FastCFI: Real-Time Control Flow Integrity Using FPGA Without Code Instrumentation

Lang Feng[1(✉)], Jeff Huang[2], Jiang Hu[1,2], and Abhijith Reddy[1]

[1] Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843, USA
{flwave,jianghu,abreddy}@tamu.edu
[2] Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843, USA
jeffhuang@tamu.edu

Abstract. 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, which forms a non-trivial hurdle to the application of CFI. 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. Experiments on popular benchmarks confirm that FastCFI can detect fine-grained CFI violations over unmodified binaries. The measurement results show an average of 0.36% performance overhead on SPEC 2006 benchmarks.

1 Introduction

Control Flow Integrity (CFI) [1] is to regulate instruction flow transitions, such as `branch`, toward target addresses conforming to the original design intention. Such 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) [34] and jump oriented programming (JOP) [3].

Software-based CFI usually competes for the same processor resource as the software application being protected [1,9,30,41], 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 much lower overhead. Indeed, hardware-based CFI has attracted significant research attention recently [13,15,26,40].

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

This work is partially supported by NSF (CNS-1618824).

B. Finkbeiner and L. Mariani (Eds.): RV 2019, LNCS 11757, pp. 221–238, 2019.
https://doi.org/10.1007/978-3-030-32079-9_13

1. **Code instrumentation.** Previous hardware CFI methods often rely on code instrumentation [5,10,22,23,25,26,37]. 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 [18].
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.
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 [17,32,40] 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 term 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 paper, we present FastCFI, which is an FPGA-based CFI system. FPGA implementation is a customized hardware solution that is much more power-efficient than software-based solution on general purpose microprocessors. 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 [31]. 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.

Table 1. Comparison among different methods.

Method	Fine-grained	Stateful	No instrumentation	<1% overhead	No false alarm
FastCFI	√	√	√	√	√
Lee [26]	×	√	×	×	√
CONVERSE [17]	√	×	√	√	×
Griffin [15]	√	√	√	×	√
FlowGuard [27]	×	√	√	×	√
CFIMon [40]	×	×	√	×	×
MoCFI [9]	√	√	×	×	×
Zhang [41]	×	√	×	×	√
kBouncer [30]	×	√	√	√	√
Ding [13]	√	√	√	×	√
Abadi [1]	√	√	×	×	√

FastCFI is the first fine-grained and stateful CFI system with negligible overhead and without using code instrumentation. Moreover, FastCFI does not produce any false alarm and has low detection latency.

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

The source code of FastCFI is available at [35]. The main contributions of our paper are:

- 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.
- A concrete system implementation based on FPGA, instead of simulation.
- A new circuit design technique that can automatically generate Verilog HDL for the application dependent component and therefore facilitates fast deployment of FastCFI systems.
- 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 those 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 which are relatively expensive as of today. FastCFI can also be applied in software supply chain 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 (Sect. 5.4) and low overhead (Sect. 5.3), indicating its high real-time capability. On 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 (Sect. 5.2) 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 paper is organized as follows. Section 2 introduces the background on CFI and control flow graph; Sect. 3 discusses previous work; Sect. 4 presents our proposed system design; Sect. 5 reports the experimental results and Sect. 6 concludes the paper.

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 Fig. 1(a), the instructions are divided into seven segments, each of which corresponds to a node in Fig. 1(b). The solid and

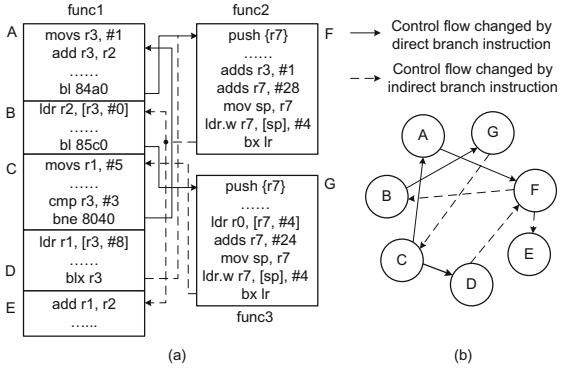

Fig. 1. Example of control flow graph.

dashed edges in Fig. 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.

Once a CFG is constructed for a software, CFI of this software execution is enforced by verifying if an execution trace conforms to the CFG. 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 F is invoked by function call from A, the last instruction in F should only return to the instruction right after A, which is in B. Therefore, function return $F \rightarrow B$ is legal while transition $F \rightarrow E$ is illegal.

3 Previous Work

3.1 Software-Based CFI

Early work on CFI was mostly realized by software implementation. The seminal work by Abadi et al. [1] proposes two code instrumentation approaches, which have average overhead of 16% and 21%, respectively, on the SPEC 2000 benchmark. Later work targeted CFI at specific application scenarios. For example, the method of Davi et al. [9] is designed for smartphones, and the work by Zhang and Sekar [41] addresses how to handle COTS binary codes. Several works [4,11,30] attempt to reduce performance overhead or avoid code instrumentation by sacrificing granularity or security coverage. For example, kBouncer [30] has very low overhead and code instrumentation is avoided in [4]. However, both methods handle ROP attacks only.

FastCFI is hardware-based CFI, which avoids some disadvantages in software-base CFI, such as high performance overhead [1,9], coarse-grained CFI policy [4,11,30], and requiring code instrumentation [1,9,41].

3.2 Hardware-Based CFI

Recently, several hardware-based CFI approaches [2,5,8,10,12–17,22–29,32,37,40] have been proposed, based on Intel Processor Trace [13,15,16,27], performance counters [40], FPGA [8,25,26], and others [17]. Intel also proposed the control-flow enforcement technology (CET) [20]. 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, great amount of the hardware-based CFI approaches require hardware modification [2,5,8,10,12,14,22–24,28,29,32,37]. Modifying hardware structure such as adding additional modules inside the processor's pipeline is not practical, since one will need 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 [5,10,22,23,25,26,37]. However, this results in large overhead, and code instrumentation itself is also not secure and sometimes even impossible. Secondly, FastCFI has a low overhead compared to some previous works [12,13,15,25,26]. High overhead is unacceptable in some real-time applications. Also, not all the hardware-based CFI are fine-grained and stateful [8,17,25,26,40]. They may miss some attacks. In operating system, false positive may delay all the processes, but some techniques used in previous works lead to this [17,40]. 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, etc. A few previous works prefer using simulator for implementation [8,10,14,22,23], but this will 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 Fig. 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. Program Trace Macrocell (PTM) generates compressed control-flow traces according to instructions processed by 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. The key ideas of FastCFI can be applied to other platforms such as x86 architecture.

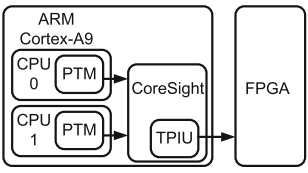

Fig. 2. System platform for the proposed CFI.

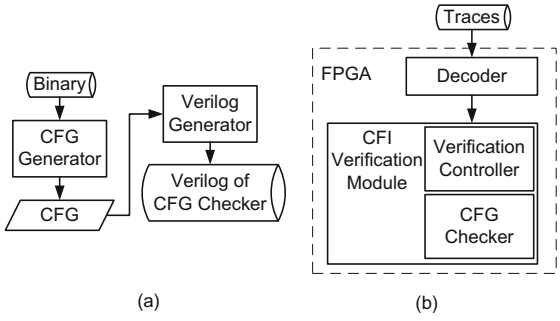

Fig. 3. System design overview: (a) offline CFG checker generator; (b) online CFI verifier.

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 Fig. 3. The CFG checker generator is a software that takes application software binary as input and generates CFG checker design in Verilog. During online software execution, a trace captured through ARM CoreSight is first decoded in order to understand its semantics. The decoded trace data is 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 FPGA.

4.3 Offline CFG Checker Generator

To give the hardware verification circuits the correct execution information which can be represented by 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 software binary, the generator first converts it to assembly code. It extracts CFG from the assembly code and generates the Verilog design of 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 target vulnerable binary is required.

We denote a sequence of assembly instructions as $I_1, I_2, ...I_{m1}, B_1, I_{m1+1}, I_{m1+2}, ..., I_{m2}, B_2, ...B_n...$, where $B_1, B_2, ...B_n$ 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 $\{I_1, I_2, ...I_{m1}, B_1\}$, $\{I_{m1+1}, I_{m1+2}, ..., I_{m2}, B_2\}$,..., 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.

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 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 branch is a tricky case, as its target address is stored in a register. 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 Sect. 4.5.

We developed a software program to automatically construct CFG from binary code. The generator further creates Verilog description for the CFG checker circuit. Meanwhile, our framework is general and can accommodate other tools such as IDA [19].

4.4 Trace Decoder

The decoder takes 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* [6], 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, its target address is contained in *Branch*. Some other types of packets, such as *I-sync* [6], 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*, etc.
- 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 current address periodically.
- 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 is to examine if flow transitions in a software execution trace are consistent with transitions specified in CFG, which is embedded in the CFG checker. In order 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 [26], it is acquired through code instrumentation. Without code instrumentation, identifying the source/current node

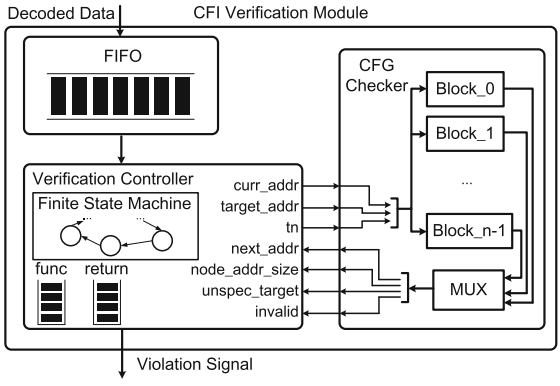

Fig. 4. Architecture of CFI verification module.

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 Fig. 1. Suppose we know the address of the first instruction of 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. When Write⊕Execute [39] feature is applied in an operating system, an attacker is not able to change the code and the target of each direct branch. Therefore, by observing T from trace decoder 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. 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 last instruction of node F is function return, which is an indirect branch. Its execution leads to a *Branch* in decoded trace packet. 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 architecture of the CFI verification module is shown in Fig. 4. Its key components, CFG checker and verification controller, are described as follows.

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 Fig. 4, each of which corresponds to a node in CFG. Assigning each CFG node in one block makes the CFG node search run in parallel, and this will greatly increase 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:

- *next_addr*: the next program counter address after executing the branch of 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, 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 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 direct branch generates *next_addr* according to input *tn*. If *tn* is T, indicating that the branch is taken, 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.** If *tn* is T, the *target_addr* is compared with the possible *next_addr* from CFG. If they are the same, 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 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. When the CFG checker is implemented in Verilog HDL, we use *if* statement for each block, where the condition is that the current address is within the range of instructions' addresses of the corresponding CFG node. Inside *if*, the *tn* and *target_addr* are examined by the three rules above. Since the only difference among the same type blocks is the parameter but not the structure, we can write three Verilog description templates for all the three types and use software to automatically instantiate one of them for each block, which is the way that the CFG checker generator in Sect. 4.3 works.

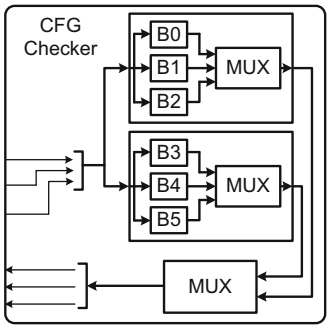

Fig. 5. An example of grouping blocks in CFG checker.

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 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 CFG and the verification module is to perform other checks for CFI which will be discussed later in this section.

For the Verilog compilation tool, optimizing a large number of blocks is more difficult than optimizing fewer blocks. Therefore, in our implementation, we develop a hierarchical approach that groups blocks into small Verilog modules. Each small module takes the checker input to all of its internal blocks, and selects an output among all of its internal blocks. For example, in Fig. 5, the CFG checker has 6 blocks, $B0$ to $B5$, which are grouped into two small modules. In this way, the compiling optimization is directed to perform in a hierarchical manner to reach different resource use and compiling time tradeoffs.

Verification Controller. The verification controller takes the decoded trace packets as input, feeds input to the CFG checker, and analyzes the checker results to locate 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 Fig. 6. It also has a function stack, which 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.

The controller operations start from the WAIT state, which attempts to capture 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 *I-sync*.

Once an executing instruction address is acquired, the controller enters the SCOPE_CHECK state, where the instruction address is sent to the CFG checker as *curr_addr* to tell

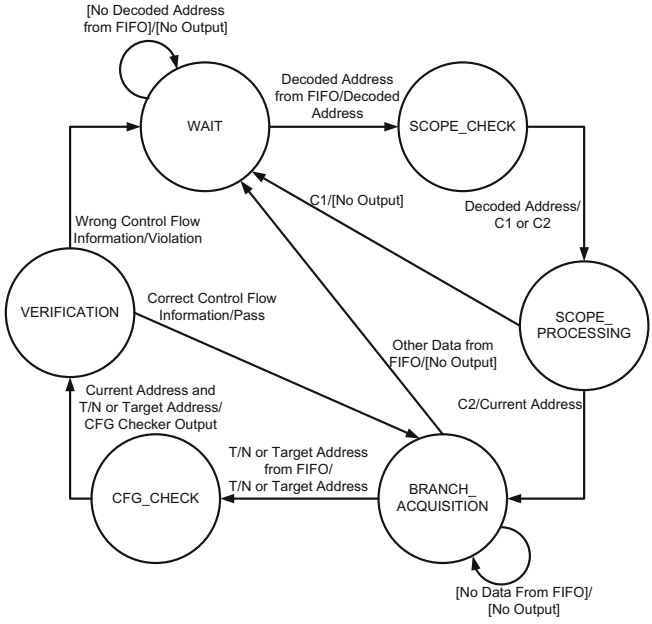

Fig. 6. State transition diagram of the controller.

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, the instruction address is not in the CFG and the next state is WAIT. If the result is C2, the controller records the context ID, which identifies the software execution to be verified, and then moves to state BRANCH_ACQUISITION.

At 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, the controller switches to the WAIT so as to update the reference instruction address. If branch information, *Atom* or *Branch*, is received, 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 VERIFICATION. This state is to analyze the checking results and keep track of instruction execution location. If condition C3 occurs, the function address range is pushed in to the function stack. The function stack top always stores the address range of the current function. C3 also implies that the previous node made a function call, and notifies the controller to push the return address onto the return stack.

If the target address of a branch instruction is specified in the CFG, either C4 or C5 will hold when the corresponding block is activated. Condition C4 indicates that CFI verification is passed without seeing any violations. 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.

Otherwise, if the target address of an indirect branch is not specified, condition C6 will hold, 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. The first case is 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 same, the current indirect branch is confirmed to be a function return, which is legal. Note that this check is stateful as it relies on historical information stored in the return stack. The second one is 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, no violation signal is triggered. The third one is Branch as a new function call. If the actual target address is not in current function, the only legal scenario is that a new function call is made. To verify if a new function call is indeed made, the controller updates the current address with the next address and waits for the next BRANCH_ACQUISITION and CFG_CHECK result. If the next result indicates C3 and the current address is the same as the new function entry address, 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.

The verification is not only stateful, but also fine-grained as its resolution is on each individual edge in the CFG. We also re-emphasize that our work is general and flexible enough to be applied with other code static analysis tools.

5 Experiments and Results

5.1 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. In addition, we use Quartus Prime 17.1 [21] for Verilog compilation and FPGA layout synthesis, and Signal Tap Logic Analyzer for FPGA signal monitoring. The Verilog compilation is done on a desktop with an Intel 3.8 GHz CPU and 16 GB RAM.

5.2 Security

We use RIPE [33,38] to evaluate the effectiveness of FastCFI. RIPE is a popular benchmark that has been used frequently in previous works [13,15] for evaluating control flow defenses. 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.

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	-	-	-	-	√

Due to the engineering difficulties, it is hard to port all RIPE functions to ARM. In total, we recovered 41 attacks (which can run successfully on ARM),

including both return oriented programming (ROP) and jump oriented programming (JOP) attacks, as shown in Table 2 (Row #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 in Table 2 show that all these attacks can be identified by FastCFI. In addition, FastCFI does not report any false positive (for the newly introduced function with no attack).

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 Fig. 7(a), in SP1, there is a function *vuln* which 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. However, stateless CFI such as [17,40] and the coarse-grained approach in [15], would not be able to identify this attack.

```
int func1(char *payload)
{
   vuln(payload);
   return 0;
}

int func2(char *payload)
{
   vuln(payload);
   return 0;
}
```

```
void vuln(char *payload){
   char buf[256];
   ...
   long new_addr=0x000084af;
   memcpy(payload+length,
      &new_addr,sizeof(long));
   memcpy(buf,payload,
      length+sizeof(long));
}
```

(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. 7. Code illustrating the stateful SP1 attack.

```
typedef struct {
   char buffer[32];
   void (*func)();
} vuln_struct;

void func_wrong(){
   ...
}

void func_correct(){
   ...
}
...
```

```
int main(int argc, char* argv[]){
   ...
   vuln_struct struct_attack;
   struct_attack.func =
      func_correct;
   memcpy(struct_attack.buffer,
      data, 64);
   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. 8. Code illustrating the fine-grained SP2 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 Fig. 8(a). In *main*, there is 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. In this way, when the *struct_attack.func()* is called, function *func_wrong* will be executed rather than the correct function *func_correct*.

For our fine-grained CFI, FastCFI can easily identify this attack. Figure 8(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 will not go through the correct edge in CFG. This will be detected by FastCFI.

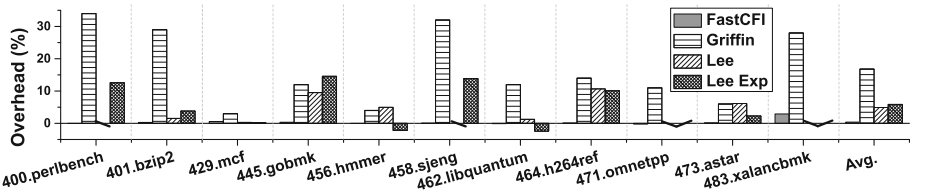

Fig. 9. The runtime overhead on SPEC 2006 benchmarks.

However, this attack cannot be identified by coarse-grained CFI techniques such as Lee et al. [26]. In [26], 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 by [26].

5.3 Performance Overhead

We used the SPEC CPU2006 benchmarks [36] to evaluate the runtime overhead of FastCFI. We successfully ran all the benchmarks, except *403.gcc*, which could not be cross compiled by the *arm-linux-gnueabihf-gcc(g++)* compiler.

The results are reported in Fig. 9, including a comparison with the results from two recent works: *Griffin* [15] and *Lee* [26]. Both results of *Lee* and *Griffin* are copied from the original papers [15,26]. For *Lee* [26], 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 [26], 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 [26]. 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 or the noise of the measurement, since the actual overhead is negligible.

Overall, FastCFI has the lowest performance overhead, only 0.36% on average. The reason is 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.

5.4 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 Fig. 10. Overall, FastCFI has a latency within dozens of clock cycles only. We note that some other hardware-based techniques such as [7] incur a latency of tens of thousands of clock cycles, due to a more complex architectural design.

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. The more data, the longer latency. In general, this can be affected by many factors, such as the target program itself, the input, or the other programs running on the same processor.

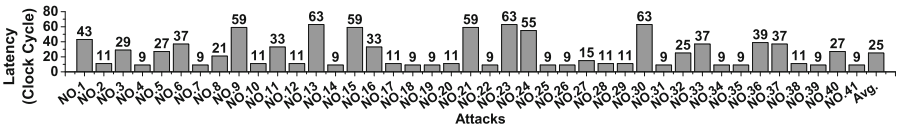

Fig. 10. The latency for FPGA to identify attacks.

5.5 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 may not fully verify the whole CFG, but a sub-CFG, and ignores the instruction flow transitions happened outside the sub-CFG. In our experiments, we always create the complete CFG first, and then select as many 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.

The resource use results are reported in Table 3. The ALM means adaptive logic module in Altera FPGA, which is the basic element of FPGA and similar to LUT (Lookup Tables). For these experiments, we group 100 blocks in one small Verilog module as discussed in Sect. 4.5. Overall, our current FPGA can support 4500−4600 CFG nodes. Note that even though with only the sub-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 20 min for each benchmark.

Table 3. Resource use on SPEC 2006 benchmarks.

Benchmark	Sub-CFG Nodes	Total CFG Nodes	# of ALMs	Compile Time	False Alarm?
400.perlbench	4563	65083	32070	18 m 55 s	None
401.bzip2	2247	2247	22840	15 m 32 s	None
429.mcf	471	471	16171	13 m 48 s	None
445.gobmk	4585	37019	31604	19 m 11 s	None
456.hmmer	4602	12286	31449	19 m 10 s	None
458.sjeng	4591	6458	18738	15 m 08 s	None
462.libquantum	1300	1300	18738	15 m 08 s	None
464.h264ref	4513	15195	32070	19 m 15 s	None
471.omnetpp	4763	31811	30250	18 m 07 s	None
473.astar	1345	1345	18995	15 m 07 s	None
483.xalancbmk	4807	173204	31576	18 m 39 s	None

6 Conclusion

We have presented an FPGA-based CFI system named FastCFI. To the best of our knowledge, it is the first to simultaneously achieves 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.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow Integrity. In: ACM Conference on Computer and Communications Security, pp. 340–353 (2005)
2. Arora, D., Ravi, S., Raghunathan, A., Jha, N.K.: Hardware-assisted run-time monitoring for secure program execution on embedded processors. IEEE Trans. Very Large Scale Integr. Syst. **14**(12), 1295–1308 (2006)
3. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: ACM Symposium on Information, Computer and Communications Security, pp. 30–40 (2011)
4. Cheng, Y., Zhou, Z., Miao, Y., Ding, X., Deng, H.R.: ROPecker: a generic and practical approach for defending against ROP attacks. In: Symposium on Network and Distributed System Security (2014)
5. Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S.: HCFI: Hardware-enforced Control-Flow Integrity. In: ACM Conference on Data and Application Security and Privacy, pp. 38–49 (2016)

6. CoreSight™ Program Flow Trace™. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf
7. Das, S., Liu, Y., Zhang, W., Mahinthan, C.: Semantics-based online malware detection: towards efficient real-time protection against malware. IEEE Trans. Inf. Forensics Secur. **11**(2), 289–302 (2016)
8. Das, S., Zhang, W., Liu, Y.: A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. IEEE Trans. Very Large Scale Integr. Syst. **24**(11), 3193–3207 (2016)
9. Davi, L., et al.: MoCFI: a framework to mitigate control-flow attacks on smartphones. In: Symposium on Network and Distributed System Security (2012)
10. Davi, L., et al.: HAFIX: Hardware-assisted Flow Integrity Extension. In: Annual Design Automation Conference, pp. 74:1–74: 6 (2015)
11. Davi, L., Sadeghi, A.-R., Lehmann, D., Monrose, F.: Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: USENIX Conference on Security, pp. 401–416 (2014)
12. de Clercq, R., Gtzfried, J., Bler, D., Maene, P., Verbauwhede, I.: SOFIA: Software and Control Flow Integrity Architecture. Comput. Secur. **68**(C), 16–35 (2017)
13. Ding, R., Qian, C., Song, C., Harris, B., Kim, T., Lee, W.: Efficient protection of path-sensitive control security. In: USENIX Conference on Security, pp. 131–148 (2017)
14. Francillon, A., Perito, D., Castelluccia, C.: Defending embedded systems against control flow attacks. In: ACM Workshop on Secure Execution of Untrusted Code, pp. 19–26 (2009)
15. Ge, X., Cui, W., Jaeger, T.: GRIFFIN: guarding control flows using Intel Processor trace. In: International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 585–598 (2017)
16. Gu, Y., Zhao, Q., Zhang, Y., Lin, Z.: PT-CFI: transparent backward-edge control flow violation detection using Intel Processor Trace. In: ACM Conference on Data and Application Security and Privacy, pp. 173–184 (2017)
17. Guo, Z., Bhakta, R., Harris, I.G.: Control-flow checking for intrusion detection via a real-time debug interface. In: International Conference on Smart Computing Workshops, pp. 87–92 (2014)
18. Huang, J., Rajagopalan, A.K.: Precise and maximal race detection from incomplete traces. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 462–476 (2016)
19. IDA. https://www.hex-rays.com/products/ida/index.shtml
20. Intel CET. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf
21. Intel Quartus Prime. https://fpgasoftware.intel.com/17.1/?edition=lite
22. Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., Ponomarev, D.: Branch regulation: low-overhead protection from code reuse attacks. In: Annual International Symposium on Computer Architecture, pp. 94–105 (2012)
23. Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., Ponomarev, D.: Efficiently securing systems from code reuse attacks. IEEE Trans. Comput. **63**(5), 1144–1156 (2014)
24. Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., Abu-Ghazaleh, N.: SCRAP: architecture for signature-based protection from code reuse attacks. In: IEEE International Symposium on High Performance Computer Architecture, pp. 258–269 (2013)
25. Lee, Y., Lee, J., Heo, I., Hwang, D., Paek, Y.: Integration of ROP/JOP Monitoring IPs in an ARM-based SoC. In: Conference on Design, Automation & Test in Europe, pp. 331–336 (2016)

26. Lee, Y., Lee, J., Heo, I., Hwang, D., Paek, Y.: Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC. ACM Trans. Des. Autom. Electron. Syst. **22**(3), 52:1–52:25 (2017)
27. Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., Guan, H.: Transparent and efficient CFI enforcement with Intel processor trace. In: IEEE International Symposium on High Performance Computer Architecture, pp. 529–540 (2017)
28. Mao, S., Wolf, T.: Hardware support for secure processing in embedded systems. In: Annual Design Automation Conference, pp. 483–488 (2007)
29. Ozdoganoglu, H., Vijaykumar, T.N., Brodley, C.E., Kuperman, B.A., Jalote, A.: SmashGuard: a hardware solution to prevent security attacks on the function return address. IEEE Trans. Comput. **55**(10), 1271–1285 (2006)
30. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: USENIX Conference on Security, pp. 447–462 (2013)
31. Putnam, A., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro **35**(3), 10–22 (2015)
32. Rahmatian, M., Kooti, H., Harris, I.G., Bozorgzadeh, E.: Hardware-assisted detection of malicious software in embedded systems. IEEE Embedd. Syst. Lett. **4**(4), 94–97 (2012)
33. RIPE. https://github.com/johnwilander/RIPE
34. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: ACM Conference on Computer and Communications Security, pp. 552–561 (2007)
35. Source Code of FastCFI. https://github.com/flwave/FastCFI
36. SPEC CPU 2006 Benchmark. https://www.spec.org/cpu2006/
37. Sullivan, D., Arias, O., Davi, L., Larsen, P., Sadeghi, A.-R., Jin, Y.: Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity. In: Annual Design Automation Conference, pp. 1–6 (2016)
38. Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., Joosen, W.: RIPE: Runtime Intrusion Prevention Evaluator. In: Annual Computer Security Applications Conference, pp. 41–50 (2011)
39. Write XOR Execute. https://en.wikipedia.org/wiki/W%5EX
40. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: detecting violation of control flow integrity using performance counters. In: IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 1–12 (2012)
41. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: USENIX Conference on Security, pp. 337–352 (2013)