected via the built-in debug port. These monitors are designed to detect JOP and ROP together for comprehensive CFI enforcement. However, there is a profound difference between these two in that their CRA monitor is not designed for operating in FPGA while ActiMon is in FPGA. For example, their CRA monitor could operate under a circumstance where the speed ratio of CPU and theirs is only up to 4:1. In contrast, ActiMon is designed efficiently enough to handle the flood of data from CPU so that it can withstand that of 8:1, which is the real frequency discrepancy. To achieve such a notable efficiency, ActiMon also adopted an optimized binary instrumentation technique that extracts only the essential information, greatly reducing the total amount of PTM packets, so that ActiMon can perform the monitoring algorithm in a more lightweight manner than [16].

Efficient monitoring algorithms often used by hardware-based solutions to monitor CRA attacks are shadow stack [21] and branch regulation [20], defending against ROP and JOP respectively. A shadow stack stores an extra copy of the return addresses corresponding to each function call. ROP attacks, which modify the return address stored in the stack, are detected by comparing the return address with the reference value securely stored in the shadow stack. In branch regulation, JOP is mitigated by allowing for branching only to any function entry point or any point within the currently executing function. HAFIX [30] implements a coarse-grained CFI that only allows a return to target the currently executing active functions. ActiMon has borrowed this policy from HAFIX to monitor ROP. In fact, our AFL-based scheme is similar to their strategy that has the label state memory storing the activated state of each function. However, ours differs from theirs in several aspects, such as the unified detection algorithm for JOP, various optimizations targeting FPGA, and so on. ActiMon manages AFL with multiple branches in parallel (see section IV) while HAFIX accesses only one entry of their label state memory at once. Aside from all these, the most noticeable difference and contribution of our work is that ActiMon can be directly applied to a COTS hardware platform while HAFIX entails modifications to the processor internal architecture.

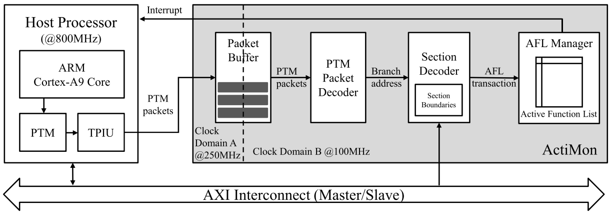


FIGURE 1. **ActiMon overall architecture.**

III. THREAT MODEL AND ASSUMPTIONS

We assume that the OS kernel, which configures the hardware modules, is uncompromised. Therefore, attackers cannot directly tamper with the configuration of ActiMon. It is assumed that the OS and CPU cooperate to forbid a memory page from being both writable and executable simultaneously by enforcing the W⊕X security protection rule. Under this assumption, attackers cannot directly execute their own code by modifying the code region of the target program. However, attackers are well aware of the implementation details of the target program and can undermine code randomization techniques such as Address Space Layout Randomization (ASLR) [40], [41]. Attackers also have full control over the stack and heap to exploit a memory overflow vulnerability. We focus on detection conventional CRAs subverting control-flow by corrupting return addresses and function pointers. Accordingly, other attacks such as COOP [42], non-control-data attack [43] and DOP [44] that are carried out by corrupting different types of data are outside of the scope of this paper. Since ActiMon necessitates the static analysis of the code, it cannot support the self-modifying code feature that allows dynamic changes in code contents. We also rule out physical attacks that try to compromise the underlying CPU and the ActiMon hardware modules.

IV. DESIGN

In this section, the design of ActiMon will be described in detail. Figure 1 depicts the overall architecture where the host ARM CPU and ActiMon are connected and interfaced. When the target program is running on the host CPU, ActiMon receives the runtime program information extracted by the built-in debug interface, ARM CoreSight PTM [45]. ActiMon is now able to execute the monitoring algorithm by tracking the target address of indirect branches gathered from the information it received. Hence, it successfully detects the control flow hijacking induced by JOP or ROP attacks. Our design goal hereby is to engineer an efficient monitoring algorithm and implement a viable architectural design for a CRA monitor on FPGA. Our ultimate objective is to evince the feasibility of FPGA as a computing device in the field for CRA defense. The following subsections will provide detailed descriptions of our algorithm and each hardware module in ActiMon.

A. PACKET BUFFER AND PACKET DECODER

First, the runtime program information extracted from ARM CPU is stored in *Packet Buffer*, then, decoded by *Packet Decoder* in order to obtain the indirect branch addresses. To elaborate and justify our design choices, details regarding ARM CoreSight PTM and TPIU will be presented as follows. ARM CoreSight PTM captures various debug information generated by the program, such as branch target addresses, exceptions, instruction set mode changes (ARM/THUMB) and current process IDs. This information is encoded into the trace packets which is routed to TPIU then, forwarded to the output pins afterward. Before TPIU emits the trace packets, the packets are temporarily stored in the internal FIFO, named *PTM FIFO* for ease of explanation. TPIU reads PTM FIFO at the rate of the clock provided from the external modules. By doing so, we achieve the synchronization of the rate between the emitted packets and the external module that processes them. Note that if the external clock is not fast enough to read PTM FIFO immediately after the packets are being stored, the PTM FIFO will soon overflow and the packets will be discarded. This overflow will eventually cause the failure of the monitoring algorithm. Since the previous study [16] assumed that the external module is operated at almost the same clock as ARM CPU, they did not need to address the overflow issue of PTM FIFO. In ActiMon, however, the modules that process PTM packets are operating on FPGA, so the clock speed is inevitably much slower than ARM CPU, as discussed in section I.

To overcome this limitation of FPGA, the speed of consuming the stored packets inside PTM FIFO is maximized by connecting 250MHz clock. It is the fastest clock that can be provided in our FPGA to the host CPU as the external clock of TPIU. However, the typical FPGA design cannot meet such tight timing constraint (i.e., 4ns clock period). ActiMon modules support 100MHz as the maximum clock rate. In order to link these two different clock domains, we have implemented the asynchronous buffer called Packet Buffer in between the host CPU and Packet Decoder. Packet Buffer is a two-port SRAM where the ports for writing/reading are separated, and they can be operated in different clock rates. In our design, Packet Buffer stores the transmitted packets in 250MHz clock rate synchronized with TPIU and Packet Decoder reads Packet Buffer in 100MHz. Then Packet Decoder decodes the read packets and analyze the legality according to the algorithm as will be explained in subsection IV-C. It takes less than five cycles to complete all the procedures from decoding to anlyzing the packets.

Although this asynchronous design can lower the possibility of overflow in PTM FIFO, it does not guarantee that all the packets will be decoded for a successful monitoring task by CRA monitor. Therefore, to assure that CRA monitor does not miss any PTM packets, which is the minimum requirement for CRA monitor, we should reduce the amount of PTM packets generated from the CPU to minimize the overflow possibility of PTM FIFO as well as Packet Buffer.

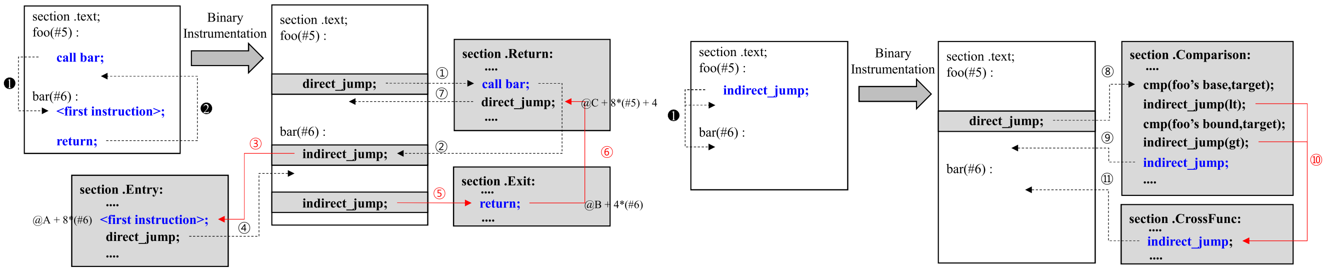
B. BINARY INSTRUMENTATION

In this subsection, we describe our special binary instrumentation mechanism to achieve the following two goals: (1) for PTM to emit only the necessary branch addresses for our CRA monitoring algorithm and (2) to supplement the lacked information for our algorithm.

Our first goal of the instrumentation is to filter out unnecessary indirect branch addresses before being stored in PTM FIFO. Recall that, ActiMon needs to track only the target addresses of the indirect branches that jump to outside of the currently executing function as mentioned in section I. To extract only those cross-function jumps, we leveraged *Address Comparators* inside PTM which only pass through the branch target addresses that match the ranges set by eight pairs of Address Comparator registers. However, regardless of how many registers are available, grouping the target of cross-function jumps that are evenly distributed throughout the code would be not only inefficient but also deterrent to achieving the first goal of ours. This is due to the fact that it will inevitably include useless information (i.e., branches that are not cross-function jumps) in a group. Hence, ActiMon is designed to only collect necessary pieces of information which will require just a single Address Comparator to do the job.

In relation to the second goal, the target address of an indirect branch can only be acquired from PTM packets coming through the debug interface. While gathering the target addresses of indirect branches are quite straightforward in our solution as the ARM debug interface is designed to provide such information, it lacks the following four classes of information: (1) type of indirect branches, (2) addresses of each function entry, (3) source and (4) target addresses of return.

We resolved these limitations by adding new code sections, so-called *trampolines*, in the binary instrumentation. The first step of the instrumentation is to move the first instructions of each function, returns, calls and indirect jumps in the original code to an associated location in the trampoline. They are then replaced with jump instructions which point to the associated place. We note that aforementioned binary modifications regarding trampolines preserve the original code layout. To be concrete, the trampolines are located at the end of the original code section and the jump instructions to the trampolines are added by overwriting an existing indirect branch instruction. Thus, all references to code, such as via function pointers, remain intact even after the binary modifications. Each trampoline code section is named *Entry*, *Exit*, *Return*, *Comparison* and *CrossFunc* respectively. We will explain each code section in detail below. Figure 2 illustrates how the trampoline code sections are inserted into the original binary. The modified codes are represented by grey-shaded areas in the figure.



(a) Instrumentation for Call and Return

(b) Instrumentation for Indirect Jump

FIGURE 2. Comparison between the original binary and the instrumented binary.

1) ENTRY SECTION

Entry section is for delivering the following information to ActiMon: (1) the fact that current control-flow moves to a function entry and (2) which function is called. As shown in Figure 2a, we make each function correspond to the dedicated location in Entry section. We move the original first instruction to the corresponding location in Entry section and insert a new indirect jump that jumps to Entry section, at the entry of the function. Also, we insert a new direct jump in Entry section strictly after the first instruction of the function that has already been inserted. So, When the target address of indirect branch ③ in Figure 2a is delivered to ActiMon, ActiMon can calculate the unique function label corresponding to the delivered branch address. For example, if the branch target address is $@A + 8 * (\#6)$ where A is the starting address of Entry section, it means that function#6 is called.

2) EXIT SECTION

Exit section is for informing ActiMon the followings: (1) which function is returned and (2) the fact that current indirect branch is a return. Similar to Entry section, ActiMon can infer the returned function label from the branch target address (emitted from ⑤) within Exit section. For example, if the target of ⑤ is $@B + 4 * (\#6)$ where B is the starting address of Exit section, function#6 is returned. We must uniquely handle one particular case by ARM which does not have the return instruction. Rather, it is implemented by `pop pc`, or `bx lr` where both `pop` and `bx` can contain the condition to be executed. To maintain the original functionality, if the original return contains the condition, we simply exchange the newly inserted indirect jump at the exit of the function with the conditional indirect jump. Return instruction inserted in Exit section will be without the condition.

3) RETURN SECTION

Return section is for delivering the target function of return to ActiMon. Two instructions per every function call are added to Return section. The first instruction is a call for the function which is to be invoked in the original binary (②). The second instruction is a direct jump to move control-flow back to the original return target (⑦). The offset within Return section corresponds to the target function of return. For example, if the target of ⑥ is $@C + 8 * (\#5) + 4$ where C is the starting address of Return section, the corresponding return targets function#5.

4) COMPARISON SECTION

Comparison section is for checking if the current indirect jump is cross-function jump and preventing the generation of PTM packets when the jump does not cross the function. All the indirect jumps are redirected to Comparison section by a direct branch as shown in Figure 2b. One thing to note here is that we configure PTM not to emit direct branch addresses. So, the newly added direct branch does not increase the amount of PTM packets. It would be acceptable to monitor only the indirect branch because CRAs mainly targets the indirect branch rather than the direct branch. In Comparison section, the instrumented codes compare the target address of the original indirect jump with the currently executing function boundaries. Function boundaries are statically extracted from the binary and inserted as a constant operand for `cmp` instruction. Only when the target address lies outside of the currently executing function, the instrumented indirect jump to CrossFunc section is executed (⑩ in Figure 2b).

5) CROSSFUNC SECTION

CrossFunc is for noticing ActiMon that the current indirect branch is an indirect jump and also cross-function jump. CrossFunc section only contains the original indirect branches substituted by the direct branch to Comparison section. ActiMon can infer the occurrence of cross-function jumps by checking the branch addresses are in CrossFunc section.

Based on these instrumented sections, the essential PTM packets for ActiMon are generated during only the following four cases (represented as the solid red arrow lines in Figure 2b): (1) moving from the original code section to Entry section ③ or (2) Exit section ⑤ or (3) Return section ⑥, and (4) moving from Comparison section to CrossFunc section ⑩. In other words, ActiMon needs to monitor the branches that jump only to the above target sections (i.e., Entry, Exit, Return and CrossFunc sections). So, we set the starting and ending addresses of each target section to the Address Comparator registers. Since the instrumented sections are arranged in consecutive addresses, a single pair of Address Comparator registers that cover the sections at once can filter out irrelevant PTM packets. This filtering greatly relieves the heavy data pressure of both PTM FIFO and Packet Buffer that was discussed in subsection IV-A.

C. JOP/ROP DETECTION PROCESS

In this subsection, we explain how ActiMon detects the illegal change of the control-flow induced by JOP and ROP attacks. In the core of our algorithm, ActiMon manages *Active Function List* (AFL) to check the existence of those attacks. If ActiMon detects a branch to Entry section, the corresponding function label in AFL is set to one, i.e., activated. On the other hand, the corresponding activated label in AFL is set back to zero when a branch to the corresponding location in Exit section is executed. Our detection algorithm is as summarized in section I. The detailed procedure is as the following.

JOP checking procedure starts from when a branch to CrossFunc section occurs. (⑩ in Figure 2b). Then, ActiMon waits for a branch to Entry section to verify the rule of branch regulation (BR) [20] where the cross-function jump must target the function entry (③). If a branch to other sections occurs in advance to the one to Entry section, ActiMon generates the interrupt that alarms the existence of a JOP attack for ARM CPU to terminate the target program immediately. Since all the cross-function indirect jumps must pass through CrossFunc section, ActiMon does not miss any malicious jump that violates the BR rule. It is worth to note that ActiMon has a delay for detecting JOP attacks. The detection delay is caused by executing several instructions, called a gadget before encountering the next cross-function indirect jump instruction. However, we believe that the delay is negligible as a single gadget is a short code sequence. [1]

In order to check the existence of ROP attacks, we regard a return targeting the non-activated function as an illegal control-flow induced by ROP attacks. The procedure for monitoring ROP begins once a branch to Return section occurs (⑥). Hereafter, ActiMon checks if the function label corresponding to the location in Return section is activated in AFL. If the corresponding location is not activated, ActiMon generates an interrupt to raise the ROP attack flag for ARM CPU. The AFL based mechanism is noteworthy in that it does not occur false positives for certain programming constructions that involve unusual stack management (e.g., C++ exceptions with stack unwinding and setjmp/longjmp) [30].

D. HARDWARE MODULES FOR THE DETECTION

In this subsection, we explain the hardware modules that perform our monitoring algorithm. Firstly, Packet Decoder extracts the branch addresses from PTM packets stored in Packet Buffer and delivers the extracted branch addresses to *Section Decoder*. Section Decoder generates AFL transactions from the delivered branch addresses by determining which code section and function corresponds to those branch target addresses. AFL transactions are then delivered to *AFL Manager*. It checks if the rules explained in subsection IV-C are violated based on AFL which is being updated by AFL transactions.

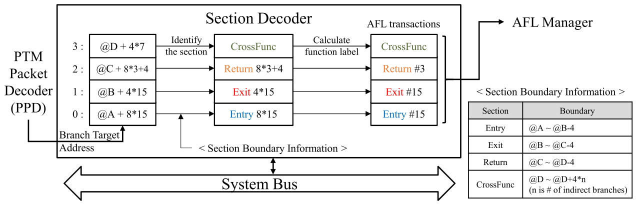


FIGURE 3. Operation of section decoder.

TABLE 1. List of AFL transactions.

AFL Transaction	Description
CrossFunc	Triggering JOP detection process (the current indirect branch is jumping out of a function boundary)
Entry#A	Activating function#A in AFL (the current control is moving to the entry of function#A)
Exit#B	Deactivating function#B in AFL (the current return is invoked from function#B)
Return#C	Triggering ROP detection process (the current return is jumping to function#C)

1) SECTION DECODER

Figure 3 depicts the operation of Section Decoder in detail. Firstly, to find which code section the branch has jumped to, each branch address is compared to the section boundary information that is set by the host CPU prior to running the target program. Each boundary can be obtained statically from the instrumented binary. Then, Section Decoder determines the function labels by calculating the relative offset from each starting address of the code sections. For example, if the relative offset of the branch address belonging to Entry section is 120 ($= 8*15$) in decimal, recalling that two instructions per the corresponding function are inserted in the entry section, the function label is determined to be #15 as shown in Figure 3. The section information and the determined function label are combined into an AFL transaction. There are four transactions for managing AFL and triggering JOP/ROP detection process as represented in Table 1. `CrossFunc` and `Return#C` are the transactions triggering AFL Manager to run the rule check for JOP and ROP detection respectively. `Entry#A` and `Exit#B` are for updating the entries of AFL.

For better performance, Section Decoder is designed to process four branch addresses simultaneously. The number of branches is related to the amount of packet data transmitted at once. More specifically, the output port of ARM CPU for emitting PTM packet data can be configured to various sizes, i.e., from 1-bit to 32-bit. We set the size of the port to the maximum value for the fastest possible extraction of the PTM packets from PTM FIFO within ARM CPU.

[1] According to our investigation, the average length of available gadget length is 5 instructions. Such a short length comes from the fact that long gadgets considerably reduce stability of execution [18].

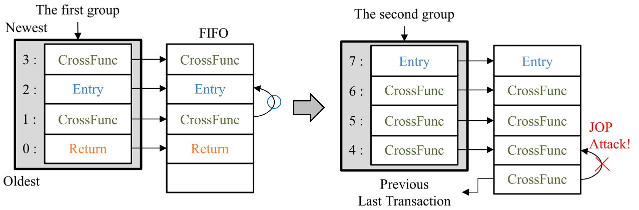
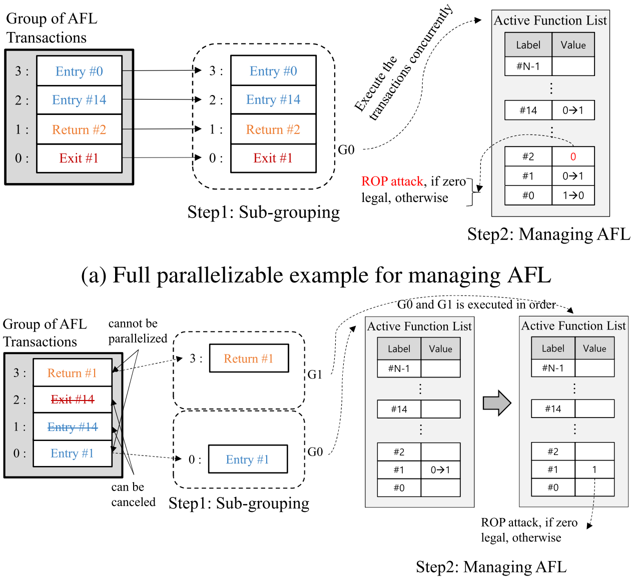


FIGURE 4. JOP detection process of AFL Manager.



(a) Full parallelizable example for managing AFL

(b) Partial parallelizable example for managing AFL

FIGURE 5. ROP detection process of AFL Manager.

Since the length of the packet containing branch information can be 8-bit at its minimum [45], Packet Decoder can decode up to four branch addresses at a time. The following AFL Manager is also designed to handle four AFL transactions at once to synchronize the generation rate of AFL transactions in Section Decoder. We found that ARM CPU holds the PTM packets in the PTM FIFO until 4 bytes packets are gathered (the concrete time or cycles is not documented [46]), which may delay the detection of attacks that have already carried out by this buffering period. We deem that this limitation would be overcome easily once if ARM provides the mode that immediately emits the PTM packet rather than buffering unconditionally.

2) AFL MANAGER

We now describe AFL Manager that manages AFL and enforces our JOP/ROP detection rule by processing AFL transactions transmitted from Section Decoder.

3) JOP DETECTION PROCESS

Figure 4 illustrates a JOP detection process of AFL Manager where thereby AFL Manager processes two groups of AFL transactions in order. As shown in the figure, AFL Manager enforces the BR rule for every group of AFL transactions delivered. If a section other than Entry section is followed after CrossFunc section, it is regarded as a JOP attack and an interrupt is immediately generated. Although the rule imposed by AFL Manager checks two consecutive AFL transactions, it is not always the case where both of them are located in the same group from Section Decoder. Figure 4 illustrates such case. Therefore, it is crucial to carefully engineer AFL Manager to keep the last AFL transaction of the previous group and combine it with the first transaction of the next group. Figure 4 shows an example of a typical JOP attack that stitches the gadgets by consecutive cross-function jumps. At the second group, an interrupt will be raised to alert ARM CPU of JOP attack because CrossFunc section is followed by other than Entry. In this case, it was another CrossFunc that came after CrossFunc.

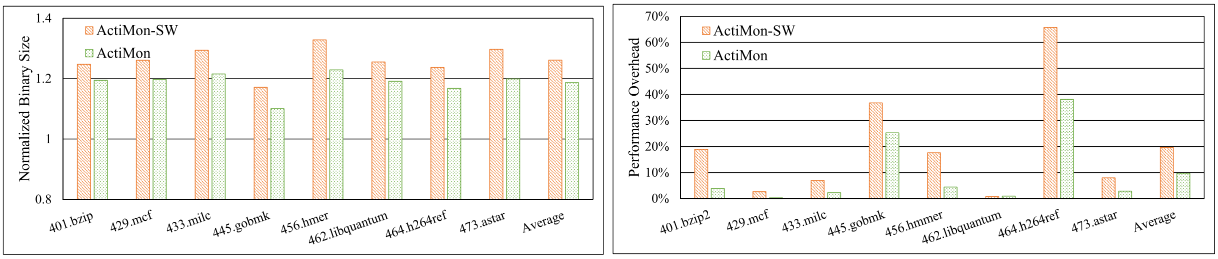
4) ROP DETECTION PROCESS

Figure 5 illustrates ROP detection process of AFL Manager. As discussed in section I, AFL Manager is designed to process multiple AFL transactions concurrently as long as there is no dependency among the transactions. Figure 5a portrays one of these full parallelizable cases. Since function labels are all different, in other words, there is no dependency issue, all the transactions are classified in the same subgroup which is to be processed in parallel. Consequently, the second transaction (denoted by Return#2) triggers the checking logic for ROP detection rule and then AFL Manager raises an alarm for ROP attack since function#2 is inactive on AFL. At the same clock cycle, function#0 and function#14 are activated, and function#1 is deactivated. It is worth to note that the value of each AFL *Value* entry is a counter to consider recursive function calls. It prevents the false positive in which ActiMon raises an alarm even though the program is normally executing the recursive function calls, by inrementing/decrementing the counter when the same function is successively called/returned.

In the second example Figure 5b, the transactions having the same function labels are found. Before dividing a group into subgroups, AFL Manager deletes the paired transactions that have no effect on AFL. Entry and Exit with the same label would be canceled out as long as Entry is followed by Exit. This prior cancellation may give another chance of performance gain by saving the clock cycles needed for updating AFL. In Figure 5b, there is no clock cycle reduction by the deletion because there is another pair with a dependency (Entry#1, Return#1). However, if a group of AFL transactions is (Entry#1, Entry#2, Exit#2, Return#0), the Entry#2/Exit#2 are canceled out and Entry#1/Return#0 is grouped into a single subgroup that is to be executed in parallel, so that only one clock cycle is needed to manage AFL in this case.

V. EVALUATION

To evaluate ActiMon, we have loaded it on a Xilinx Zynq-7000 ZC706 platform which is equipped an SoC XC7Z045 FFG900-2 incorporating a dual-core ARM Cortex-A9 processor and an FPGA together. We have built



(a) Binary size overhead comparison

(b) Performance overhead comparison

FIGURE 6. Overhead comparison between ActiMon-SW and ActiMon.

the host system with the A9 processor that is set by the maximum possible 800MHz clock speed and deployed Xilinx ARM Linux kernel 4.9 as the host OS. Also, the two CoreSight modules, PTM and TPIU, in the Cortex-A9 processor are enabled to extract branch traces from the CPU. The ActiMon modules are developed in Verilog HDL to be mapped on the FPGA. In the following subsections, we evaluate ActiMon in terms of storage and performance overhead, then report the synthesis result and the security evaluation.

A. STORAGE AND PERFORMANCE OVERHEAD EVALUATION

Since no existing hardware-based CRA monitor solution can operate validly in our environment where the monitor is eight times slower than CPU, we implemented the software version of ActiMon, named by ActiMon-SW in order to compare with our ActiMon. To implement ActiMon-SW, we instrumented additional codes to the target binary which execute every process in ActiMon operating in FPGA. The target binaries were selected from the SPEC CPU2006 benchmark [22].

Figure 6a depicts the comparison of the storage overhead due to the binary instrumentation between ActiMon-SW and ActiMon. The y-axis value represents the instrumented binary size that is normalized by adjusting the size of the original binary to be 1. The higher bar means that more codes are inserted to the original binary. Since each code of the benchmark program includes a different number of branches, the storage overhead varies. ActiMon incurred only 18.7% storage overhead on average, while 26.2% for ActiMon-SW. If we simply compare the number with the previous work with ignoring their limitations, the storage overhead of 18.7% by ActiMon is relatively comparable to that of 16.6% by [16].

Figure 6b illustrates a graph comparing the performance overhead between ActiMon-SW and ActiMon. The performance overhead was calculated by measuring the execution time of both the original binary and instrumented binary. The higher bar in the figure means that the performance overhead is greater. ActiMon-SW shows the performance overhead of 19.7% on average while 9.77% for ActiMon. In other words, The performance overhead was reduced to half by offloading the monitoring algorithm to our ActiMon operated on FPGA.

The performance overhead also varied with respect to each benchmark program similar to the storage overhead. However, the performance overhead correlated to the number of branch execution in runtime, while the storage overhead was proportional to the number of branch instructions in the code. As shown in Figure 6b, our instrumented codes corresponding to indirect branches are executed more frequently especially in `445.gobmk` and `464.h264ref`. The internal buffers (PTM FIFO, Packet Buffer) did not overflow during running those two benchmark programs as well.

TABLE 2. Synthesis result of ActiMon.

Modules		LUTs	FFs	BRAM
ActiMon	Packet Buffer	115	229	4
	Packet Decoder	3064	954	0
	Section Decoder	1624	100	0
	AFL Manager	95214	7799	0
	Total	100017	9082	4
	% over FPGA resources	45.75%	2.08%	0.73%

B. SYNTHESIS RESULT

We have synthesized ActiMon onto the FPGA in the Zynq-7000 board and quantified the logics necessary for the hardware modules of ActiMon in terms of lookup tables for logic (LUTs), flip-flops (FFs) and memory elements (BRAMs). Such modules include Packet Buffer, Packet Decoder, Section Decoder and AFL Manager. The synthesis results are shown in Table 2. The total hardware resources available for the Zynq board are 218600 LUTs, 437200 FFs, and 545 BRAMs, and ActiMon utilizes 45.75%, 2.08%, and 0.73% of that for each hardware resource respectively. As shown in Table 2, AFL Manager which comprises of AFL and logics for managing AFL, occupies most of the resources within ActiMon (i.e., 95.20% of LUTs and 85.86% of FFs).

The fact that the percentage of resource usage is highly concentrated on one module (i.e., AFL Manager) is related to our design choice for the performance, in which our AFL was implemented by flip-flops, not by BRAM. If AFL was made of BRAM, additional clock cycles would have been required to access entries of AFL. In contrast, flip-flops do not require a clock cycle to access the value because the data is always loaded on Q port. More importantly, we can access multiple entries concurrently as explained in subsection IV-D which is only feasible if AFL was built with flip-flops. As a result, such FFs/LUTs are respectively mapped to AFL entries and the logics (e.g., muxes and adders) for reading and updating all the entries of AFL concurrently. In our evaluation, the number of entries of AFL is implemented as 2048, and each entry value can range from 0 to 7. We decided on the number of the entry to be 2048 so as to cover the maximum number of functions in our benchmark programs used for the evaluation (i.e., $6k$ $(= 2048 * 3)$ bits are allocated to AFL in total). If we try to adjust the size of the AFL according to the specific target program, we merely have to adjust the parameters (i.e., the number of entries and the bit-width of each one) of the AFL Manager Verilog-HDL code and then load the resynthesized ActiMon on FPGA. By utilizing the programmability of FPGA in this way, it is easy to update the ActiMon module.

C. SECURITY EVALUATION

To evaluate the detection capability of our ActiMon, we have implemented five CRA samples as explained in [16]. Two samples are ROP attacks to open a shell, the other two samples are JOP attacks that aim to achieve the same goal as the first two ROP attacks, and another sample is a ROP attack to invoke `mprotect` system call. For this, we used a vulnerable program that has buffer overflow vulnerability and exploited it to launch the CRA samples. More specifically, we corrupted the stack by inputting a file that has characters of a length greater than the allocated for the destination of `strcpy`. Since at least one gadget comes from an inactive function in all tested ROP samples, ActiMon correctly detected the existence of such the ROP attacks. Similarly, since every corrupted target of indirect jumps in all tested JOP samples is always beyond the currently executing function bounds and does not target a function entry, ActiMon, again, successfully detected such JOP attacks.

To analyze the reliability of ActiMon more statistically, we analyzed the false positive and the false negative rate. First, there is no false positives—ActiMon does not make a false alarm on normal program executions. Our interested cross-function jumps are induced by a function call or a return instruction. Since every legal function call should transfer the program control to entries of the target functions, they must obey our JOP detection rule, BR. Similarly, all legal returns jump back to the call sites which must be activated already in our AFL.

On the other hand, ActiMon involves a little false negatives regarding its monitoring capability. Simply put, ActiMon might be bypassed by JOP and ROP attacks that stitch gadgets still in active functions as shown in [47], but we believe the risk is minor. First, the false negative rate for detecting JOP attacks is virtually zero. To carry out JOP attacks, a dispatcher gadget (to orchestrate the sequence of gadgets) and a system call instruction (to do something beyond the compromised program) are required, but it is hardly that both the dispatcher gadget and system call exist together in an active function. Experimentally, for instance, no such a case is found in the commonly used libraries such as libc or libssl [20].

Similarly, the false negative rate of ActiMon for ROP attacks is also limited. We evaluated the false negative rate for detecting ROP by analyzing gadgets reduction. To measure to what extent ActiMon reduces the set of valid gadgets, we calculated and compared the number of the gadgets in a program and the number of the actually available gadgets at each return instruction. Table 3 shows the gadget reductions by ActiMon. We utilized *ROPgadget* to identify ROP gadgets in code and *Callgrind*, a sub tool of *Valgrind*, to log each function call and call counts. After logging all call history of each benchmark program, we built a call-graph to identify possible active functions from the *main* function to the leaf functions, and calculated the min/max/average number of ROP gadgets from the activated functions. As a result, ActiMon reduces the number of gadgets by 96.1% on average. ActiMon exhibits 92.56% reduction on average even at the worst case (max.) among all the return instructions. This reduction is relatively better than those of related work, 88.93% [32] and 91.92% [33].

TABLE 3. ROP gadgets reduction of ActiMon.

Benchmark	number of gadgets at return			total gadgets	avg. gadget reduction
	min.	max.	avg.		
401.bzip2	20	331	316	923	65.76%
429.mcf	9	38	27	256	89.45%
433.milc	11	54	35	1548	97.74%
445.gobmk	57	753	382	12198	96.87%
456.hmmer	28	73	38	3750	98.99%
462.libquantum	15	76	28	753	96.28%
464.h264ref	41	467	99	4984	98.01%
473.astar	31	75	53	691	92.33%
Total	212	1867	978	25103	96.10%

VI. CONCLUSION

The recent rise of FPGA as a versatile embedded computing device has motivated us to implement a CRA monitor (i.e., ActiMon) inside an ARM system that is built together with FPGA within a COTS FPGA SoC platform readily available in the market today. To overcome the performance handicap of FPGA, we have crafted ActiMon to achieve fast performance sufficient to detect realistic JOP/ROP attacks on a real SoC FPGA platform. We ascribe such achievement to its lightweight, unified algorithm based on AFL. To further improve the performance, ActiMon processes multiple branches in parallel whenever possible. Additionally, our binary instrumentation mechanism alleviates the performance overhead of ActiMon by filtering out all jumps except essential cross-function jumps before transmitting them to it. Consequently, the experiments demonstrated that unlike existing solutions piggybacking the high speed of custom SoCs, our ActiMon was able to meet the minimum performance requirements for successful JOP/ROP detection, even when running on SoC FPGA, with acceptable storage overhead of 18.7% and performance overhead of 9.77% on average.

REFERENCES

[1] Amazon. (2018). *AWS EC2 FPGA Development Kit*. [Online]. Available: https://github.com/aws/aws-fpga

[2] Intel. (2018). *Intel(r) Programmable Acceleration Card (PAC) With Intel(r) Arria(r) 10 GX FPGA Datasheet*. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-a10.pdf

[3] (2018). *Intel(r) Xeon(r) Gold 6138 Processor*. [Online]. Available: https://en.wikichip.org/wiki/intel/xeon_gold/6138p

[4] Xilinx. (2018). *Zynq-7000 SOC Data Sheet: Overview*. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[5] (2018). *Versal: The First Adaptive Compute Acceleration Platform (ACAP)*. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf

[6] Microchip. (2019). *Polarfire FPGA: Building a MI-V Processor Subsystem*. [Online]. Available: https://www.microsemi.com/document-portal/doc_download/136945-tu0775-polarfire-fpga-building-a-risc-v-processor-subsystem-tutorial

[7] K. Neshatpour, M. Malik, M. A. Ghodrat, and H. Homayoun, "Accelerating big data analytics using FPGAs," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, p. 164.

[8] R. Dhanabal, S. K. Sahoo, V. Bharathi, K. Dowluri, B. S. R. P. Varma, and V. Sasiraju, "Fpga based image processing unit usage in coin detection and counting," in *Proc. Int. Conf. Circuits, Power Comput. Technol. (ICCPCT)*, Mar. 2015, pp. 1–5.

[9] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA based neural network accelerator," *CoRR*, vol. abs/1712.08934, Dec. 2017, pp. 1–26. [Online]. Available: http://arxiv.org/abs/1712.08934

[10] B. Nagy, P. Orosz, and P. Varga, "Low-reaction time FPGA-based DDoS detector," in *Proc. NOMS IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2018, pp. 1–2.

[11] A. Boutros, B. Grady, M. Abbas, and P. Chow, "Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2017, pp. 1–6.

[12] A. Surendar, "FPGA based parallel computation techniques for bioinformatics applications," *Int. J. Res. Pharmaceutical Sci.*, vol. 8, pp. 124–128, Jan. 2017.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Secur.*, New York, NY, USA, 2005, pp. 340–353, doi: 10.1145/1102120.1102165.

[14] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *Proc. 5th Int. Conf. Inf. Syst. Secur.* Berlin, Germany: Springer-Verlag, 2009, pp. 163–177, doi: 10.1007/978-3-642-10772-6_13.

[15] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2015, pp. 914–926, doi: 10.1145/2810103.2813644.

[16] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Using CoreSight PTM to integrate CRA monitoring IPs in an arm-based SoC," *ACM Trans. Des. Autom. Electr. Syst.*, vol. 22, p. 52:1–52:25, 2017.

[17] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[18] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur.*, New York, NY, USA, 2011, pp. 30–40, doi: 10.1145/1966913.1966919.

[19] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, New York, NY, USA, 2007, pp. 552–561, doi: 10.1145/1315245.1315313.

[20] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Proc. 39th Annu. Int. Symp. Comput. Archit.* Washington, DC, USA: IEEE Computer Society, 2012, pp. 94–105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337171

[21] T.-C. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proc. 21st ICDSC*, Apr. 2001, pp. 409–417.

[22] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006, doi: 10.1145/1186736.1186737.

[23] 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 *Proc. IEEE Symp. Secur. Privacy*. Washington, DC, USA: IEEE Computer Society, May 2013, pp. 559–573, doi: 10.1109/SP.2013.44.

[24] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proc. 22nd USENIX Conf. Secur.* Berkeley, CA, USA: USENIX Association, 2013, pp. 337–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534796

[25] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 12, pp. 1295–1308, Dec. 2006.

[26] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, New York, NY, USA, 2016, pp. 38–49, doi: 10.1145/2857705.2857722.

[27] W. He, S. Das, W. Zhang, and Y. Liu, "No-jump-into-basic-block: Enforce basic block CFI on the fly for real-world binaries," in *Proc. 54th Annu. Design Autom. Conf.*, New York, NY, USA, 2017, pp. 23:1–23:6, doi: 10.1145/3061639.3062291.

[28] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proc. 1st ACM Workshop Secure Execution Untrusted Code*, New York, NY, USA, 2009, pp. 19–26, doi: 10.1145/1655077.1655083.

[29] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from code reuse attacks," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*. Washington, DC, USA: IEEE Computer Society, Feb. 2013, pp. 258–269, doi: 10.1109/HPCA.2013.6522324.

[30] 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 *Proc. 52nd Annu. Design Autom. Conf.*, New York, NY, USA, 2015, pp. 74:1–74:6, doi: 10.1145/2744769.2744847.

[31] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *Proc. 53rd Annu. Design Autom. Conf.*, New York, NY, USA, 2016, pp. 163:1–163:6, doi: 10.1145/2897937.2898098.

[32] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu, "Control flow integrity based on lightweight encryption architecture," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 7, pp. 1358–1369, Jul. 2018.

[33] P. Qiu, Y. Lyu, D. Zhai, D. Wang, J. Zhang, X. Wang, and G. Qu, "Physical unclonable functions-based linear encryption against code reuse attacks," in *Proc. 53nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[34] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-assisted control-flow integrity checking," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 458–471, Feb. 2019.

[35] Z. Guo, R. Bhakta, and I. G. Harris, "Control-flow checking for intrusion detection via a real-time debug interface," in *Proc. Int. Conf. Smart Comput. Workshops*, Nov. 2014, pp. 87–92.

[36] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Integration of ROP/JOP monitoring IPs in an arm-based SoC," in *Proc. Conf. Design, Autom. Test Eur.*, San Jose, CA, USA: EDA Consortium, 2016, pp. 331–336. [Online]. Available: http://dl.acm.org/citation.cfm?id=2971808.2971884

[37] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient security monitoring with the core debug interface in an embedded processor," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, pp. 8:1–8:29, May 2016, doi: 10.1145/2907611.

[38] 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 *Proc. 4th Workshop Hardw. Architectural Support Secur. Privacy*, New York, NY, USA, 2015, pp. 3:1–3:8, doi: 10.1145/2768566.2768569.

[39] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–12.

[40] E. Gökta , R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 105–119.

[41] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 121–138.

[42] 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 *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 745–762.

[43] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, ''Non-control-data attacks are realistic threats,'' in *Proc. 14th Conf. USENIX Secur. Symp.*, vol. 14. Berkeley, CA, USA: USENIX Association, 2005, p. 12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251410

[44] H. Hong, S. Shweta, A. Sendroiu, L. C. Zheng, S. Prateek, and L. Zhenkai, ''Data-oriented programming: On the expressiveness of non-control data attacks,'' in *Proc. IEEE Symp. Secur. Privacy (S P)*, May 2016, pp. 969–986.

[45] AC Ltd. (2017). *Arm CoreSight Architecture Specification V3.0*. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0029e/coresight_v3_0_architecture_specification_IHI0029E.pdf

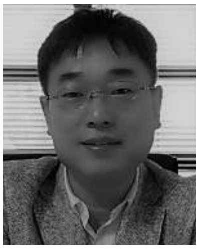
[46] (2017). *CoreSight PTM-A9 Techincal Reference Manual*. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0401c/DDI0401C_coresight_ptm_a9_r1p0_trm.pdf

[47] M. Theodorides and D. Wagner, ''Breaking active-set backward-edge CFI,'' in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2017, pp. 85–89.

HYUNYOUNG OH received the B.S. and M.S. degrees in electrical and electronics engineering from Yonsei University, South Korea, in 2005 and 2007, respectively. He is currently pursuing the Ph.D. degree in electrical and computing engineering with Seoul National University, South Korea. He was a SoC Designer with Samsung Electronics, South Korea, from 2007 to 2017. His research interest includes hardware-backed system security against various types of threats.

MYONGHOON YANG received the B.S. degree in electronics and radio engineering from Kyunghee University, South Korea, in 2016, and the M.S. degree in electrical and computing engineering from Seoul National University, South Korea, in 2019. His research interest includes hardware-backed system security against various types of threats.

YEONGPIL CHO received the B.S. degree in electrical engineering from POSTECH, South Korea, in 2010, and the Ph.D. degree in electrical and computer engineering from Seoul National University, South Korea, in 2018. He is currently a Professor with the School of Software, Soongsil University. His research interest includes system security against various types of threats.



YUNHEUNG PAEK received the B.S. and M.S. degrees in computer engineering from Seoul National University, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, in 1997. He is currently a Professor with the Department of Electrical and Computer Engineering, Seoul National University. His research interests include system security with hardware, secure processor design against various types of threats, and machine learning-based security solution.