FastCFI: Real-time Control-Flow Integrity Using FPGA without Code Instrumentation

LANG FENG, Nanjing University, China and Texas A&M University, USA
JEFF HUANG, JIANG HU, and ABHIJITH REDDY, Texas A&M University, USA

Control-Flow Integrity (CFI) is an effective defense technique against a variety of memory-based cyber attacks. CFI is usually enforced through software methods, which entail considerable performance overhead. Hardware-based CFI techniques can largely avoid performance overhead, but typically rely on code instrumentation, forming a non-trivial hurdle to the application of CFI. Taking advantage of the tradeoff between computing efficiency and flexibility of FPGA, we develop FastCFI, an FPGA-based CFI system that can perform fine-grained and stateful checking without code instrumentation. We also propose an automated Verilog generation technique that facilitates fast deployment of FastCFI, and a compression algorithm for reducing the hardware expense. Experiments on popular benchmarks confirm that FastCFI can detect fine-grained CFI violations over unmodified binaries. When using FastCFI on prevalent benchmarks, we demonstrate its capability to detect fine-grained CFI violations in unmodified binaries, while incurring an average of 0.36% overhead and a maximum of 2.93% overhead.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Computer systems organization** → **System on a chip**;

Additional Key Words and Phrases: Control-flow integrity, field-programmable gate array, security

ACM Reference format:
Lang Feng, Jeff Huang, Jiang Hu, and Abhijith Reddy. 2021. FastCFI: Real-time Control-Flow Integrity Using FPGA without Code Instrumentation. *ACM Trans. Des. Autom. Electron. Syst.* 26, 5, Article 39 (May 2021), 30 pages.
https://doi.org/10.1145/3458471

1 INTRODUCTION

Control-Flow Integrity (CFI) [1] inspects the transitions between instruction sequences (e.g., branches to other code segments) to ensure the program behaves as intended. This regulation can prevent software execution from being redirected to erroneous address or malicious code. It is widely recognized as an effective approach to defend against a variety of security attacks including **Return-Oriented Programming (ROP)** [36] and **Jump-Oriented Programming (JOP)** [4].

This work is partially supported by NSF (CNS-1618824).
Authors' addresses: L. Feng, Nanjing University, 163 Xianlin Road, Qixia District, Nanjing, China, 210023, Texas A&M University, 400 Bizzell Street, College Station, Texas, USA, 77840; email: flang@nju.edu.cn; J. Huang, J. Hu, and A. Reddy, Texas A&M University, 400 Bizzell Street, College Station, Texas, USA, 77840; emails: {jeffhuang, jianghu, abreddy}@tamu.edu.

1084-4309/2021/05-ART39 $15.00
https://doi.org/10.1145/3458471

Software-based CFI usually competes for the same processor resource as the software application being protected [1, 10, 32, 43], and therefore it tends to incur large performance overhead unless its resolution is very coarse grained. Alternatively, CFI can be realized through hardware-based enforcement, which is performed largely external to software execution and thus involves a much lower overhead. Indeed, hardware-based CFI has attracted significant research attention recently [14, 16, 27, 42].

Apart from a relatively low overhead, there are some other issues of hardware CFI that are worth a look.

(1) **Code instrumentation**. Previous hardware CFI methods often rely on code instrumentation [6, 11, 23, 24, 26, 27, 39]. Additional code is added to the application software being protected for more executing information. This causes performance overhead, may introduce extra security vulnerability, and is not always practically feasible [19].
(2) **Granularity**. Fine-grained CFI can detect detailed violations that would be missed by coarse-grained CFI. For example, the only legal instruction flow transitions are from code segment A to B, denoted by $A \rightarrow B$, and $C \rightarrow D$. A coarse-grained CFI may only check if a transition target is legitimate without examining the transition source. As such, an illegal transition $A \rightarrow D$ would pass such coarse-grained CFI check as D is a legitimate target, while fine-grained CFI can detect this violation. However, fine-grained CFI usually leads to larger performance overhead [1], thus, granularity is sacrificed by successive researches for the best tradeoff [5, 12, 32].
(3) **Stateful CFI**. Whether or not a transition is legal may depend on its history. For example, a transition $C \rightarrow D$ is legal only when its previous transition is $A \rightarrow C$, while transition $B \rightarrow C$ is also legal. Therefore, transition $B \rightarrow C \rightarrow D$ is illegal. Most previous hardware CFI works [13, 18, 34, 42] are stateless, and in this case only check $C \rightarrow D$ without examining its history.

The first issue affects practical applications. The next two issues are for security in terms of CFI coverage. To the best of our knowledge, there is no previous work that well addresses all of these issues along with low overhead.

In this article, we present FastCFI, which is an FPGA-based CFI system. FPGA implementation is a customized hardware solution that is typically around 100 times more power-efficient than software-based solutions on general purpose microprocessors [28]. In embedded applications, such as autonomous vehicles, such power-efficiency is particularly desirable. At the same time, the reconfigurability of FPGAs provides an important flexibility that is not available in dedicated ASIC solutions. Largely due to these appealing advantages, Microsoft adopts FPGA for its datacenters [33]. FastCFI also inherits the computing efficiency and flexibility of FPGA. The computing efficiency arises from the fact that FPGA computing can considerably circumvent system overhead and intrinsically support parallel processing.

FastCFI is the first fine-grained and stateful CFI system with negligible overhead and without using code instrumentation. Our system, FastCFI, is a low-latency CFI-regularization system, which we show does not produce any false alarms.

A comparison between FastCFI and some major works is provided in Table 1.

The main contributions of our article are as follows:

- A CFI system without code instrumentation or processor architecture/instruction set modification.
- A detailed design of a hardware-based fine-grained and stateful CFI system with low latency.

Table 1. Comparison among Different Methods

Method	Fine-grained	Stateful	No instrumentation	< 1% overhead	No false alarm
FastCFI	√	√	√	√	√
Lee [27]	×	√	×	×	√
CONVERSE [18]	√	×	√	√	×
Griffin [16]	√	√	√	×	√
FlowGuard [29]	×	√	√	×	√
CFIMon [42]	×	×	√	×	×
MoCFI [10]	√	√	×	×	×
Zhang [43]	×	√	×	×	√
kBouncer [32]	×	√	√	√	√
Ding [14]	√	√	√	×	√
Abadi [1]	√	√	×	×	√

- A concrete system implementation based on FPGA, instead of simulation.
- A new circuit design technique, which facilitates fast deployment of FastCFI systems, by automatically generating Verilog HDL for the application dependent component.
- A control flow graph compression technique is proposed to reduce hardware expense.
- An extensive evaluation on both popular security and performance benchmarks (never done before in FPGA-based work to the best of our knowledge).

We anticipate several application scenarios of FastCFI. FastCFI can be applied to various electronic systems, especially security-critical ones such as banks, public security systems, and military defense systems. These systems often have high real-time and security requirements, and can afford additional hardware resources such as FPGAs. FastCFI can also be applied in software supply chains to secure users of potentially vulnerable third-party software, the binaries of which can be analyzed, but do not allow code instrumentation. FastCFI has low latency (Section 6.5) and low overhead (Section 6.4), indicating its high real-time capability. On the one hand, the programs running on the processors will not be disturbed. On the other hand, once there is an attack, FastCFI can identify it immediately. FastCFI also has high precision (Section 6.3) without any false alarm. These properties ensure the system's security. Furthermore, FastCFI does not depend on code instrumentation and thus, makes the implementation be practical.

The rest of this article is organized as follows. Section 2 introduces the background on CFI and control-flow graph; Section 3 discusses previous work; Section 4 presents our proposed system design; Section 5 presents the implementation; Section 6 reports the experimental results and Section 7 concludes the article.

2 CFI AND CONTROL FLOW GRAPH

The specification of CFI is a **Control-Flow Graph (CFG)** of the target program, in which each node corresponds to one segment or block of instructions and each directed edge indicates a legal transition between instruction segments. In the example of Figure 1(a), the instructions are divided into seven segments, each of which corresponds to a node in Figure 1(b). The solid and dashed edges in Figure 1(b) indicate transitions by direct and indirect branch instructions, respectively. For example, the edge from node A to F implies that the branch instruction `bl 84a0` in A is taken and the software execution switches from A to F.

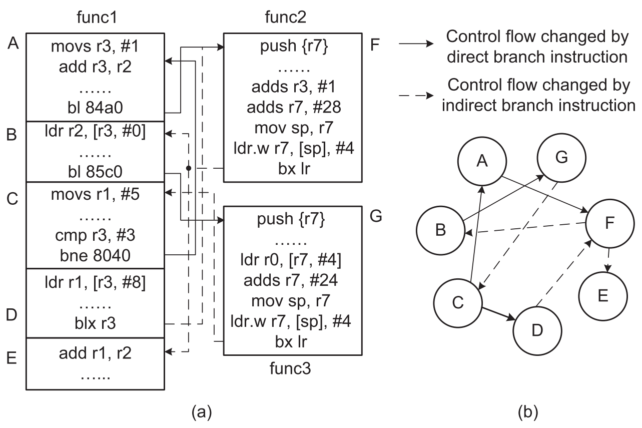


Fig. 1. Example of control-flow graph.

Once the program's CFG has been constructed, a CFI-technique can be applied to ensure the program is working as intended. For instance, transition $A \rightarrow B$ is illegal as there is no edge from A to B in the CFG. CFI for function returns can be stateful. For example, there are edges from F to both B and E. However, if the last instruction in A (which is a function call) invokes `func2`, which is included in F, then the last instruction (which is a function return) in F should only return to the instruction right after A, which is the first instruction in B. Therefore, function return $F \rightarrow B$ is legal while transition $F \rightarrow E$ is illegal. More details about CFG generation are described in Section 4.3.

3 PREVIOUS WORK

3.1 Software-based CFI

Early works on CFI were mostly realized by software implementations. The seminal work by Abadi et al. [1] proposed two code instrumentation approaches with and without using a protected shadow call stack, which have average overhead of 21% and 16%, respectively, on the SPEC 2000 benchmark.

Later, some CFI works targeted specific application scenarios. For example, one work specifies the target CFI verification device as smartphones [10]. The article focuses on CFI for ARM architecture and the smartphone operating system, and binary instrumentation is used. The overhead ranges from 1% to 21% for different benchmarks, and there are false positives raised by the approach.

When constructing **Control-Flow Graph (CFG)**, using commercial off-the-shelf binaries in the programs becomes a challenge. The work [43] proposed bin-CFI and strict-CFI for solving this. Code instrumentation is used in their approach. But this approach cannot identify function pointer overwrite attacks, thus it is a coarse-grained CFI. Its average performance overhead is 8.54%.

A CFI verification implementation called kBouncer is introduced in work [32], which implements the checking of the return addresses. kBouncer only allows a return address of a function to point at a call-preceded instruction, which is the instruction following a call instruction. Since kBouncer only handles ROP, the average overhead is 1%, but it is a coarse-grained CFI.

The work [5] proposed a two phase approach to detect ROP. Unlike most CFI, the approach in this article does not rely on code instrumentation, which makes it have only 2.6% overhead. It first analyzes the binaries offline, and finds out all the potential gadgets that may be used by ROP. The gadgets mean the code segments that may be used by the attackers to perform their attacks, which is defined in work [36]. Then for the online detection stage, it uses a sliding window for triggering the checking. When checking CFI, it checks if the control flow jumps to potential gadgets.

The combination of multiple coarse-grained CFI approaches is implemented in Reference [12]. It enforces CFI policy for all kinds of indirect branch instructions (indirect jumps, indirect calls, and function returns) and realizes an overall better solution. The disadvantage of CFI, which is the high performance overhead, is also mentioned in this article.

In contrast to software-based CFI implementations, our work uses hardware-based CFI. Using software for checking CFI inevitably incurs performance overhead. CFI implementations with the fine-grained and stateful policy have huge performance overhead [1, 10], while works with low performance overhead are only able to do incomplete CFI verification [5, 12, 32].

Moreover, to gather information from the target program, code instrumentation is needed in most of the cases [1, 10, 43]. Code instrumentation is one of the critical reasons for high performance overhead, and it is also not practical. For our work, by designing hardware circuits, we achieve fine-grained, stateful, and low performance overhead at the same time without using code instrumentation.

3.2 Hardware-based CFI

Recently, several hardware-based CFI approaches [3, 6, 9, 11, 13–18, 23–27, 29–31, 34, 39, 42] have been proposed, based on Intel Processor Trace [14, 16, 17, 29], performance counters [42], FPGA [9, 26, 27], and others [18]. Intel also proposed the **Control-flow Enforcement Technology (CET)** [21]. However, processors that support CET are still not available. Meanwhile, CET only implements the weakest form of CFI in that there's only a single class of valid targets and is too weak to protect against the larger class of code reuse attacks.

Besides these approaches, a great amount of the hardware-based CFI approaches require hardware modification [3, 6, 9, 11, 13, 15, 23–25, 30, 31, 34, 39]. Modifying hardware structure such as adding additional modules inside the processor's pipeline is not practical, since one needs to repeat the whole design flow, which is a tedious task.

Compared to previous hardware-based CFI, FastCFI has novelties in multiple directions. Firstly, FastCFI does not depend on code instrumentation. Previous hardware-based approaches leverage code instrumentation for getting more information [6, 11, 23, 24, 26, 27, 39]. However, this results in large overhead, and code instrumentation itself is also not secure and sometimes even impossible. Second, FastCFI has a low overhead compared to some previous works [13, 14, 16, 26, 27]. High overhead is unacceptable in some real-time applications. Also, not all the hardware-based CFI are fine-grained and stateful [9, 18, 26, 27, 42]. They may miss some attacks. In the operating system, false positives may delay all the processes, but some techniques used in previous works lead to this [18, 42]. Through results obtained in FastCFI we show that such cases are avoided in our CFI solution. Hardware-based CFI is harder to be implemented than software-based CFI due to the cost, difficulties in manufacturing, resources, and so on. A few previous works prefer using simulators for implementation [9, 11, 15, 23, 24], but this does not guarantee the functionality, because there are differences between simulation and real world conditions. We use FPGA to implement the hardware design. By taking advantage of existing devices in the processor, we avoid changing the structure of the processor and are able to build a real system for CFI verification.

4 THE PROPOSED SYSTEM DESIGN

4.1 System Platform

FastCFI is developed on a platform depicted in Figure 2. It is composed of an ARM Cortex-A9 processor and an FPGA. The CFI of a software execution on the ARM core is verified by the FPGA.

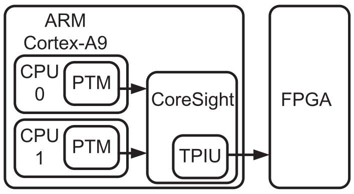


Fig. 2. System platform for the proposed CFI.

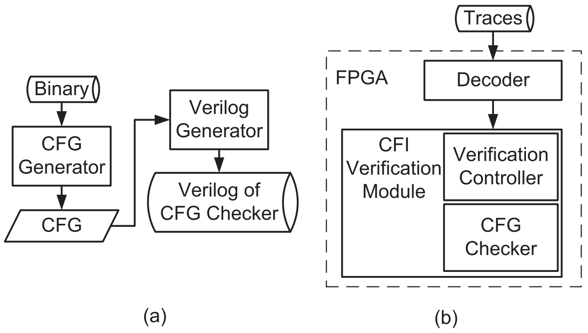


Fig. 3. System design overview: (a) offline CFG checker generator; (b) online CFI verifier.

Program Trace Macrocell (PTM) generates compressed control-flow traces according to instructions processed by the ARM core. The traces are generated in real time while instructions are executing in the ARM core. The CoreSight debug module in the ARM core can obtain traces from PTM and send the traces to FPGA through the **Trace Port Interface Unit (TPIU)**, which acts as a bridge between the trace data and a data stream. Therefore, FPGA can process the traces and do CFI verification parallely to the execution of the target software program to be verified. The key ideas of FastCFI can be applied to other platforms such as x86 architecture.

4.2 System Design Overview

The system design of FastCFI consists of an offline CFG checker generator and an online CFI verifier, as depicted in Figure 3. The CFG checker generator is a software that takes application software binary as input and generates the CFG checker design in Verilog. During online software execution, a trace captured through ARM CoreSight is first decoded to understand its semantics. The decoded trace data are then fed to the CFI verification module, which is composed of a verification controller and a CFG checker. Both the decoder and the verification module are implemented on the FPGA.

4.3 Offline CFG Checker Generator

To give the hardware verification circuits the correct execution information that can be represented by a CFG, target software binary has to be analyzed, and CFG should be extracted. Since we implement the CFG as a hardware circuit called CFG checker in the verification module for higher speed, the output of the CFG checker generator is the CFG checker's Verilog HDL file.

Given a software binary, the generator extracts CFG and generates the Verilog design of the CFG checker circuit. Then, the CFG checker is mapped on FPGA. The generator is able to help the fast implementation of CFI verification given a system to be protected, and only the vulnerable target binary is required.

We denote a sequence of instructions as $I_1, I_2, \ldots, I_{m1}, B_1, I_{m1+1}, I_{m1+2}, \ldots, I_{m2}, B_2, \ldots B_n \ldots$, where $B_1, B_2, \ldots B_n$ are branch instructions (e.g., jmp, call, ret, etc.) and the others are non-branch instructions. Then, the instruction sequence is partitioned into multiple segments $\{I_1, I_2, \ldots I_{m1}, B_1\}$, $\{I_{m1+1}, I_{m1+2}, \ldots, I_{m2}, B_2\}, \ldots$, each of which has a single branch instruction at its end. Each instruction segment forms a node in the CFG. In the sequel, we use CFG node and instruction segment interchangeably when the context is clear. Although the operating frequency of FPGA is typically much lower than the ARM core, the DFI verification at FPGA can keep pace with the target program execution for two reasons. First, branch instructions usually account for only a small portion of the entire code. Second, the trace packet generation at the ARM core is actually at a lower speed, closer to that of the FPGA.

By examining the source node and target node of each branch instruction, the generator can establish edges of the CFG. Recognizing the source node is trivial, but finding the target node can be quite difficult. The target address of a direct branch instruction is hardcoded in the binary and can be easily found. Indirect branches such as a function pointer are tricky, as their target addresses are stored in the registers. Such address can be a constant hardcoded somewhere in the binary, and can be recovered through tracing instructions. The more difficult case is where the target address depends on software input data at runtime. As such, it is almost impossible to find the address with an offline static analysis. Despite this difficulty, we find how to perform partial CFI check for unspecified target address and this technique will be described in Section 4.5. For the offline CFG checker generator, it does not generate the CFG nodes of dynamically linked library functions, system-level calls, or exception handlers. Like many previous influential works [1, 18, 27], control flow transitions inside these cases cannot be verified as the corresponding source code is not available. However, the control-flow transitions from the target software program to them can be checked by our system. Besides, our system does not have false alarm when the control-flow transitions get back to the target software program from them, as discussed in Section 4.6. Note that some of the virtual address randomization approaches such as **Address Space Layout Randomization (ASLR)** do not affect the results of our approach, since ASLR will not randomize the instruction's address [2]. Besides, randomization is not as strong as CFI in terms of security, since the former improves security opportunistically, while the latter provides certain security guarantees. Therefore, if any randomization policy is conflicted with CFI, then we can enable CFI and disable randomization for better security.

We developed a software program to automatically construct a CFG from binary code. The generator further creates Verilog description for the CFG checker circuit. Meanwhile, our framework is general. The static analysis tool, which is the tool that extracts the CFG based on the binary, can be replaced by any other tools such as IDA [20]. The main part of our system is based on CFG but not the binary. Therefore, our system is independent of the instruction set.

4.4 Trace Decoder

The decoder takes the software execution trace from TPIU as input, interprets its semantic and extracts information that is relevant to CFI. A trace consists of many packets, each of which is usually a few bytes. Two types of packets are of particular relevance to CFI, *Atom* and *Branch address* [7], which is simply called *Branch* subsequently. An *Atom* tells if a direct branch is taken or not, and indicates the case that an indirect branch is not taken. If an indirect branch is taken, then its target address is contained in *Branch*. Some other types of packets, such as *I-sync* [7], can periodically indicate the current instruction address.

The decoder extracts the following required information:

- Context ID that identifies the current program.
- The current program state.
- The current packet type: *Atom*, *Branch*, or *I-sync*, and so on.
- The current instruction address, which is obtained from *Branch*, or *I-sync*. Note that this information is not always available and the scenarios of its availability are complex. The starting address of a program is available at *I-sync*, which continues to provide the current address periodically.

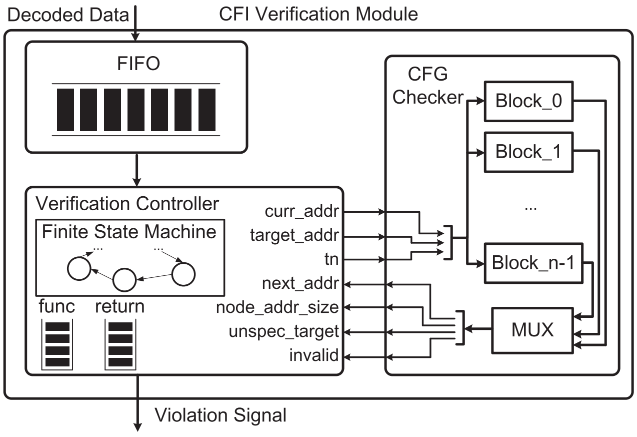


Fig. 4. Architecture of CFI verification module.

- T/N from *Atom*, where T indicates that a branch is taken and N means an indirect branch is not taken.
- Program exception and PTM buffer overflow information.

The TPIU channel in the ARM core has 32-bit bitwidth, which means 4 bytes of packets can be sent to FPGA in every clock cycle. When implementing, we design a 3-phase pipeline decoder to increase the throughput and match the speed of the TPIU.

4.5 CFI Verification Module

The CFI verification module examines if flow transitions in a software execution trace are consistent with transitions specified in the CFG constructed by us, which is embedded in the CFG checker. To do so, we need to obtain the source node and target node of a branch instruction from the execution trace. The source node of a branch instruction, which is equivalent to the current instruction address of the branch, is often unavailable in trace packets. In [27], it is acquired through code instrumentation. Without code instrumentation, identifying the source/current node is much more difficult. We solve this difficulty by using the periodically available instruction address information and tracking the other addresses by following the CFG.

Consider the example in Figure 5, which is the same example as Figure 1. Each gray number is the address of the first instruction in the instruction segment near the number. Suppose we know the current instruction address is at node A. The last instruction of A, `bl 84a0`, is a branch to node F, whose execution results in an *Atom* with T indicating that the branch is taken. Note that every direct branch has only one deterministic target when it is taken. To protect the direct branch, i.e., the code itself, Write⊕Execute [41] is used. Write⊕Execute is a technique applied in the operating system. It can ensure each memory page can only be writable or executable. Therefore, when it is applied, an attacker is not able to change the code and the target of each direct branch, since the code is executable. Hence, by observing T from trace decoder (the first trace packet in Figure 5) and examining the CFG in the CFG checker, we know that the software execution now moves to node F even if the current instruction address is not available at trace packets. The parts in Figure 5 related to this control-flow transition are shown as red. Since the transition from A to F changes the current function from `func1` to `func2`, `bl 84a0` is inferred as a function call. Therefore, `func2` should return to the next instruction of `bl 84a0` of `func1`, which is the first instruction of B. The

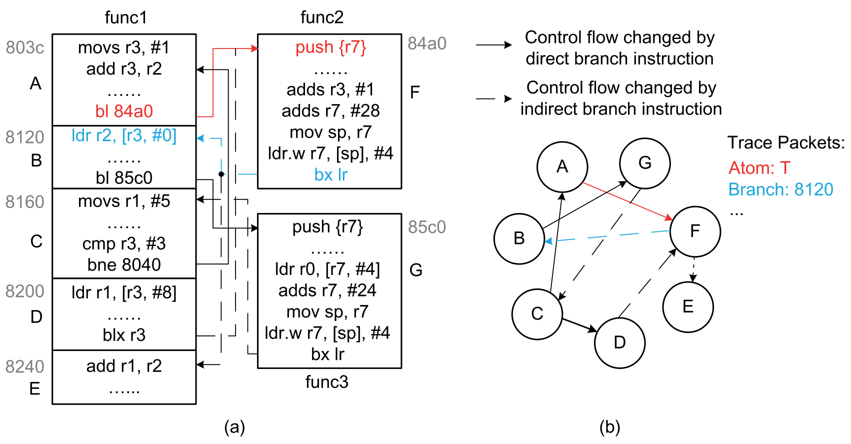


Fig. 5. Example of the transitions in the CFG.

last instruction of node *F* is function return, which is an indirect branch. Its execution leads to a *Branch* in decoded trace packet (the second trace packet in Figure 5). By receiving this *Branch*, we can be aware of the occurrence of a transition from *F*. The target address is contained in *Branch* and we can examine if it is consistent with the target node *B* in the CFG. The parts in Figure 5 related to this control-flow transition are shown in blue color.

The architecture of the CFI verification module is shown in Figure 4. Its key components, CFG checker and verification controller, are described as follows.

4.5.1 CFG Checker. The CFG checker is an FPGA circuit that contains CFG information and outputs specific CFG details for given execution trace information. It has *n* blocks, as shown in Figure 4, each of which corresponds to a node in the CFG. Assigning each CFG node in one block makes the CFG node search run in parallel, and this greatly increases the performance of FastCFI.

In detail, there are three main inputs to the checker circuit, all of which are from the decoded trace packets or earlier computations.

- *curr_addr*: current instruction address from trace or earlier calculation.
- *target_addr*: indirect branch target address decoded from *Branch*.
- *tn*: T/N information decoded from *Atom*.

The four main outputs are as follows:

- *next_addr*: the next program counter address after executing the branch of the current node according to CFG.
- *node_addr_size*: the start address and size of current node, and function size if the current node is the first node of a function, where the size is equivalent to difference between end and start addresses of a node/function.
- *invalid*: a binary signal whose assertion indicates that the *target_addr* does not conform to the *next_addr*.
- *unspec_target*: a binary signal whose assertion indicates that an indirect branch target depends on application input and is not specified in CFG.

Each block first checks if an input *curr_addr* is within the node corresponding to this block. If so, then the block is activated and always generates its *node_addr_size* output. The other outputs vary depending on three different types of blocks. Since each node in the CFG contains only one branch instruction at its end, the categorization of blocks is based on their branch instructions.

(1) **Direct branch**. An activated block with a direct branch generates *next_addr* according to input *tn*. If *tn* is *T*, indicating that the branch is taken, then the *next_addr* can be found in CFG and is hardcoded in the FPGA. Otherwise, the *next_addr* is the address of the next instruction.
(2) **Indirect branch with constant target**. When a block with an indirect branch is activated, the *target_addr* is compared with the possible *next_addr* from the CFG. If they are the same, then the *next_addr* is sent to output. Otherwise, signal *invalid* asserts.
(3) **Indirect branch with unspecified target**. In this case, *next_addr* is not specified in the CFG as the target address depends on software application input and cannot be identified in the offline analysis. Then, *next_addr* is output as *target_addr* and at the same time, signal *unspec_target* asserts.

Note that at most one block can be activated in the checker circuit. The checker outputs cover the following scenarios.

C1 **No output:** Current address is not in any CFG nodes.
C2 **There is output:** Current address is in one CFG node, whose start address is found. The start address of the next node is also found.
C3 **Output contains function size:** The current node is at the beginning of a function. The address range of this function is found.
C4 **No *invalid* or *unspec_target* assertion:** The actual control-flow is valid after executing the branch instruction in the current node. The next address after the current node is found so that the actual software execution position is located. Meanwhile, the current node has a direct branch or has an indirect branch with constant target, which is the same as the actual execution target address.
C5 ***invalid* asserts but no *unspec_target* assertion:** The current node has an indirect branch with constant target, which is different from the target address of the actual software execution.
C6 ***unspec_target* asserts but no *invalid* assertion:** The current node has an indirect branch with unspecified target in the CFG and the verification module is to perform other checks for CFI that will be discussed later in this section.

When increasing the number of the blocks in the CFG checker, the FPGA frequency can be affected. However, according to our experiment, the CFG checker is able to contain enough number of blocks without having to decrease the FPGA frequency to a level where the attacks cannot be identified.

4.5.2 Verification Controller. The verification controller takes the decoded trace packets as input, feeds the input to the CFG checker, and analyzes the checker results to locate the current instruction address, if not available from the trace packets, and performs CFI verification. It is mainly a finite state machine with state transition diagram provided in Figure 6. It also has a function stack that stores information about the current function, and a return stack that stores function return addresses. These two stacks are the critical parts for realizing the stateful attribute of the proposed system.

Overall, our proposed verification controller needs to follow the steps below to complete the CFI verification:

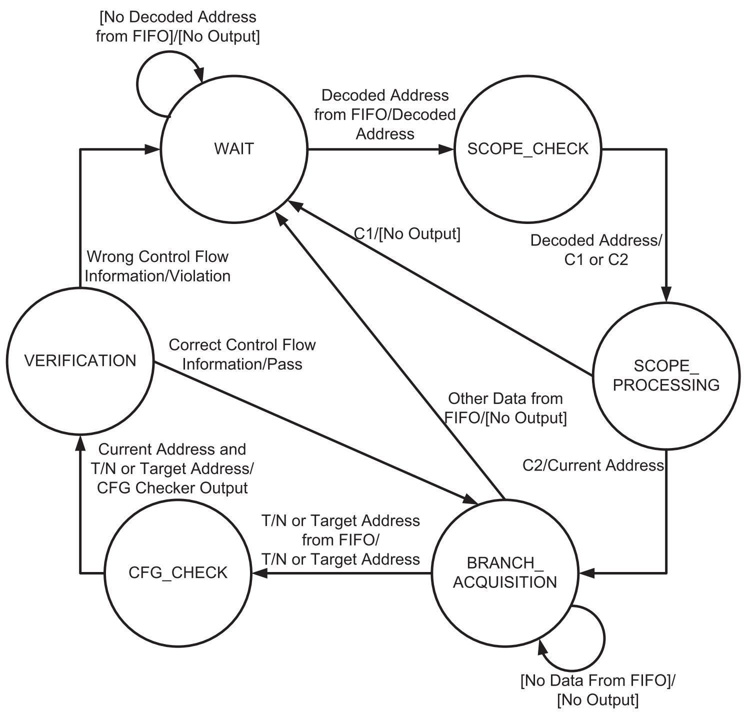

[No Decoded Address from FIFO]/[No Output]
WAIT
Decoded Address from FIFO/Decoded Address
SCOPE_CHECK
C1/[No Output]
Decoded Address/ C1 or C2
Wrong Control Flow Information/Violation
VERIFICATION
Correct Control Flow Information/Pass
SCOPE_ PROCESSING
Other Data from FIFO/[No Output]
C2/Current Address
Current Address and T/N or Target Address/ CFG Checker Output
T/N or Target Address from FIFO/ T/N or Target Address
CFG_CHECK
BRANCH_ ACQUISITION
[No Data From FIFO]/ [No Output]

Fig. 6. State transition diagram of the controller.

(1) To begin with, the controller should check if the current decoded trace packet belongs to the target program. If not, then the verification will be meaningless. Only if it is, the system can then begin the CFI verification, and the context ID of this trace packet will be recorded and only the trace packet with the same context ID will be processed.
(2) The controller should track the current CFG node where the program counter is. If not, then we cannot recover the control flow transition, since the source CFG node of such transition will never be generated by PTM.
(3) The controller should raise an alarm once there is a CFI violation, i.e., there is an invalid control flow transition. The violation can be found either directly from the output of the CFG checker, or the analysis of the controller.

Below, we give a detailed description about the finite state machine in the verification controller that realizes the proposed steps.

The controller operations start from the WAIT state, which attempts to capture the executing instruction address from decoded trace packets. This address provides a reference for the verification module to track the software execution location, and can be obtained from *Branch* or some *I-sync*.

Once an executing instruction address is acquired, the controller enters SCOPE_CHECK state, where the instruction address is sent to the CFG checker as *curr_addr* to tell if it is in the scope of CFG. After the scope checking is finished, SCOPE_PROCESSING state is entered where the controller analyzes the checking result and decides what to do next. If the result is C1, then the instruction address is not in the CFG and the next state is WAIT. One example of C1 is system level call. If the result is C2, then the controller records the context ID, which identifies the software execution to be verified, and then moves to state BRANCH_ACQUISITION.

At state BRANCH_ACQUISITION, the controller attempts to capture decoded *Atom* or *Branch*, and feeds *tn* or *target_addr* to the CFG checker. If the received trace packet is *I-sync* with current instruction address, then the controller switches to the WAIT state so as to update the reference instruction address. If branch information, *Atom* or *Branch*, is received, then it enters the CFG_CHECK state, where the CFG checker processes the *Atom* or *Branch* information, along with *curr_addr*.

When the CFG checking is finished, the controller switches to the VERIFICATION state. This state is to analyze the checking results and keep track of instruction execution location. If condition C3 occurs, then the function address range is pushed into the function stack. Because whether or not a state transits does not depend on C3 (and C4, C5, C6), it is not shown in Figure 6. Although C3 (and C4, C5, C6) might be a contribution factor for the state transition, but C3 (or C4, C5, C6) alone cannot cause the state transition. The function stack top always stores the address range of the current function. Condition C3 also implies that the previous node made a function call, and notifies the controller to push the return address onto the return stack.

Conditions C4, C5, and C6 are exclusive to each other, and will be discussed separately. Condition C4 indicates that CFI verification is passed without any violation. Then, the controller updates the current address with the next address and the state goes back to BRANCH_ACQUISITION. Condition C5 shows CFI violation, then the controller outputs a violation signal and goes back to the WAIT state.

Condition C6 means the target address of an indirect branch is not specified in CFG, which is a very difficult case for CFI verification as the CFG alone does not immediately tell if the actual target address is legal or not. Despite the difficulty, our controller continues to evaluate three sub-cases and detect as many CFI violations as possible.

(1) **Function return.** The controller compares the actual target address from a trace packet with the return address at the top of the return stack. If they are the same, then the current indirect branch is confirmed to be a function return, which is legal. Please note this check is stateful as it relies on history information stored in the return stack.
(2) **Branch within current function.** The controller checks if the actual target address is within the range of function address at the top of the function stack. If the check result is yes, then no violation signal is triggered.
(3) **Branch as a new function call.** If the actual target address is not in current function, then the only legal scenario is that a new function call is made. To verify if a new function call is indeed made, the controller sets a *contingency_check* flag, updates the current address with the next address and waits for the next BRANCH_ACQUISITION and CFG_CHECK result. The next CFG checking pays attention to the *contingency_check* flag. If the next result indicates C3 and the current address is the same as the new function entry address, then a new function call is confirmed. Evidently, this is also a stateful check.

Any other scenario beyond the above three is illegal and then a CFI violation signal is triggered. Although it is possible that a code segment in the same function is illegally reused and it passes the check, we have greatly restricted the attackers' capability. A similar approach for verifying the indirect branch with unspecified target is also used in Reference [23] and Reference [27]. Then, the controller updates the current address and goes back to either the BRANCH_ACQUISITION or WAIT state.

The verification is not only stateful but also fine-grained as its resolution is on each individual edge in the CFG. For the ease of description, we present the finite state machine with six states as above. In our implementation, we compact them into four states so that FPGA resource utilization is more efficient.

4.6 Sub-CFG Verification

Our system has a limitation that FPGA may not accommodate the entire CFG if a software program is too large. In this case, we must focus on a subgraph of CFG, which is more critical than the other parts, i.e., a sub-CFG is embedded in the CFG checker. Consider that the complete CFG is $G(V, E)$, where V and E are the sets of nodes and edges, respectively. The sub-CFG $G'(V', E')$ is constructed by selecting a subset of nodes $V' \subset V$ for G'. If and only if an edge $(A \rightarrow B) \in E$ has $A \in V'$ and $B \in V'$, then it is included in E'. An edge $(A \rightarrow B) \in E'$ can be verified by our system. An edge $(A \rightarrow B) \in E$ with $A \notin V'$ and $B \notin V'$ cannot be verified. The other scenarios are the following:

S1. $B \in V, B \notin V', A \in V'$, and $(A \rightarrow B)$ is a direct branch.
S2. $B \in V, B \notin V', A \in V'$, and $(A \rightarrow B)$ is an indirect branch with constant target address.
S3. $B \in V, B \notin V', A \in V'$, and $(A \rightarrow B)$ is an indirect branch with unspecified target address.
S4. $A \in V, A \notin V', B \in V'$, and $(A \rightarrow B)$ is a direct branch.
S5. $A \in V, A \notin V', B \in V'$, and $(A \rightarrow B)$ is an indirect branch with constant target address.
S6. $A \in V, A \notin V', B \in V'$, and $(A \rightarrow B)$ is an indirect branch with unspecified target address.

For S1–S3, when the state of the verification controller is at BRANCH_ACQUISITION, condition C1 occurs. To be conservative to avoid the false alarms, the state of the verification controller is changed to WAIT, and the function and return stacks are reset. When the stack needed by one time of CFI verification is empty, no CFI violation alarm is raised for this time. When S4–S6 occurs, C2 occurs and the state changes back to BRANCH_ACQUISITION. Our system cannot verify scenarios S3–S6, like the cases of dynamically linked library functions in many influential works [1, 18, 27], but can still verify scenarios S1 and S2. In cases of S1 and S2, the CFG checker circuit block for source node $A \in V'$ contains the next address although the target node B is not in V'. Therefore, CFI of these edges can still be verified by our system.

4.7 CFG Compression

According to Section 4.5, the direct branches are safe when the Write⊕Execute [41] technique is applied. The blocks corresponding to CFG nodes with direct branches are only used by the verification controller for tracking the current node and instruction address. In this case, we found one scenario where the current node does not have to be precisely tracked by the verification controller, and therefore the CFG nodes and the CFG checker blocks can be trimmed to reduce hardware expense.

Let us consider a simple case. If there is a set $\mathcal{S}$ of CFG nodes with direct branches, every CFG node in $\mathcal{S}$ has no path to other CFG nodes outside this set, then once the current instruction address is at the address range of one node in $\mathcal{S}$, all the subsequent control-flow transitions can be ignored. This is because all the later control-flow transitions must be from direct branches and they do not have to be checked. Then the CFG checker blocks corresponding to the nodes in $\mathcal{S}$ can be removed and thus, the hardware expense is saved.

However, this simple case rarely exists in a realistic program. In a realistic program, there are often indirect branches (such as function return) that have to be checked. A set $\mathcal{S}$ of CFG nodes with direct branches is likely to have at least one path to CFG nodes with indirect branches. Therefore,

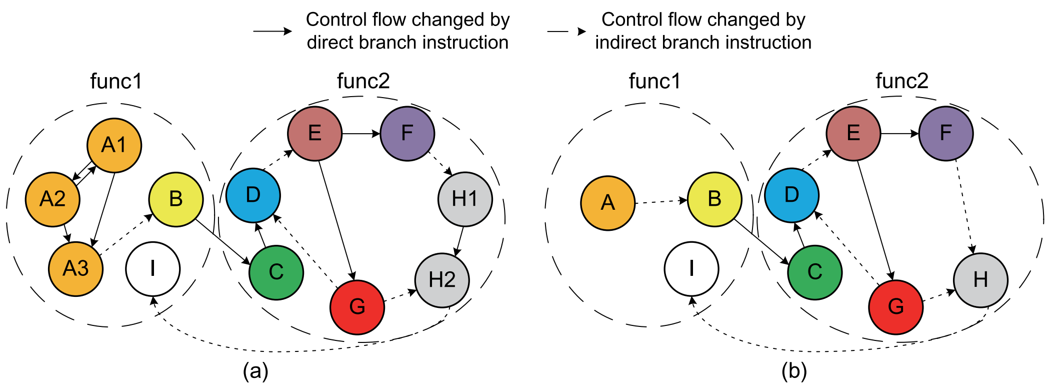

Control flow changed by direct branch instruction
Control flow changed by indirect branch instruction
func1
func2
A1
A2
A3
B
I
D
E
F
H1
H2
C
G
(a)
func1
func2
A
B
I
D
E
F
C
G
H
(b)

Fig. 7. An example of CFG optimization.

we allow one additional CFG node k with an indirect branch to be included in $\mathcal{S}$, and the possible outgoing edges from $\mathcal{S}$ are only from node k. In this way, when the current instruction address is in the address range of one node in $\mathcal{S}$, all the later direct branch control-flow transitions can be ignored, and only indirect branch control-flow transitions have to be checked by CFI verification module. The indirect branch transitions are only from node k that contains the only indirect branch in $\mathcal{S}$. All the nodes in $\mathcal{S}$ can be replaced by one new node N named *coalesced node*, of which the address range is the union of all the address ranges of the nodes in $\mathcal{S}$. The edges to and from $\mathcal{S}$ are also replaced by the edges to and from N. The CFG checker blocks corresponding to the nodes in $\mathcal{S}$ are replaced by only one block corresponding to N, thus, the hardware consumption is reduced. Once the instruction address is in the address range of N, CFI verification module can ignore all the decoded trace packets from direct branches, until there is a decoded trace packet from an indirect branch, which indicates the indirect branch k is taken. Then, the control-flow from the indirect branch k is checked.

Meanwhile, the additional indirect branch k has to be unconditional, because if it is conditional and is not taken, PTM would generate an *Atom* trace packet for this control-flow transition, which cannot be distinguished from the *Atom* trace packets of the direct branches.

However, if there are two or more additional CFG nodes with indirect branches in $\mathcal{S}$, when the current instruction address is in the address range of one node in $\mathcal{S}$ and there is an indirect branch control-flow transition, then there is no clue about which indirect branch in $\mathcal{S}$ is the source of this control-flow transition. Then, there is no way to check if the control-flow transition is legal or not. Therefore, only one additional CFG node with the indirect branch is allowed to be included in $\mathcal{S}$.

Based on this idea, an algorithm is developed to further compress the CFG and the CFG checker by replacing multiple CFG nodes by one coalesced node, and each coalesced CFG node still corresponds to one block in the CFG checker. The CFG checker can spend less hardware resource without losing any precision.

Consider the example in Figure 7, where Figure 7(a) represents the CFG and Figure 7(b) represents the compressed CFG. In Figure 7(a), the CFG nodes are from two functions `func1` and `func2`, which are marked by the dashed circles. The legal control-flow transitions by direct branches and indirect branches are illustrated by the solid and dotted arrows, respectively. One can easily figure out when the control-flow transits between node $A1$, $A2$, and $A3$, there are only direct branch control-flow transitions, thus, PTM will not generate any *Branch* but only *Atom* trace packets until the current control-flow is $A3 \rightarrow B$. Since the direct branches do not have to be checked and only $A3$ contains an indirect branch, node $A1$, $A2$, and $A3$ can be replaced by one node A shown in Figure 7(b). The CFG checker block corresponding to A is basically generated in the same way described in Section 4.5, except:

- SP1 The block is activated if an input *curr_addr* is within the range of instructions' addresses of one of the corresponding nodes before replacement.
- SP2 Only if there is *target_addr* decoded from *Branch*, the block begins checking. Otherwise, the block outputs *curr_addr* to *next_addr* port. The output port *node_addr_size* is the start address and instruction size (in bytes) of the node with the indirect branch.

In this way, if the current instruction address is at $A1$, $A2$, or $A3$, then all the T/N information from *Atom* trace packets are ignored until $A3 \rightarrow B$, which lets the decoder decode a *target_addr* by a *Branch* trace packet.

Assuming that there is a set of CFG nodes $\mathcal{S}$, $\mathcal{S}$ can be replaced by one coalesced CFG node if and only if $\mathcal{S}$ satisfies the following requirements:

- R1 $\mathcal{S}$ contains one and only one CFG node k with an indirect branch, and this indirect branch should be unconditional.
- R2 Except node k, there is no path from each node in $\mathcal{S}$ to another node, which is not in $\mathcal{S}$.
- R3 There is no node in $\mathcal{S}$ of which the instructions are at the beginning of one function.

Using Figure 7(b) as an example for illustrating the requirements, $\{H1, H2\}$ can be replaced by H. Note that after the replacement, although $H2$ no longer exists, illegal transition $F \rightarrow H2$ can still be identified, because the indirect branch target is checked by the block corresponding to F but not $H2$. Set $\{E, F\}$ does not satisfy R2 and cannot be replaced. If they are replaced, once the current control-flow is $E \rightarrow G$, then it is ignored by the CFG checker, because this control-flow is from a direct branch. Then, the current instruction address kept by the verification controller is not updated to G, and the legal control-flow $G \rightarrow D$ is wrongly regarded as the control-flow from set $\{E, F\}$ to D, which is illegal. Thus, a false alarm is raised. Meanwhile, if C is at the beginning of the function *func2*, then set $\{C, D\}$ does not satisfy R3 and cannot be replaced by a coalesced node, since the verification controller pushes the function address range to the function stack when the current instruction address is in the address range of C. If $\{C, D\}$ is replaced, then the verification controller cannot tell if the current instruction address is at C or D and, thus, cannot tell if the function stack should be pushed or not.

Given a CFG graph G, the algorithm to find the compressed graph G' is shown as Algorithm 1. The first part (lines 1–11) of the algorithm is to find if a CFG node can be a candidate, which can be included in a set to be compressed. Line 1 construct a new graph G^d without edges representing indirect control-flow transitions. This new graph can be used to find the set $\mathcal{P}$ of all the nodes that each node n can reach through the direct control-flow transitions. Lines 4 and 5 correspond to this part. Each node n is a candidate only if n is likely to be included into a set to be compressed without violating any of the requirements (R1 to R3). First, if a node n can reach multiple nodes $k_1, k_2..$ with indirect branches, then n cannot be included into any set $\mathcal{S}$ to be compressed. This is because $\mathcal{S}$ can only contain one node k_i with an indirect branch according to R1. If $n \in \mathcal{S}$, then there must be paths from n to another node k_j that is outside $\mathcal{S}$, and this violates R2. Second, n should have no path to a node m that is at the beginning of one function. Otherwise, if $n \in \mathcal{S}$ and $m \in \mathcal{S}$, then R3 is violated, and if $m \notin \mathcal{S}$, then R2 is violated. In summary, only if node n passed the two checks mentioned above (lines 6 and 8) is n chosen as candidate and appended into candidate set C (line 11).

ALGORITHM 1: The algorithm to construct compressed CFG.

```
Input: $G$:- CFG graph
Output: $G'$:- Compressed CFG graph
1  Construct $G^d = G$, and remove all the edges $n \rightarrow m$ in $G^d$ if node $n$ contains an indirect branch;
2  Construct a candidate set $C = \emptyset$;
3  for each node $n$ in $G^d$ do
4      Do depth first search (DFS) in $G^d$ from $n$;
5      Construct a set $\mathcal{P}$ containing all the searched nodes during the DFS;
6      if there exists a node $m \in P$ that is at the beginning of a function then
7          Continue;
8      else if !(there is only one $k \in P$ with an indirect branch, and this indirect branch is unconditional)
       then
9          Continue;
10     else
11         $C \leftarrow C \cup \{n\}$;
12 Construct a graph $G^{dr}$, where the nodes are the same as $G^d$, with all the edges reversed;
13 for each node $n$ in $G^{dr}$ do
14     if $n$ contains an unconditional indirect branch then
15         Do DFS in $G^{dr}$ from $n$; Stop searching the edge $m_i \rightarrow m_j$ if $m_j \notin C$;
16         Construct a set $\mathcal{S}$, which contains all the searched nodes during the DFS this time;
17         Add a new node $N$ to $G$;
18         Add an edge $l \rightarrow N$ if there exists $l \rightarrow m$ in $G$ where $m \in \mathcal{S}$;
19         Add an edge $N \rightarrow l$ if there exists $m \rightarrow l$ in $G$ where $m \in \mathcal{S}$;
20         $\forall m \in S, G = G - m$;
21 Output $G\prime = G$;
```

The second part (lines 12–21) of the algorithm is to compress the CFG. According to R1, we can construct the set $\mathcal{S}$ to be compressed starting from each node n with an unconditional indirect branch (lines 13 and 14). By constructing a new graph G^{dr} that is the same as G^d, with all the edges reversed, we can search all the nodes at n's predecessor side by depth first search (lines 15-16). However, we should stop searching a branch $m_i \rightarrow m_j$ once m_j is not the candidate, which means m_j cannot be included into $\mathcal{S}$. After we get the set $\mathcal{S}$ that can be compressed, new node N is added, which is used to replace the nodes in set $\mathcal{S}$ (lines 17–20). After all the possible replacement, the compressed graph is returned to the output (line 21).

A complete example is shown in Figure 8, where Figure 8(a), (c), and (e) are G, and Figure 8(b) and (d) are G^{dr}. In Figure 8(a), nodes C and D are two nodes with unconditional indirect branches in G. The first part of Algorithm 1 finds $C = \{B, C, D\}$. The second part of Algorithm 1 constructs G^{dr} as Figure 8(b). Since D is a node with an unconditional indirect branch, DFS is applied starting from D, shown as blue parts in Figure 8(b). Because $A \notin C$, DFS stops searching edge $B \rightarrow A$, and thus, $\mathcal{S} = \{B, D\}$. Then, node E is added to replace B and D, shown in Figure 8(c). Similarly, DFS is also applied starting from C in G^{dr} shown in Figure 8(d), and the final compressed CFG G is shown in Figure 8(e).

After G' is created, the blocks in the CFG checker are automatically generated the same way as the CFG checker generated from G in Section 4.5. The behavior of each block in the CFG checker is still the same, except the additional behaviors SP1 and SP2 for the blocks corresponding to the node that is generated during compression.

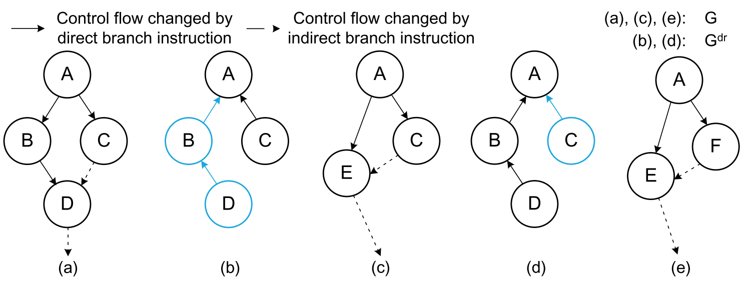

Control flow changed by direct branch instruction
Control flow changed by indirect branch instruction
(a), (c), (e): G
(b), (d): G^dr
A
B
C
D
E
F
(a)
(b)
(c)
(d)
(e)

Fig. 8. An example for explaining CFG optimization algorithm.

5 HARDWARE IMPLEMENTATION

All the hardware modules are described in Verilog HDL and synthesized to FPGA implementation. This section is focused on the CFG checker and the pipelined trace decoder, which are the two relatively sophisticated modules compared to the others. Most of the hardware modules are applicable for general software applications, except the CFG checker design, which is specific to each individual software application. To mitigate the design overhead of the CFI system for each application due to different CFG checker requirements, we develop an automated Verilog generation technique for CFG checkers. As such, there is no need to manually write Verilog code for each application, which can be quite time consuming. Although Verilog code can also be obtained through **high-level synthesis (HLS)** of C code, manually writing C code for the CFG check of each application is not efficient either.

5.1 Automatic Verilog Generation for CFG Checker

The CFG checker is a Verilog module where each CFG node is implemented as a block. The block is a hardware circuit that can check if the control-flow from its corresponding CFG node is valid or not. Different application softwares have different CFGs and thus require different CFG check implementations. However, in a CFG checker, there are only three types of blocks corresponding to the CFG nodes with three kinds of branch instructions, which are direct branches, indirect branches with constant targets, and indirect branches of which the targets are unspecified. Different CFGs only need to be implemented by using these three types of blocks, and blocks of the same type share the same structure with different parameters. Therefore we can prepare three Verilog description templates for all the three types of blocks. Given a CFG, each of its nodes can be mapped to one of the templates and thereby the Verilog code of an entire checker can be automatically generated. The Verilog generator is generic and is independent of the instruction set, since its input is the CFG instead of the executable binary, and the CFG only contains the control-flow transitions without detailed instruction information. The software program is designed by us and written in Python, and it is different from general HLS programs that can synthesize a general level language program into a hardware circuit. The software program designed by us is specifically designed only for generating the CFG checker, with the use of 3 Verilog templates. The input of our software program is the CFG but not high level language such as C.

The pseudo codes of the three Verilog templates are shown in Figure 9, 10, and 11. The parentheses of the `function size` in Figure 9, 10, and 11 mean that only when the corresponding CFG node is the entry of a function, function size is included. For input `tn`, “0” indicates N (the branch is not taken) and 1 implies T (the direct branch is taken). For `invalid` and `unspec_target`, “0” and “1” means false and true, respectively.

```
 1  if(curr_addr in the range of this CFG node){
 2      node_addr_size=node start address, node size, (function size);
 3      if(tn==0){
 4          next_addr=the address of the closest instruction after curr_addr;
 5      }else{
 6          next_addr=target address of this direct branch;
 7      }
 8      invalid=0;
 9      unspec_target=0;
10  }
```

Fig. 9. The Verilog pseudo code template for a CFG checker block with direct branch.

```
 1  if(curr_addr in the range of this CFG node){
 2      node_addr_size=node start address, node size, (function size);
 3      if(tn==0){
 4          next_addr=the address of the closest instruction after curr_addr;
 5          invalid=0;
 6      }else if(target_addr is one of the valid addresses){
 7          next_addr=target_addr;
 8          invalid=0;
 9      }else{
10          invalid=1;
11      }
12      unspec_target=0;
13  }
```

Fig. 10. The Verilog pseudo code template for a CFG checker block with indirect branch with constant target.

```
 1  if(curr_addr in the range of this CFG node){
 2      node_addr_size=node start address, node size, (function size);
 3      if(tn==0){
 4          next_addr=the address of the closest instruction after curr_addr;
 5      }else{
 6          next_addr=target_addr;
 7      }
 8      invalid=0;
 9      unspec_target=1;
10  }
```

Fig. 11. The Verilog pseudo code template for a CFG checker block with indirect branch with unspecified target.

When the current address is in the range of a CFG node corresponding to a block, the condition of the *if* statement (line 1 in Figure 9, 10, and 11) holds, and the start address of the node and the instruction size (in bytes) of the node are output to port `node_addr_size`. If this CFG node is a function entry, then `node_addr_size` of the corresponding block also contains the instruction size of the function (line 2 in Figure 9, 10, and 11).

If a CFG node contains a direct branch, then the corresponding block is realized as shown in Figure 9. When the direct branch is taken, PTM sends an *Atom* trace packet that can be decoded as *T*, and the verification controller inputs 1 to `tn`. Then, the next instruction address `next_addr` is updated as the target address of the direct branch (line 6). Otherwise, if the direct branch is not taken, then *N* is decoded from the trace decoder and the verification controller inputs 0 to `tn`, and this makes the block output the address of the instruction right after this direct branch as the next instruction address (line 4). Since direct branch is assumed to be immune from attacks, `invalid` is outputted as 0. The target of the direct branch is not unspecified, so `unspec_target` is set to 0.

If a CFG node contains an indirect branch with constant targets, then the corresponding block is realized as shown in Figure 10. When the indirect branch is taken, PTM sends an *Branch* trace packet that can be decoded as a target address, and the verification controller inputs this target address to `target_addr`. The block begins to check if the target address is a valid target or not. If it is valid, then the next instruction address is updated as this target address, and `invalid` is 0, since this control-flow is valid (lines 7 and 8). Otherwise, this control-flow is invalid (line 10). If the indirect branch is not taken, then an *Atom* trace packet is generated from PTM and *N* is decoded from the trace decoder. The verification controller inputs 0 to `tn`, and this makes the block output the address of the instruction right after this direct branch as the next instruction address (line 4). The target of the indirect branch is not unspecified, so `unspec_target` is output as 0.

If a CFG node contains an indirect branch with unspecified target, then the corresponding block is realized as shown in Figure 11. Figure 11 is similar to Figure 10 except that the target address input is not checked by this block, but directly output as the next instruction address. The target address is checked by the verification controller described in Section 4.5.2. Meanwhile, `unspec_target` is set to 1 due to the unspecified target (line 9).

For the coalesced CFG node, according to Section 4.7, the outgoing edges are from the unconditional indirect branch in the coalesced node, so the Verilog template for the CFG checker block of coalesced CFG node is based on the block with the indirect branch, which is Figure 10 or 11, depending on if the indirect branch has constant targets or not. The only difference is that the Verilog template for the coalesced CFG node ignores all the *T/N* information until there is `target_addr` decoded according to Section 4.7, which means the indirect branch in the coalesced CFG node is taken.

Each block has its own output, and the MUX is implemented by bitwise OR to select the checker output among all blocks, as shown in Figure 4. For each block, if the current address is not in the range of the CFG node corresponding to this block, then all the output bits of this block are 0, indicating no output from this block. Because the current address belongs to at most one block, all the outputs of the other blocks are 0, then the checker output is always from the block where the current address belongs to. If the checker output is 0, then the current address is out of the ranges of the CFG nodes represented by all blocks (condition C1).

The CFG checker design is written in Verilog and compiled by Quartus Prime 17.1 to generate the circuits on FPGA. In general, the compilation tool strives to achieve optimized logic circuits with high performance and low resource use. However, optimizing a large number of blocks is more difficult than optimizing fewer blocks. Therefore, we develop a hierarchical approach that groups blocks into small Verilog modules. Note that the group is a different concept from the CFG set to be replaced during CFG compression in Section 4.7. The CFG checker is never trimmed during the hierarchical approach. For example, in Figure 12, the CFG checker has 6 blocks, *B*0 to *B*5, which are grouped into two small modules. Each small module takes the checker input to all of its internal blocks, and selects an output among all of its internal blocks. In this way, the compiling optimization is directed to perform in a hierarchical manner to reach different resource use and compiling time tradeoffs.

5.2 Pipelined Trace Decoder

We propose the implementation of a fast decoder for the traces generated by ARM CoreSight. Although this decoder is designed for ARM CoreSight, it does not affect the generality of overall FastCFI. Since ARM CoreSight works in real time in parallel to the target program to be verified,

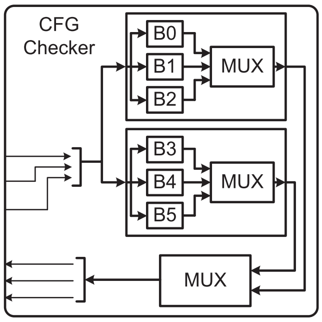


Fig. 12. An example of grouping blocks in a CFG checker.

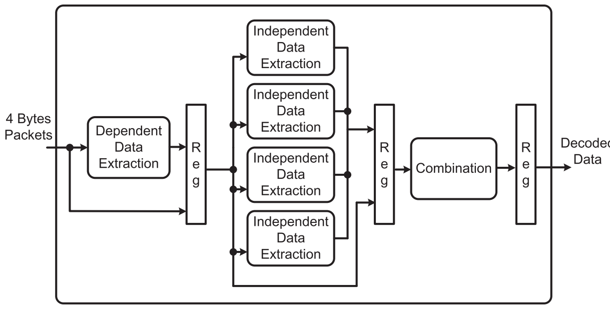

4 Bytes Packets
Dependent Data Extraction
Reg
Independent Data Extraction
Independent Data Extraction
Independent Data Extraction
Independent Data Extraction
Reg
Combination
Reg
Decoded Data

Fig. 13. Structure of the pipelined decoder.

to match the speed of ARM CoreSight and reduce the detection latency, the decoder is required to work at a high speed. The TPIU channel in the ARM core is 32-bit wide, which means 4 bytes of packets can be sent to FPGA in every clock cycle. To match this speed, we must design a decoder that can decode 4 bytes in each clock cycle. A naïve design is to have the same 4 units of decoding modules, and each module can decode 1 byte. However, each packet may contain multiple consecutive bytes, and the decoding of the latter byte may depend on the decoded information from the former byte. For example, the highest bit of the first byte in *Branch* trace packet indicates if there is the second byte in this *Branch* packet. For the naïve design, if the decoding module unit *A* is decoding the first byte of *Branch* trace packet, then decoding module unit *B* for decoding the next byte needs to wait the result of *A* to identify if the byte for *B* is the second byte of this *Branch* packet or the first byte of a new trace packet. Moreover, not only the bytes in one *Branch* packet are dependent, but also the latter *Branch* packet depends on the former *Branch* packet, because each *Branch* packet only carries the value of the changed bits compared to the instruction address decoded from the most recent *Branch*. Due to the dependency, when using the naïve design, the 4 units need to be designed in a serial chain. However, such design is vulnerable to timing errors when implemented under our experiment setup, since the serial chain structure has a long critical path. Meanwhile, for each byte of a trace packet, not all the bits have dependency relationships. For example, the decoding of the byte next to the first byte of *Branch* trace packet only depends on the highest bit of the first byte of *Branch* trace packet. Based on this idea, we design a pipelined decoder to perform decoding operations without timing errors.

The structure of the pipelined decoder is illustrated in Figure 13. The pipeline has three stages: dependent data extraction, independent data extraction, and combination. The first stage decodes all the dependent information in each byte, so the four bytes are decoded serially in this stage. However, this stage only decodes the necessary bits in each byte that have the dependency, but not all 8 bits in each byte like the naïve design. Because there are fewer bits needed to be decoded, the circuit in this stage is simpler than the structure of the naïve design, thus, there is no timing error in this stage. Then, the second stage extracts the independent data of every byte (such as certain bits of the branch target address), based on the decoded information of the first stage (such as the trace packet type of each byte). For example, we may extract bit 13 to bit 7 of target address from the second byte of *Branch*. The second stage can be performed in parallel, since only the independent part of the 4 bytes are extracted. Compared to the naïve design, which extracts the data using serial combinational logic, using parallel circuits addressed the long critical path problem, and the timing error is also avoided in this stage. After that, the third stage combines all the information we have extracted from the 4 bytes of packets. To accommodate *Branch* trace packets, which may depend on previous *Branch* trace packets, this stage is also serial. Since this stage only does information combination serially but not the complete decoding procedure serially, the serial circuit can be implemented by fewer gates and has a shorter critical path than the serial structure of the naïve design, thus, there is also no timing error in our experiments.

Note that the pipeline registers also carry the raw packet information and transmit it through different stages. For example, for the second stage, not only the output of the first stage, but also the raw packets are needed.

6 EXPERIMENTS AND RESULTS

6.1 Experiment Setup

All our experiments were run and measured on an Altera DE1-SoC board, containing a Cyclone V FPGA working at 50 MHz and an ARM Cortex-A9 dual core processor working at 1 GHz on which we loaded a Linux kernel. The Altera DE1-SoC board with the ARM core and the FPGA is a very common and affordable platform. In addition, we use Quartus Prime 17.1 [22] for Verilog compilation and FPGA layout synthesis, and Signal Tap Logic Analyzer for FPGA signal monitoring. All synthesis settings of Quartus remain default. The Verilog compilation is done on a desktop with an Intel 3.8 GHz CPU and 16 GB RAM.

6.2 CFI without Code Instrumentation

All the experiments are performed with no instrumentation to the target program. In the proposed hardware-based CFI approach, the information required for CFI violation identification is gathered from the output of TPIU and computed by using CFG. The results show that even without any code instrumentation, it is feasible to realize fine-grained and stateful hardware-based CFI with negligible performance overhead. Avoiding code instrumentation, not only improves performance but more importantly, enables a practical solution eliminating implementation difficulties and the related potential security issues.

6.3 Security

We use RIPE [35, 40] to evaluate the effectiveness of FastCFI. RIPE is a popular benchmark that has been used frequently in previous works [14, 16] for evaluating control-flow defenses, and it is the dominating benchmark of control-flow attacks. However, RIPE is designed for Intel processors, and does not directly run on our ARM platform. There are numerous processor architecture specific assembly and shell codes in RIPE, which we had to modify for the ARM processor. However, the modifications do not change how the attacks perform.

For example, one of the modifications is that the shellcode used in RIPE has been replaced by the shellcode for ARM, which is from Reference [37]. Also, many attacks are based on changing the return address of the function called `perform_attack`. The pointer of the return address in the unmodified RIPE program is obtained by `__builtin_frame_address(0)+4`. However, for the ARM compiler, sometimes the return address is not only stored at the pointer `__builtin_frame_address(0)+4`, but also at a different pointer that can be calculated by the value stored in register `r7`. For RIPE program compiled by the ARM compiler, the function where ROP attack performs uses the latter pointer for getting the return address for function return, thus, we modify the ROP attack target from the former return address pointer to the latter one in RIPE program. After this modification, the same buffer overflow approach that is used by original RIPE program is used to change the return address and perform the attack.

It is hard to port all RIPE functions to ARM because of the engineering difficulties. Most attacks in RIPE are based on buffer overflow related to 10 methods: `memcpy()`, `strcpy()`, `strncpy()`,

Table 2. Security Performance for Different Attack Methods

No.	Overflow Technique	Attack Code	Target Code Pointer	Location	Identify?
1	direct	createfile	ret	stack	✓
2	direct	createfile	funcptrstackvar	stack	✓
3	direct	createfile	structfuncptrstack	stack	✓
4	direct	createfile	funcptrheap	heap	✓
5	direct	createfile	structfuncptrheap	heap	✓
6	direct	createfile	structfuncptrbss	bss	✓
7	direct	createfile	funcptrdata	data	✓
8	direct	createfile	structfuncptrdata	data	✓
9	indirect	createfile	ret	stack	✓
10	indirect	createfile	funcptrstackvar	stack	✓
11	indirect	createfile	funcptrstackparam	stack	✓
12	indirect	createfile	funcptrheap	stack	✓
13	indirect	createfile	funcptrbss	stack	✓
14	indirect	createfile	funcptrdata	stack	✓
15	indirect	createfile	ret	heap	✓
16	indirect	createfile	funcptrstackvar	heap	✓
17	indirect	createfile	funcptrstackparam	heap	✓
18	indirect	createfile	funcptrheap	heap	✓
19	indirect	createfile	funcptrbss	heap	✓
20	indirect	createfile	funcptrdata	heap	✓
21	indirect	createfile	ret	bss	✓
22	indirect	createfile	funcptrstackvar	bss	✓
23	indirect	createfile	funcptrstackparam	bss	✓
24	indirect	createfile	funcptrheap	bss	✓
25	indirect	createfile	funcptrbss	bss	✓
26	indirect	createfile	funcptrdata	bss	✓
27	indirect	createfile	ret	data	✓
28	indirect	createfile	funcptrstackvar	data	✓
29	indirect	createfile	funcptrstackparam	data	✓
30	indirect	createfile	funcptrheap	data	✓
31	indirect	createfile	funcptrbss	data	✓
32	indirect	createfile	funcptrdata	data	✓
33	direct	returnintolibc	ret	stack	✓
34	direct	returnintolibc	funcptrstackvar	stack	✓
35	direct	returnintolibc	structfuncptrstack	stack	✓
36	direct	returnintolibc	funcptrheap	heap	✓
37	direct	returnintolibc	structfuncptrheap	heap	✓
38	direct	returnintolibc	structfuncptrbss	bss	✓
39	direct	returnintolibc	funcptrdata	data	✓
40	direct	returnintolibc	structfuncptrdata	data	✓
41	direct	rop	ret	stack	✓
42	-	-	-	-	No False Alarm
SP1	-	-	-	-	✓
SP2	-	-	-	-	✓

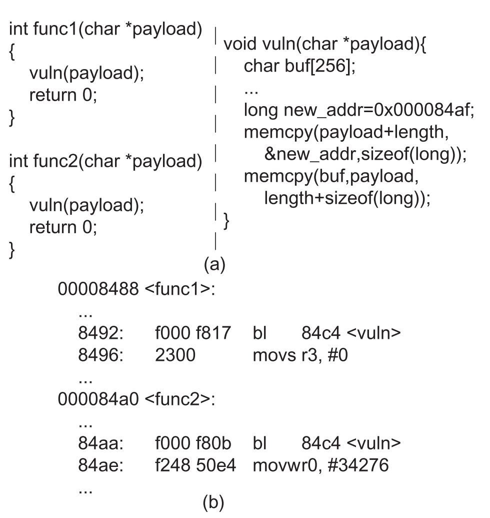
sprintf(), snprintf(), strcat(), strncat(), sscanf(), fscanf(), and homebrew(). All these methods can be performed on Intel processors, but only memcpy() and homebrew() work for our modified RIPE on ARM-based Linux system, and using other methods can cause the program to crash. However, the way a buffer overflow happens does not affect how an attack is performed. The difference is which API is used to copy data to the buffer. Therefore, we focused on only the attacks related to memcpy() and homebrew(). In total, we recovered 41 attacks (which can run successfully on ARM), including both ROP and JOP attacks, as shown in Table 2 (Rows #1–41). To assess the precision (i.e., no false positive), we also added a new function (Row #42) in RIPE and let it run without attack.

The results are shown in Table 2. "Overflow Technique" indicates whether the attack target can be directly reached by sequentially overflowing from a buffer. "Attack Code" is the code to complete what the attacker aims at. "Target Code Pointer" indicates what the attack target is. "Location" represents where the attack target is [40]. The results in Table 2 show that all these attacks can be identified by FastCFI. In addition, FastCFI does not report any false positives (for the newly introduced function with no attack). The results are the same with and without CFG compression.

Fine-grained, stateful attacks.. We also designed two special attacks not included in RIPE, as shown in the last two rows of Table 2.

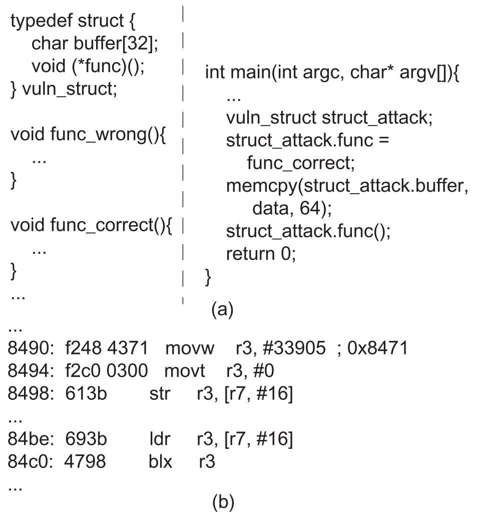
SP1 is a stateful attack that cannot be detected by stateless CFI techniques. As shown in Figure 14(a), in SP1, there is a function vuln that may be called by function func1 or func2. So in the CFG, the node with function return of vuln has edges to nodes in both func1 and func2. However, only one of them is valid each time vuln is called. If func1 calls vuln, then vuln can only return to func1. In our test, we use buffer overflow to change the return address of vuln to func2, even if it is called by func1. Our experiment shows that FastCFI can easily identify this attack with and without CFG compression. However, stateless CFI such as in References [18, 42] and the coarse-grained approach in Reference [16], would not be able to identify this attack.

SP2 is a fine-grained attack. In SP2, the attack changes a function call, making it call another unintended function in the program's binary. The C code is shown in Figure 15(a). In main, there is



```
int func1(char *payload)      void vuln(char *payload){
{                               char buf[256];
   vuln(payload);               ...
   return 0;                    long new_addr=0x000084af;
}                               memcpy(payload+length,
                                   &new_addr,sizeof(long));
int func2(char *payload)        memcpy(buf,payload,
{                                  length+sizeof(long));
   vuln(payload);             }
   return 0;
}
                           (a)
     00008488 <func1>:
        ...
        8492:    f000 f817   bl     84c4 <vuln>
        8496:    2300        movs r3, #0
        ...
     000084a0 <func2>:
        ...
        84aa:    f000 f80b   bl     84c4 <vuln>
        84ae:    f248 50e4   movwr0, #34276
        ...
                           (b)
```

Fig. 14. Code illustrating the stateful SP1 attack.



```
typedef struct {
   char buffer[32];
   void (*func)();          int main(int argc, char* argv[]){
} vuln_struct;                 ...
                               vuln_struct struct_attack;
void func_wrong(){             struct_attack.func =
   ...                            func_correct;
}                              memcpy(struct_attack.buffer,
                                   data, 64);
void func_correct(){           struct_attack.func();
   ...                         return 0;
}                           }
...                            (a)

...
8490:  f248 4371   movw    r3, #33905  ; 0x8471
8494:  f2c0 0300   movt    r3, #0
8498:  613b        str     r3, [r7, #16]
...
84be:  693b        ldr     r3, [r7, #16]
84c0:  4798        blx     r3
...
                               (b)
```

Fig. 15. Code illustrating the fine-grained SP2 attack.

a structure `struct_attack`, which contains a buffer and a function pointer. Usually, the function pointer in the memory is right after the buffer. The user data, which can be controlled by the attacker, is copied to the buffer through *memcpy*. An attacker can input the data with a larger size than the buffer, and put the address of the function `func_wrong` right after the 32-byte's data `buffer`. In this way, when the `struct_attack.func()` is called, function `func_wrong` is executed rather than the correct function `func_correct`.

For our fine-grained CFI, FastCFI can easily identify this attack. Figure 15(b) shows part of the assembly in `main`. The instruction at `84c0` is the function call `struct_attack.func()`. The program would jump to the address stored in `r3`. By backtracking the value in r3, we can find that it should be `0x8471`, where there is the entry of `func_correct`. This is a typical example of indirect branch with constant target address that we discussed before. We create the CFG with a node containing the instruction at `84c0`, and the only outgoing edge of this node is to the node containing the entry of `func_correct`. If the buffer overflow is performed by an attacker, then the control-flow does not go through the correct edge in CFG. This can be detected by FastCFI, both with and without CFG compression.

However, this attack cannot be identified by coarse-grained CFI techniques such as Lee et al. [27]. In Reference [27], it only checks if the indirect branch instruction performed as a function call is at the function's entry. For the example above, the attacked target address of the indirect branch instruction at `84c0` is still the function entry. This would be ignored in Reference [27].

6.4 Performance Overhead

We used the SPEC CPU2006 benchmarks [38] to evaluate the runtime overhead of FastCFI. We successfully ran all the benchmarks.[1]

The results are reported in Figure 16, including a comparison with the results from two recent works: Griffin [16] and Lee [27]. The work Griffin [16] uses Intel PT to generate the traces, but since it uses software to check CFI, the performance overhead is not negligible. The work Lee [27]

[1]Except *403.gcc*, which could not be cross compiled by the *arm-linux-gnueabihf-gcc(g++)* compiler.

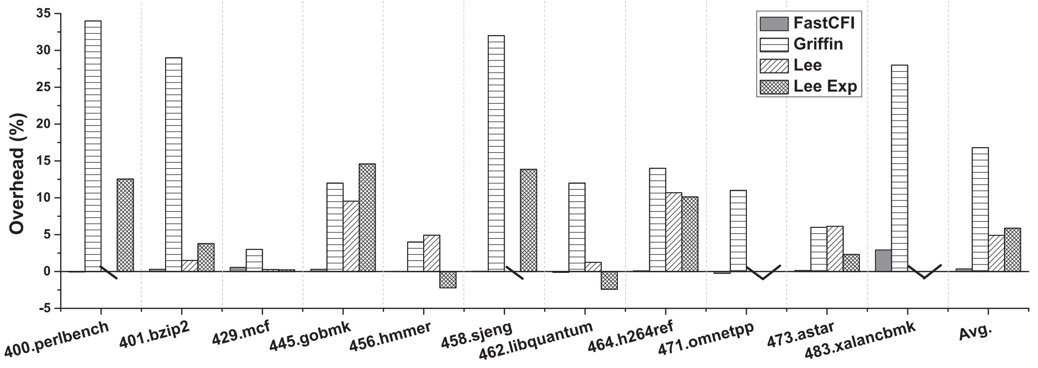


Fig. 16. The runtime overhead on SPEC 2006 benchmarks.

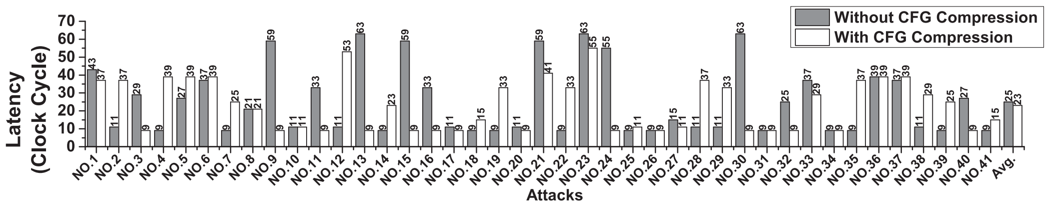


Fig. 17. The latency for FPGA to identify attacks.

also uses PTM and TPIU, but it instruments the program that also leads to performance overhead. Both results of Lee and Griffin are copied from the original papers [16, 27]. For Lee [27], some benchmarks are marked with "\", because they were not evaluated in Lee's work. Besides, we also did the code instrumentation and repeated the overhead experiments in Reference [27], the results are shown as *Lee Exp*. The benchmarks not evaluated in *Lee Exp* (marked with "/") are also not evaluated by *Lee*'s original work [27]. Moreover, *400.perlbench* and *458.sjeng* are not evaluated by *Lee* but evaluated by our repeated experiment *Lee Exp*. There are some benchmarks, such as *471.omnetpp*, which have overhead less than 0. This is likely due to cache effects, random numbers used in the programs, or the noise of the measurement, since the actual overhead is negligible for these benchmarks. The maximum frequency of the ARM core is not changed when using FastCFI, since FastCFI never modifies the hardware architecture of the ARM core, and only uses the provided hardware modules, which are PTM and CoreSight. The maximum frequency of the design on FPGA is about 35 MHz, which is fast enough to match the speed of the generation of the trace packets.

Overall, FastCFI has the lowest performance overhead, only 0.36% on average. The low overhead can be attributed to the fact that we do not add or modify anything on the software side, and there is no code instrumentation or running of other programs. The only overhead is caused by enabling the PTM device.

6.5 Latency

We also evaluated the latency introduced by FPGA to detect CFI violations, since it relies on TPIU to communicate the trace between the ARM core and FPGA. The latency is the clock cycles needed by FPGA to identify the attacks after receiving the trace packet containing the CFI violation information. The results are shown in Figure 17. Overall, FastCFI has a latency within dozens of clock cycles only. We note that some other hardware-based techniques such as Reference [8] incur a

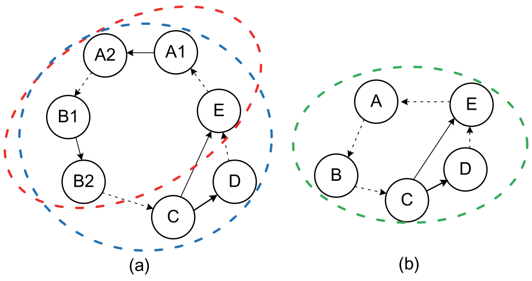


Fig. 18. An CFG compression example.

latency of tens of thousands of clock cycles, due to a more complex architectural design. Meanwhile, the CFG compression does not have an obvious effect on the latency.

The latency varies between different attacks. This depends on the quantity of data in the FIFO when the wrong control-flow information comes. The data in the FIFO must be processed sequentially by the CFI verification module, and more data can incur longer latency. In general, this can be affected by many factors, such as the target program itself and the input. Besides, the target program is run on the operating system, and there are some other background programs that can cause context switches. Running the target program at different moments can cause the context switch to happen at different points of the target program. Therefore, the interval of two trace packets corresponding to two branch instructions received by FPGA is not necessarily the same each time the target program is running. This is why sometimes the latency is even higher when there is CFG compression than there is no CFG compression. However, on average, with CFG compression, the latency is lower.

6.6 Circuit Resource Use and Compilation Time

Resource use is important for hardware design. Due to the resource limitation of our FPGA, for some benchmarks the system cannot fully verify the whole CFG, but only a sub-CFG, and ignores the instruction flow transitions that happen outside the sub-CFG. In our experiments, we always create the complete CFG first, apply CFG compression next, and then select as many CFG nodes or coalesced CFG nodes as our FPGA can contain for the sub-CFG. In practice, the sub-CFG can be specified by the user or developer, who may choose the most security sensitive parts of the code to protect against CFI attacks. Although the code coverage is not the full program for some benchmarks, the most important contribution is that FastCFI can check the sub-CFG precisely and quickly, with negligible performance overhead. This is a tradeoff between the security and the coverage. Besides, the code coverage can be increased by using a larger FPGA. The basic element of Altera FPGA is called **Adaptive Logic Module (ALM)**, which is similar to lookup table. Our FPGA has around 30K ALMs, while the state-of-the-art FPGA can contain more than 900K ALMs, which is enough for a large program in our benchmarks.

The resource use results are reported in Table 3. The results are separated into two parts in terms of the columns, one without CFG compression and another with CFG compression. The results are also separated into two parts in terms of the rows, one with partial CFG implemented and another with full CFG implemented. An example in Figure 18 is used to illustrate the metrics used for evaluating the effectiveness of CFG compression. The example CFG and its corresponding compressed CFG are shown in Figure 18(a) and (b), respectively. In Table 3, for the results without CFG compression, "CFG Nodes" represents the number of CFG nodes in the non-compressed CFG included in the CFG checker, such as the nodes in the set of the red circle in Figure 18(a), which is $\{A1, A2, B1, B2, E\}$. The set of these CFG nodes is named S. For the results with CFG

Table 3. Resource Use on SPEC 2006 Benchmarks

	Benchmark	Without CFG Compression			With CFG Compression					Total CFG Nodes	False Alarm?
		CFG Nodes	# of ALMs	Compile Time	Compressed CFG Nodes	CFG Nodes	CFG Coverage Increase	# of ALMs	Compile Time		
Partial CFG	400.perlbench	4563	32070	18m55s	4611	5816	27.46%	31559	21m00s	65083	None
	445.gobmk	4585	31604	19m11s	4547	6133	33.76%	31308	20m36s	37019	None
	456.hmmer	4602	31449	19m10s	4597	5269	14.49%	30590	20m23s	12286	None
	458.sjeng	4591	18738	15m08s	4719	5405	17.73%	31048	20m57s	6458	None
	464.h264ref	4513	32070	19m15s	4448	5641	24.99%	30757	19m02s	15195	None
	471.omnetpp	4763	30250	18m07s	4719	4837	1.55%	28507	20m26s	31811	None
	483.xalancbmk	4807	31576	18m39s	4605	5243	9.07%	29608	20m37s	173204	None
	Benchmark	Without CFG Compression			With CFG Compression					Total CFG Nodes	False Alarm?
		CFG Nodes	# of ALMs	Compile Time	Compressed CFG Nodes	CFG Nodes	ALM Use Decrease	# of ALMs	Compile Time		
Full CFG	401.bzip2	2247	22840	15m32s	1796	2247	19.51%	20518	15m57s	2247	None
	429.mcf	471	16171	13m48s	262	471	13.20%	15480	14m32s	471	None
	462.libquantum	1300	18738	15m08s	1223	1300	4.76%	18367	16m33s	1300	None
	473.astar	1345	18995	15m07s	1116	1345	10.21%	18169	16m11s	1345	None

compression, "Compressed CFG Nodes" represents the number of CFG nodes in the compressed CFG included in the CFG checker, such as the nodes in the set of the green circle in Figure 18(b), which is $\{A, B, C, D, E\}$. The set of these CFG nodes is named S'. For the results with CFG compression, "CFG Nodes" represents the number of the CFG nodes in the non-compressed CFG, which the CFG nodes in the compressed CFG included in the CFG checker correspond to, such as the nodes in the set of the blue circle in Figure 18(a), which is $\{A1, A2, B1, B2, C, D, E\}$. The set of these CFG nodes is named S'', where the nodes are the nodes in S' before CFG compression. "CFG Coverage Increase" indicates percentage increase of the number of nodes in S'' comparing to the number of the nodes in S. For the example in Figure 18, the CFG coverage increase is $\frac{7-5}{5} = 40\%$. The decoder and CFI verification module (without the CFG checker) uses 2755 and 4015 ALMs, and all the parts (such as the modules communicating with ARM processor and FPGA) excluding the CFG checker use 10938 ALMs. "ALM use decrease" indicates the percentage of the number of ALMs needed decreased by applying CFG compression, when the CFG can be fully implemented. For these experiments, we group 100 blocks in one small Verilog module as discussed in Section 4.5.1. Overall, without CFG compression, our current FPGA can support around 4,600 CFG nodes. When the CFG compression is applied, for the benchmarks where partial CFG is implemented, the FPGA can still support around 4600 CFG nodes in the compressed CFG, which corresponds 18.44% more non-compressed CFG nodes (the nodes in S'') compared to the implementation without CFG compression (the nodes in S). For the small benchmarks where full CFG is implemented, CFG compression can decrease the number of ALM needed, and the number of ALMs needed by the CFG checker is decreased by 11.93% on average. Both partial and full CFG implementation benefits are from the fact that the compressed CFG has less hardware expense when the compressed CFG is implemented in the CFG checker, because some CFG nodes coalesced into fewer nodes. Note that even with only partial CFGs, FastCFI does not report any false alarms on the studied benchmarks. As also reported in Table 3, the Verilog compilation time, including FPGA layout synthesis, in our experiments is less than around 20 minutes for each benchmark.

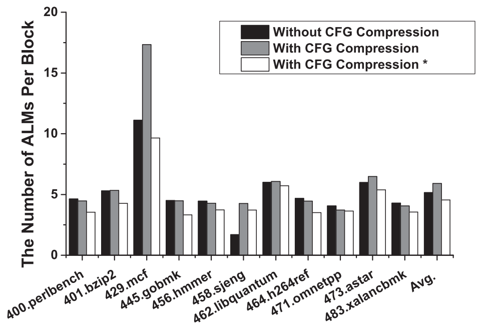


Fig. 19. The number of ALMs per block of different benchmarks.

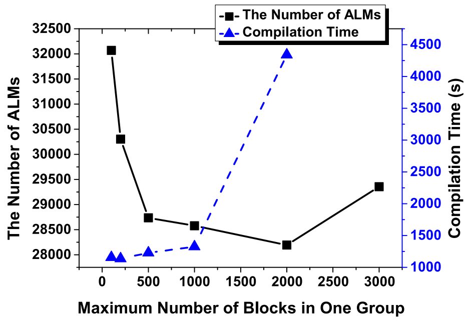


Fig. 20. The relationship between the number of ALMs, compilation time and maximum number of blocks in one block group of 483.xalancbmk.

The number of ALMs used by each block in CFG checker is shown in Figure 19. It does not vary much between different benchmarks, except *429.mcf* and *458.sjeng*. The result is the ratio between the number of ALMs needed for the CFG checker and the number of CFG checker blocks. The number of CFG checker blocks is defined in the following: The CFG checker blocks in the black bar correspond to the CFG nodes in S. The CFG checker blocks in the gray bar correspond to the CFG nodes in S'. The CFG checker blocks in the white bar, of which the legend is marked by "*", correspond to the CFG nodes in S''. The average number of ALMs per block is about 5. The block corresponding to the CFG node in the compressed CFG (grey bars) spends more ALMs than the block corresponding to CFG node without compression (black bars), because the address range of the coalesced node is the union of the address ranges of all the nodes before CFG compression, and this may require more comparisons in *if* statement of the block if the address ranges of the nodes before CFG compression are not continuous. However, by CFG compression, lots of CFG nodes are compressed to coalesced nodes, and the total number of CFG nodes in the compressed CFG is decreased. This overwhelms the disadvantage that the average number of ALMs needed by each block of the compressed CFG is increased, which is proved by the results of the white bars. The results of the white bars indicate that each block corresponding to the node in S'' spends fewer ALMs than the block corresponding to the node in S. Since node in S and node in S'' are both the nodes in the non-compressed CFG, the results indicate that it costs less hardware expense (fewer ALMs) to implement each node in the non-compressed CFG by using CFG compression than not using CFG compression.

We also evaluated the performance improvement (as described in Section 5.1) by using small Verilog modules of block groups instead of putting Verilog codes of all the blocks in the top CFG checker Verilog module. We use a typical benchmark *483.xalancbmk* to find out the relationship between the number of blocks in one group and the resource use. The experiments are done without CFG compression, and results are shown in Figure 20.

Overall, using more small Verilog modules reduces the time for compilation. When more blocks are grouped into one small Verilog module, it increases the optimization workload of each small module for the compilation tool, and the time complexity of Verilog compilation is also not linear. This results in the large time cost when the number of blocks is about 2,000 in one group. For the number of ALMs, the results show an odder behavior. In theory, if we put more blocks in one group, then the number of ALMs should decrease because of the better optimization. However, in Figure 20 after the number of blocks in one group is more than 2,000, the number of ALMs increases a lot. The reason may be that when there are too many blocks in one small Verilog module, it is hard for the compilation tool to find the optimized solution, even though there may exist a solution that is more optimized. When the number of blocks is not large, the compilation tool is competent to find almost the best solution, which makes the curve in Figure 20 as we expect.

7 CONCLUSION

We have presented an FPGA-based CFI system named FastCFI. To the best of our knowledge, it is the first to simultaneously achieve low overhead, fine-grained and stateful verification, and independence of code instrumentation. It does not produce false alarms and has low detection latency. It successfully detects all CFI violations in major benchmarks and incurs an average overhead of 0.36%. While it offers the computing efficiency of FPGAs, its deployment is nearly as convenient as software due to our automated Verilog generation technique. These advantages make FastCFI be feasible to be applied to the systems having high real-time and security requirements.

ACKNOWLEDGMENTS

We thank all the reviewers for their careful and detailed comments that greatly help us improve our article.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, 340–353.
[2] Address Space Layout Randomization. 2015. Retrieved from http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/15/09_lecture.pdf.
[3] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. 2006. Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Trans. VLSI Syst.* 14, 12 (2006), 1295–1308.
[4] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, 30–40.
[5] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. 2014. Ropecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the Symposium on Network and Distributed System Security* (2014).
[6] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the ACM Conference on Data and Application Security and Privacy* (2016), 38–49.
[7] CoreSight Program Flow Trace. 2011. Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf.
[8] Sanjeev Das, Yang Liu, Wei Zhang, and Chandramohan Mahinthan. 2016. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE Trans. Inf. Forens. Secur.* 11, 2 (2016), 289–302.
[9] Sanjeev Das, Wei Zhang, and Yang Liu. 2016. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. *IEEE Trans. VLSI Syst.* 24, 11 (2016), 3193–3207.
[10] Lucas Davi, Ra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad reza Sadeghi. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the Symposium on Network and Distributed System Security.*
[11] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the Annual Design Automation Conference*, 74:1–74:6.
[12] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the USENIX Conference on Security*, 401–416.
[13] Ruan de Clercq, Johannes Gtzfried, David bler, Pieter Maene, and Ingrid Verbauwhede. 2017. SOFIA: Software and control flow integrity architecture. *Comput. Secur.* 68 (2017), 16–35.
[14] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *Proceedings of the USENIX Conference on Security*, 131–148.
[15] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. 2009. Defending embedded systems against control flow attacks. In *Proceedings of the ACM Workshop on Secure Execution of Untrusted Code*, 19–26.

[16] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding control flows using Intel processor trace. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 585–598.

[17] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 173–184.

[18] Zonglin Guo, Ram Bhakta, and Ian G. Harris. 2014. Control-flow checking for intrusion detection via a real-time debug interface. In *Proceedings of the International Conference on Smart Computing Workshops*, 87–92.

[19] Jeff Huang and Arun K. Rajagopalan. 2016. Precise and maximal race detection from incomplete traces. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 462–476.

[20] IDA. 2011. Retrieved from https://www.hex-rays.com/products/ida/index.shtml.

[21] Intel CET. 2019. Retrieved from https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[22] Intel Quartus Prime. 2017. Retrieved from https://fpgasoftware.intel.com/17.1/?edition=lite.

[23] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the Annual International Symposium on Computer Architecture*, 94–105.

[24] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2014. Efficiently securing systems from code reuse attacks. *IEEE Trans. Comput.* 63, 5 (2014), 1144–1156.

[25] Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2013. SCRAP: Architecture for signature-based protection from code reuse attacks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 258–269.

[26] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. 2016. Integration of ROP/JOP monitoring IPs in an ARM-based SoC. In *Proceedings of the Conference on Design, Automation & Test in Europe*, 331–336.

[27] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. 2017. Using coresight PTM to integrate CRA monitoring IPs in an ARM-based SoC. In *Proceedings of the ACM Transactions on Design Automation of Electronic Systems*, 52:1–52:25.

[28] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.* 52, 6 (2019), 1–39.

[29] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and efficient CFI enforcement with Intel processor trace. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 529–540.

[30] Shufu Mao and Tilman Wolf. 2007. Hardware support for secure processing in embedded systems. In *Proceedings of the Annual Design Automation Conference*, 483–488.

[31] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. 2006. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.* 55, 10 (2006), 1271–1285.

[32] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the USENIX Conference on Security*, 447–462.

[33] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2015. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro* 35, 3 (2015), 10–22.

[34] Mehryar Rahmatian, Hessam Kooti, Ian G. Harris, and Elaheh Bozorgzadeh. 2012. Hardware-assisted detection of malicious software in embedded systems. *IEEE Embed. Syst. Lett.* 4, 4 (2012), 94–97.

[35] RIPE. 2011. Retrieved from https://github.com/johnwilander/RIPE.

[36] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security*, 552–561.

[37] Shellcodes Database. 2008. Retrieved from http://shell-storm.org/shellcode/.

[38] SPEC CPU 2006 Benchmark. Retrieved from https://www.spec.org/cpu2006/.

[39] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. 2016. Strategy without Tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *Proceedings of the Annual Design Automation Conference*, 1–6.

[40] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the Annual Computer Security Applications Conference*, 41–50.
[41] Write XOR Execute. Retrieved from https://en.wikipedia.org/wiki/W%5EX.
[42] Yubin Xia, Yutao Liu, H. Chen, and B. Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 1–12.
[43] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the USENIX Conference on Security*, 337–352.

Received February 2020; revised February 2021; accepted March 2021