7 Hardware Control Flow Integrity

Yier Jin, Dean Sullivan, Orlando Arias, Ahmad-Reza Sadeghi, Lucas Davi

Control-Flow Integrity (CFI) is a promising and general defense against control-flow hijacking with formal underpinnings. A key insight from the extensive research on CFI is that its effectiveness depends on the precision and coverage of a program's Control-Flow Graph (CFG). Since precise CFG generation is highly challenging and often difficult, many CFI schemes rely on brittle heuristics and imprecise, coarse-grained CFGs. Furthermore, comprehensive, fine-grained CFI defenses implemented purely in software incur overheads that are unacceptably high.

In this chapter, we first specify a CFI model that captures many known CFI techniques, including stateless and stateful approaches as well as fine-grained and coarse-grained CFI policies. We then design and implement a novel hardware-enhanced CFI. Key to this approach is a set of dedicated CFI instructions that can losslessly enforce any CFG and diverse CFI policies within our model. Moreover, we fully support multi-tasking and shared libraries, prevent various forms of code-reuse attacks, and allow code protected with CFI to interoperate with unprotected legacy code. Our prototype implementation on the SPARC LEON3 is highly efficient with a performance overhead of 1.75% on average when applied to several SPECInt2006 benchmarks and 0.5% when applied to EEMBC's CoreMark benchmark.

7.1 Introduction

Control-flow integrity has been proposed as a general defense technique against control-flow hijacking attacks [Abadi et al. 2005a, Abadi et al. 2009]. In particular, it defends against modern code-reuse attacks, such as Return-Oriented Programming (ROP) [Roemer et al. 2012]. These attacks are prevalent, Turing-complete,

and are repeatedly leveraged to compromise commonly used applications such as web browsers [Marschalek 2014] and document viewers [jduck 2010]. CFI mitigates these attacks by ensuring that an application follows a legitimate control-flow path. The legitimate paths are manifested in the application's control-flow graph derived during an offline static analysis phase. Whenever an attacker attempts to subvert the execution to follow an illegal control-flow path, CFI detects this malicious control flow and immediately terminates the process. In addition, CFI is not vulnerable to memory disclosure and side channel attacks [Snow et al. 2013, Bittau et al. 2014, Seibert et al. 2014], and allows verifiable security [Abadi et al. 2005b].

A number of CFI schemes have been proposed that aim at introducing practical CFI enforcement incurring almost no overhead [Zhang and Sekar 2013, Zhang et al. 2013, Pappas et al. 2013, Cheng et al. 2014]. On the other hand, these schemes enforce coarse-grained CFI policies that an attacker can bypass [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Göktaş et al. 2014b, Schuster et al. 2014]. In parallel to the development of practical CFI schemes, a number of defenses have been proposed that focus on a CFI subclass, i.e., only protecting indirect virtual calls to C++ virtual methods [Gawlik and Holz 2014, Zhang et al. 2015, Prakash et al. 2015]. However, Schuster et al. [2015] recently demonstrated that modern programs offer a large number of valid virtual methods. Hence, an attacker can exploit the available virtual methods to launch a code-reuse attack. Further, Google recently released a CFI compiler extension for virtual calls [Tice et al. 2014] that resists the latest attacks on virtual method exploitation [Schuster et al. 2015], but can be circumvented by means of stack attacks [Liebchen et al. 2015]. Last, Carlini et al. [2015e] question the overall benefit of CFI, since even fine-grained CFI protection still offers a large code base of valid CFG nodes and edges that an attacker can exploit.

The continued success of code-reuse attacks has several reasons. First, many CFI defenses evaluate their effectiveness based on existing exploits, which naturally do not align to any given CFI policy. These exploits are typically more sophisticated and can be rewritten to align to the CFI-enforced CFG [Carlini et al. 2015e, Liebchen et al. 2015, Carlini and Wagner 2014, Davi et al. 2014, Göktas et al. 2014a]. Second, CFI defenses leverage unreliable metrics, such as gadget reduction, gadget length, or average indirect branch reduction (AIR), to measure CFI precision [Zhang and Sekar 2013, Kayaalp et al. 2012, Niu and Tan 2014a, Mohan et al. 2015, Arias et al. 2015, Tice et al. 2014]. These metrics have frequently over-estimated the provided security and have been shown to be bypassable [Carlini et al. 2015e, Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014].

In this chapter, we first address the mismatch between a sound CFI policy and various insecure implementations by revisiting CFI to provide a comprehensive model covering many CFI policies proposed today. We then develop a precise, stateful CFI policy that enables us to address the granularity of a given CFI scheme and make informed design decisions regarding protection and cost of coverage. We then introduce a general-purpose, hardware-enhanced CFI platform that scales to the coverage provided by any CFG, enables highly efficient enforcement of diverse CFI policies, and losslessly enforces any provided CFG.

We also evaluate runtime attacks and CFI vulnerabilities using hardware-enhanced CFI by first evaluating its effectiveness based on the most current code-reuse attacks [Carlini et al. 2015e, Schuster et al. 2015, Carlini and Wagner 2014, Göktas et al. 2014a, Davi et al. 2014, Checkoway et al. 2010, Tran et al. 2011, Liebchen et al. 2015]. These attacks are able to perform malicious actions while adhering to the restrictions imposed by a CFI-protected system. We additionally address attacks targeting both C and C++ applications as well as JIT-compiled programs. Our evaluation focuses on fundamental attacks that manipulate or otherwise violate our CFI policy, assuming we are provided with a precise CFG.

We take a hardware-based approach for several reasons. First, as we will show, CFI in hardware along with dedicated CFI instructions scales for any CFG and various CFI policies from very coarse-grained to highly precise control-flow checks. Second, a hardware-based approach allows us to instantiate a CFI processor module that highly improves the efficiency of CFI while offering strong, precise CFI protection. Third, hardware-based CFI enables precise stateful CFI policy enforcement. Fourth, a CFI processor module can be associated to an on-chip, dedicated memory that securely isolates CFI data (e.g., CFG information).

We conducted a performance evaluation of our approach using SPEC2006 benchmarks and CoreMark micro-benchmarks on the SPARC LEON3 processor [Gaisler Research 2017]. Our system is highly efficient, incurring almost no performance overhead; on average only 1.75% for SPEC and 0.5% for CoreMark. Our hardware-enhanced CFI area overhead is negligible and can be clocked up to 3 GHz using a 32/28 nm process.

In summary, our core contributions of this chapter are as follows.

CFI model. We revisit CFI to reason about the protection offered by various CFI implementations and policies that have been presented thus far, and present *precise, stateful CFI*.

Scalable, precise CFI enforcement. We present a design that scales to any CFG provided and losslessly enforces the provided CFG.

Comprehensive prevention. Our CFI hardware platform prevents many known code-reuse attacks: traditional ROP [Shacham 2007], ROP without returns [Checkoway et al. 2010, Davi et al. 2014, Carlini and Wagner 2014] dynamic ROP [Snow et al. 2013], JOP [Checkoway et al. 2010], and full-function reuse [Schuster et al. 2015, Tran et al. 2011].

CFI hardware platform. We present the design, implementation, and evaluation of a scalable and highly efficient hardware-enhanced CFI implementation for the open source SPARC LEON3 hardware platform. Our hardware platform features new CFI instructions that support precise enforcement at diverse CFG granularities.

We stress that the goal of this chapter is the introduction and design of a hardware CFI framework that can enforce CFI policies of different precision, including coarse- and fine-grained variants. Our work is explicitly not about sophisticated static analysis of source code or advanced binary analysis to extract fine-grained control-flow graphs. Generation of CFGs for real-world software remains an open research problem. However, issues in CFG generation are orthogonal to the challenges we address: making CFI enforcement efficient by adding dedicated instructions and supporting hardware.

7.2 Threat Model and Assumptions

Our threat model follows the traditional CFI threat model. We assume an adversary who has arbitrary read and write access to data memory, and read access to code memory. As a consequence, the CFI threat model tolerates memory disclosure attacks, i.e., it allows information leakage but still protects applications against memory corruption attacks. The attacker can be either a local or remote attacker. However, the attacker only has access to user applications, as kernel exploits can undermine any security mechanism implemented for user-space applications.

CFI aims at defending against runtime exploits that violate the integrity of the program's control flow to perform malicious actions. That said, we target benign applications that an attacker attempts to compromise, but do not protect against applications that are inherently malicious. This includes cases where the attacker modifies the binary either in disk or memory. Further, we focus on code-reuse attacks but not code injection attacks, whichtoday are prevented by means of Data Execution Prevention (DEP) [Andersen and Abella 2004].

It is important to note that CFI does not defend against the so-called non-control-data attacks [Chen et al. 2005]. These attacks do not modify any code

pointer but non-control data, such as an authentication variable. Recently proposed hybrid attacks, called Control-Flow Bending (CFB), include both exploitation of non-control data and control data [Carlini et al. 2015e]. Our threat model focuses on the control-data part of the attack.

Return-oriented programming is a generic attack instantiation of code-reuse attacks: it combines short instruction sequences (gadgets) from various functions to generate a new malicious program [Roemer et al. 2012]. Typically, these sequences end with a return instruction to transfer control to the subsequent sequence [Shacham 2007]. That said, these attacks exploit *backward edges* (returns) of a program's control-flow graph. However, an attacker can also leverage sequences that terminate with an indirect call or jump instruction [Checkoway et al. 2010], that is, code-reuse attacks that exploit *forward edges* in the CFG. Sometimes these attacks are referred to as Jump-Oriented Programming (JOP) [Checkoway et al. 2010]. Both attack variants, ROP and JOP, have been shown to be Turing-complete, meaning that the identified code sequences form a Turing-complete language. We aim at defending against these attacks based on control-flow integrity in hardware.

Another code-reuse attack variant is a function-reuse attack that only invokes a chain of library functions [Nergal 2001, Schuster et al. 2015, Tran et al. 2011]. Existing CFI schemes rarely provide protection against these attacks. In fact, preventing these attacks is highly challenging: Consider a program that legitimately invokes a critical function, e.g., `open()`, via an indirect call. As a consequence, the critical function is considered a legitimate control-flow target in CFI. Protection of these code-reuse attacks are within the scope of our threat model.

7.3 Requirements

The requirements that satisfy the goals of a lossless, scalable, and highly efficient hardware-enhanced CFI framework are given below.

Precision. We must losslessly enforce any CFG with which we are provided. In general, it may be impossible to resolve a precise CFG either because source code is unavailable or the analysis is imprecise. In any case we must strictly enforce what we are given.

Scalability. The effectiveness of any CFI approach depends on the CFG precision. Hence, we require that our CFI scheme scales to any level of CFG precision. Given a CFG, we should be capable of enforcing precise CFI. Our system should also be capable of enforcing coarse-grained CFI if no precise CFG is available.

Efficiency. One of the main limitations of software-based CFI approaches are their significant performance overhead. As a consequence, we require negligible performance overhead for our CFI scheme.

Stateful. We require stateful CFI since stateless CFI is vulnerable to stitching gadgets [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Göktaş et al. 2014b, Schuster et al. 2014] and control-flow bending attacks [Carlini et al. 2015e].

Compatibility. A CFI scheme needs to co-exist with legacy programs that are not instrumented with CFI.

Security. Based on a precise CFG, we require the CFI scheme to cover all of the existing code-reuse attacks including traditional return-oriented programming [Shacham 2007], jump-oriented programming [Checkoway et al. 2010], just-in-time code-reuse attacks [Snow et al. 2013], and whole function-reuse attacks [Schuster et al. 2015].

7.4 Modeling CFI

It has been more than a decade since Abadi et al. [2005a] introduced the idea of control-flow integrity. Since then, various CFI implementations have been proposed, each with different performance and security metrics. We have noticed that there is a division in the community regarding what encompasses a "fine-grained" or "coarse-grained" policy and the kind of security each provides. In the remainder of this section, we present an abstract description of CFI and then use it to state the requirements of a theoretical CFI policy that can provide as much protection as possible. We start by defining a control-flow graph. Subsequently, we extend this mechanism to include the notion of execution state in a process. Using these definitions, we develop a CFI policy that can yield the maximum protection possible under the framework of what is computable and decidable. Our definitions can be shown to be equivalent to those presented in Abadi et al. [2005b], while presenting extensions to incorporate missing elements when needed.

7.4.1 Control-Flow Graph

To introduce our definition of a control-flow graph, we need to define its components. Let $\mathbb{C}$ be the set of control-flow instructions and $\mathbb{I}$ be the set of non-control-flow instructions. Then we say that a node $\mathbb{N}_i$ consists of a sequential set of non-control-flow instructions. That is, $\mathbb{N}_i = \{I_1, I_2, I_3, \ldots, I_z\}$, $I_{1,\ldots,z} \in \mathbb{I}$. An *edge* E_j is given by an instruction $I_C \in \mathbb{C}$, which transfers control ($\rightarrow$) to a new node $\mathbb{N}_i$

(e.g., a call to a new function or jump to a case statement within the same function) or to the same node $\mathbb{N}_{i-1}$ (e.g., a for loop). That is, $E_j : \mathbb{N}_{i-1} \rightarrow (\mathbb{N}_i \vee \mathbb{N}_{i-1})$. Using these definitions, a *precise control-flow graph* for an arbitrary program P is characterized by the set of 2-tuples $\mathbb{CFG} = \{CFG_{(0,0)}, CFG_{(0,1)}, \ldots, CFG_{(m,n)}\}$, where $CFG_{(i,j)} = (\mathbb{N}_i, E_j)$. We should note that not all combinations (i, j) need to exist in a CFG. Furthermore, if a program has no dead code (unreachable code), it can be shown that the CFG is connected.

7.4.2 Control-Flow Integrity Policy

A control flow integrity policy must ensure that a program follows the intended execution path given by its CFG. Accordingly, a CFI policy defines a model for program execution such that whenever a control-flow instruction executes, it targets a valid destination in its CFG. An ideal control-flow graph will provide *all* valid target destinations for an arbitrary program.

CFI policies are therefore constrained to enforcing all possible benign paths in an arbitrary CFG. Any CFI policy can be evaluated by its ability to completely enforce the intended execution path of a program or the extent to which it completely reflects a program's CFG. Therefore, a CFI policy's precision is a fundamental metric of its coverage and/or protection. The CFI policy must losslessly enforce only valid CFG paths. The precision with which a CFI policy completely reflects a CFG is given by granularity. The granularity G of a CFI policy K is determined by how closely it reflects the precise CFG of a process P.

7.4.2.1 Precise Static CFI

We consider the granularity G of a CFI policy to be either precise or coarse. Consider the portion of a CFG for a process shown in Figure 7.1. The set of 2-tuples $\mathbb{CFG} = \{(\mathbb{N}_1, E_1), (\mathbb{N}_2, E_3), (\mathbb{N}_4, E_5), (\mathbb{N}_5, E_2), (\mathbb{N}_5, E_4), (\mathbb{N}_5, E_6)\}$ represent the static CFG. A precise static CFI approach contains a representation and enforcement of *only* these node-edge 2-tuples.

Definition 7.1 Precise static CFI. The *precise static CFI policy* K for a process P is given by strict enforcement of its $\mathbb{CFG}$, that is, $K : CFI_P \rightarrow \mathbb{CFG}$.

Figure 7.1 reflects a portion of a static CFG for a particular process, where shaded areas represent functions. Edges E_1, E_3, and E_5 represent function calls and edges E_2, E_4, and E_6 function returns. Although the CFG in Figure 7.1 depicts control flow for the process, there is insufficient information to determine the proper behavior of a return path, or backward edge. Although the CFG depicts the path $\mathbb{N}_4 \rightarrow \mathbb{N}_5 \rightarrow \mathbb{N}_3$ through edges E_5 and E_4 as viable, it is logically incorrect, as the

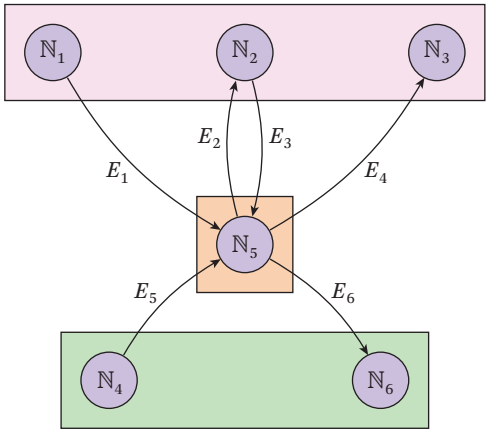

Figure 7.1 Portion of a CFG.

return target should be $\mathbb{N}_6$. As such, a CFI policy that enforces this CFG without any extra information is inherently incomplete, as backward edges are loosely handled. This is exactly the point of weakness that has been exploited in recent CFI [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Göktaş et al. 2014b, Schuster et al. 2014] and control-flow bending attacks [Carlini et al. 2015e].

7.4.2.2 Precise, Stateful CFI

Given this limitation of the CFG, it is crucial to introduce the concept of *state* and add it to the CFG.

Definition 7.2 CFG state. A *CFG state* is a set $\mathbb{S}_k = \{E_0, E_1, E_2, \ldots, E_p\}$ of valid non-jump forward edges on a CFG for a process P.

We do not include jump edges in our CFG state definition because there is no state to be recovered by the transition, i.e., they do not store a return address on the stack. Furthermore, we allow backward edges to remove elements from the state set in an orderly fashion. We combine this concept of state with the CFG to make a *precise, stateful CFG*. We define a stateful CFG to be a set of 3-tuples, $\mathbb{CFG}_S = \{CFG_{(0,0,0)}, CFG_{(0,1,1)}, \ldots CFG_{(m,n,o)}\}$, where $CFG_{(i,j,k)} = (\mathbb{N}_i, E_j, \mathbb{S}_k)$.

Figure 7.2 reflects the CFG with states added. Here, backward-edge paths are only taken if the proper state is preserved. As such, execution path $\mathbb{N}_4 \rightarrow \mathbb{N}_5 \rightarrow \mathbb{N}_3$ through edges E_5 and E_4 is now illegal because the state is not correctly preserved in execution, i.e., $\mathbb{S}_3 \neq \mathbb{S}_2$ in the stateful CFG. We call a CFI policy capable of enforcing a stateful CFG a *precise, stateful CFI policy*.

Definition 7.3 Precise, stateful CFI. The *precise, stateful CFI* policy K for a process P is given by strict enforcement of its stateful $\mathbb{CFG}_S$, that is, $K: CFI_P \rightarrow \mathbb{CFG}_S$.

7.4.2.3 Coarse-Grained CFI

A coarse-grained CFI policy is any policy that does not meet the requirements of precise, stateful CFI. Consider again the execution path $\mathbb{N}_4 \rightarrow \mathbb{N}_5 \rightarrow \mathbb{N}_3$ through edges E_5 and E_4 shown in Figure 7.2. This execution path is illegal as it does not maintain the execution state imposed by the stateful CFG. As such, a CFI policy that allows for this execution path to exist contains erroneous edges. We define $\mathbb{E}$ to be the set of erroneous edges included in the CFI policy enforcement and consider this policy to be *coarse-grained*.

Definition 7.4 Coarse-grained CFI. The *coarse-grained CFI policy* K for a process P is given by $K: CFI_P \rightarrow \mathbb{CFG}' = \mathbb{CFG}_s \cup \mathbb{E}$, where $\mathbb{E}$ is the set of unintended edges.

Corollary 7.1 Granularity of a policy. The *granularity* G of the policy K is said to increase as $|\mathbb{E}|$ increases.

At this point, it is noteworthy to mention that all CFI schemes known to the authors add some factor $\mathbb{E}$. Even the original CFI implementation for x86 considers two destinations as equivalent when the CFG contains edges from the same set of sources [Abadi et al. 2009].

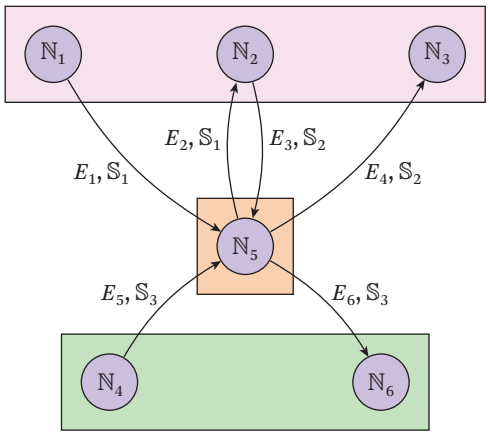

Figure 7.2 Portion of a stateful CFG.

7.5 Constructing a Precise Stateful CFI Policy

We use Figure 7.2 throughout this section to define both a precise forward-edge and backward-edge stateful CFI policy.

7.5.1 Precise Forward-Edge Stateful CFI Considerations

A precise forward-edge stateful CFI policy must be capable of strictly enforcing the intended execution path of a process according to its stateful CFG. That is, it must reflect forward-edge transitions and it must not introduce granularity by allowing erroneous edges in an execution path.

Consider the stateful CFG depicted in Figure 7.2. We define $\not\succ(\mathbb{N}_i)$ to be the set of valid targets $\{\mathbb{N}_{i+1}, \mathbb{N}_{i+2}, \ldots\}$ for node $\mathbb{N}_i$. For example, $\not\succ(\mathbb{N}_5) = \{\mathbb{N}_2, \mathbb{N}_3, \mathbb{N}_6\}$. Since this node has multiple branch targets ($|\not\succ(\mathbb{N}_5)| \geq 2$), we call it a *divergent node*. If the edges leaving a divergent node are caused by indirect jumps, such as those in a jump table, or an indirect call targeting multiple functions, the stateful CFG is unable to fully predict the behavior of the branches as this requires taking into account user input. Full computation of a process behavior under these circumstances reduces to the halting problem. This results in a lower bound in the coarseness of a forward-edge stateful CFI policy. For divergent nodes, the most precision that can be obtained in a forward-edge CFI policy is by checking the branch target of a node against the members of its $\not\succ$-function.

Consider again the CFG depicted in Figure 7.2. We define $\not\preceq(\mathbb{N}_i)$ to be the set of valid nodes $\{\mathbb{N}_{i-1}, \mathbb{N}_{i-2}, \ldots\}$ that can target node $\mathbb{N}_i$. For example, $\not\preceq(\mathbb{N}_5) = \{\mathbb{N}_1, \mathbb{N}_2, \mathbb{N}_4\}$. Since this node has multiple branch sources ($|\not\preceq(\mathbb{N}_5)| \geq 2$), we call it a *convergent node*. Function entries that are targeted from multiple indirect call instructions exhibit this behavior. A stateful CFG must then be able to encode transition information in such a way that the source of the transition can be differentiated and validated. For example, the CFI policy must reflect that $\mathbb{N}_5$ is targeted by $\mathbb{N}_1$ along edge E_1, as opposed to $\mathbb{N}_4$ along edge E_5. Failure to do so results in a coarse-grained CFI policy.

7.5.2 Constructing a Precise Forward-Edge Stateful CFI Policy

Eliminating the granularity due to divergent and convergent nodes is therefore necessary to ensure that a forward-edge CFI policy is precise. We separate our precise forward-edge stateful CFI policy into two categories: (1) indirect jumps and (2) indirect calls and indirect tail call jumps.

7.5.2.1 Indirect Jumps

An indirect jump is constrained to targeting a valid destination, as given by the stateful CFG. In compiled code, indirect jumps, with the exception of indirect tail call jumps, will always target constructs within function bounds. We require that indirect jumps may only target a member of the source node's $\not\succ$-function as given by the stateful CFG.

7.5.2.2 Indirect Calls and Indirect Tail Call Jumps

Indirect calls and tail call jumps are constrained to targeting valid destinations, as given by the stateful CFG. In portable, standards-compliant code, these destinations are function entries. We require that indirect calls and tail call jumps may only target a member of the source node's $\not\succ$-function as given by the stateful CFG. Furthermore, any additional state information about this transition must be recorded by the policy.

7.5.3 Precise Backward-Edge Stateful CFI Considerations

A precise backward-edge CFI policy must be capable of exactly enforcing the intended execution path of a process according to its stateful CFG. It must not introduce granularity by allowing erroneous edges in an execution path.

Consider the $\not\succ$-function for node $\mathbb{N}_5$ in Figure 7.2, where $\not\succ(\mathbb{N}_5) = \{\mathbb{N}_2, \mathbb{N}_3, \mathbb{N}_6\}$ and the corresponding $\not\lessdot$-function $\not\lessdot(\mathbb{N}_5) = \{\mathbb{N}_1, \mathbb{N}_2, \mathbb{N}_4\}$. In the case that node $\mathbb{N}_5$ is the epilogue of a function, the precise stateful CFI policy must be able to use state information to identify the valid return path. It must be able to utilize state information given by the forward-edge transition from a member of $\not\lessdot(\mathbb{N}_5)$ to validate the backward-edge transition into a member of $\not\succ(\mathbb{N}_5)$. For example, if the path is given by $\mathbb{N}_4 \to \mathbb{N}_5$, then the state information provided is $\mathbb{S}_3$. Only a member of $\not\succ(\mathbb{N}_5)$ with state $\mathbb{S}_3$ may be targeted, in this case $\mathbb{N}_6$.

7.5.4 Constructing a Precise Backward-Edge Stateful CFI Policy

Eliminating erroneous backward edges caused by divergent nodes in a CFG is therefore necessary for any precise backward-edge stateful CFI policy. We can enforce this policy by accurately depicting the execution path based on a program's forward-edge behavior. Resolving a valid transition for a backward edge is only a matter of restoring to the previous state in the execution path. More precisely, a precise backward-edge stateful CFI policy must only allow returns to the *most recent forward-edge transition*. As such, a precise backward-edge stateful CFI policy maintains a representation of these transitions.

7.5.4.1 Return Instructions

Return instructions are constrained to follow only the edges with a matching state as described in the stateful CFG, i.e., the code location following the call instruction that resulted in the execution of the returning function. For example, in Figure 7.2 the CFI policy must enforce the backward edge E_6 if node $\mathbb{N}_5$ was accessed using the forward edge E_5, as this maintains the state $\mathbb{S}_3$.

7.6 Hardware-Enhanced CFI: Design and Implementation

7.6.1 Overview

To restrain the execution of a program to its stateful CFG, a precise stateful CFI policy must enforce the policies outlined in Sections 7.5.2 and 7.5.4. However, a general challenge in designing a system capable of enforcing a precise, stateful CFI policy is how to encode and record the backward- and forward-edge state of a process and how to ensure efficient enforcement.

To solve these problems, we extend the instruction set of an architecure and add dedicated hardware. The Instruction Set Architecture (ISA) extensions enable dynamic creation of a stateful CFG, which in turn allows us to encode, record, and enforce precise, stateful CFI. The execution-path behavior of the program is *encoded* in our ISA extensions, where dedicated hardware is instructed to validate the forward- and backward-edge state of the program. In particular, we track both forward and backward edges by means of CFI instructions each processing a label: `cfiins lbl`. Forward-edge state is encoded by a CFI instruction, where the label (`lbl`) is a valid target determined by the CFG and recorded in a *label state register*. Backward-edge state is encoded by the execution path's forward-edge behavior as an `cfiins lbl`, where the label (`lbl`) is recorded in a *label state stack*.

A Label State Stack (LSS) is used to record backward edges to tightly couple caller/callee pairs and ensure only the most recently executed forward edge is returned to. A Label State Register (LSR) is used to record forward edges because there are inherent program semantics that prevent it from being coupled with the label state stack, such as fall-through in a case statement (see Section 7.6.4). We *enforce* precise, stateful CFI using a simple state machine supervising execution. If a violation of the stateful CFI policy is detected, a fault is triggered, resulting in the termination of the process.

The ensuing subsections describe the semantics of the ISA extensions and their interaction with the hardware subsystem. Figure 7.3 illustrates a stateful CFG for a snippet of code and accompanies Figure 7.4, which depicts the code snippet beginning at a function entry.

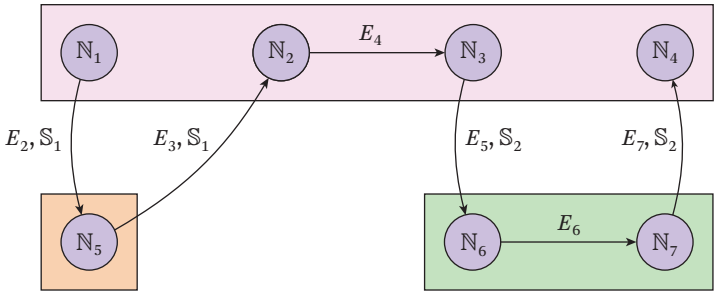

Figure 7.3 Stateful control-flow graph

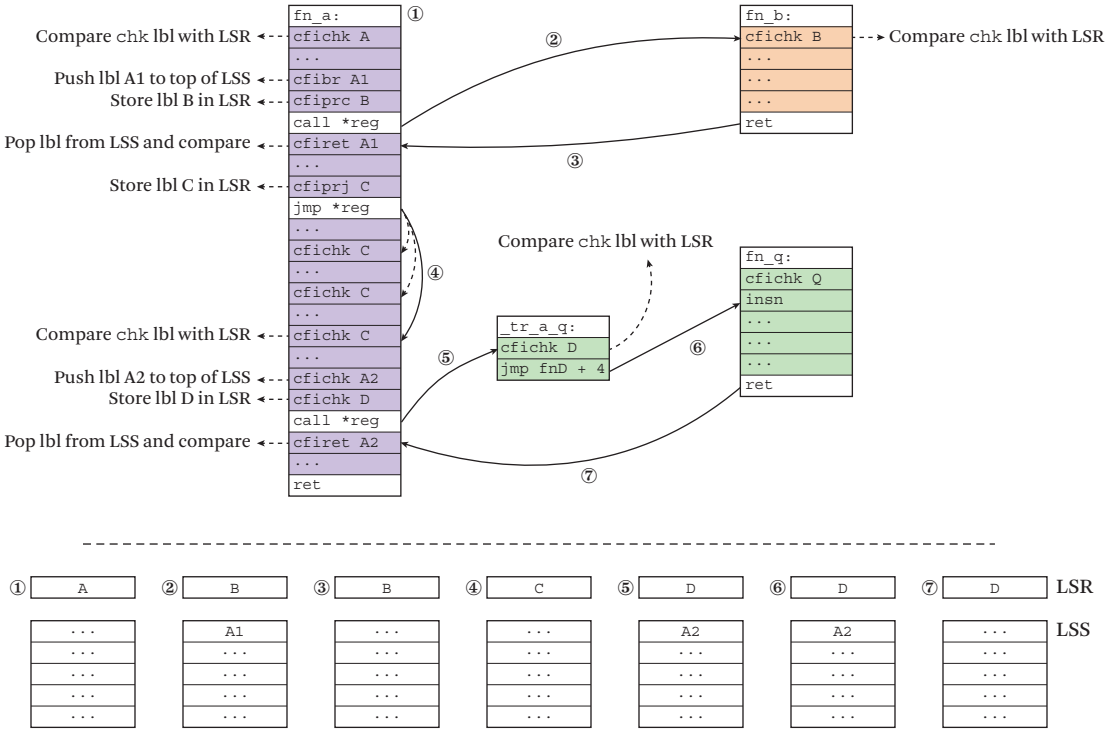

Figure 7.4 Stateful dynamic control-flow graph creation

AU: Please check circled references carefully.

7.6.2 CFI Instruction Semantics and Instrumentation

Alongside Figure 7.3, we use Figure 7.4 as an example to highlight the stateful CFI instruction semantics and their interaction with the LSR and LSS. In the code snippet, the process begins execution at the prologue of `fn_a`, labeled ①. Control

Table 7.1 **Additions to the Instruction Set Architecture**

Instruction	Syntax	Semantics
`cfibr`	`cfibr lbl`	Push `lbl` to top of LSS, flagging a call site as currently active. Unique `cfibr lbl` issued per call.
`cfiret`	`cfiret lbl`	Pop and compare `lbl` with label at the top of the LSS (returns only). Must be issued on valid return sites.
`cfiprj`	`cfiprj lbl`	Store `lbl` in the LSR, flagging intended jump target for subsequent check. Must precede indirect jump instruction.
`cfiprc`	`cfiprc lbl`	Store `lbl` in the LSR, flagging indended call target for subsequent check. Must precede call instructions and indirect tail call jump instructions.
`cfichk`	`cfichk lbl`	Compare `lbl` with value stored in the LSR. A mismatch results in a control-flow violation, triggering a fault. Must be issued in targets of indirect jumps or function entries.

flow is transferred to `fn_b` along ② and returns to `fn_a` along ③. An indirect jump into a jump table is then made in `fn_a` along ④. Control flow is then transferred to `fn_q` through a *trampoline* along ⑤ and ⑥, which returns to `fn_a` through ⑦. Aside from the trampoline, the execution path is one that may be normally encountered in an arbitrary program.

Both forward and backward edges on a stateful CFG must be checked by the CFI policy during code execution. To aid this process while reducing execution overhead, we introduce five instructions extending the ISA. Table 7.1 lists our newly added instructions. Integral to the functionality of the system are the placement and semantics of the CFI instructions, as these aid in the construction and encoding of the stateful CFG.

`cfibr` **Instruction.** The `cfibr` instruction is issued before every call. Predicated with a label, the instruction pushes its label to the top of the LSS, thereby flagging the call site as active and adding a *new state* in execution to the stateful CFG. For instance, in Figure 7.4, the `cfibr` instructions in function `fn_a` push their accompanying labels onto the LSS. This is illustrated prior to calling `fn_b`, where `cfibr A1` pushes the label `A1` onto the stack. In the program's CFG shown in

Figure 7.3, this is equivalent to encoding state S_1. On any call, a new label is added into the LSS, flagging a new call site as active and setting the new execution *state*. For a recursive function, if the last pushed label matches the label of the currently executing `cfibr`, a per label counter is incremented instead of pushing a new label into the LSS. This aids with reducing hardware overhead in the LSS memory.

`cfiret` **Instruction.** The `cfiret` instruction is issued after every call site and is predicated with a label. This label matches the label given by the `cfibr` preceding the indirect call instruction. In Figure 7.4, the call to `fn_b` is instrumented with a `cfibr A1`/`cfiret A1`, which encodes state S_1 in Figure 7.3. When a `cfiret` instruction is executed, the accompanying label is checked against the value on the top of the LSS. This evaluates the backward edge of the function, ensuring that the state of execution has been maintained during the return.

Instrumenting each instruction after every call site with a *unique label* eliminates granularity by removing erroneous edges $\mathbb{E}$. For instance, the state S_2 in Figure 7.3 is not a valid state for $\mathbb{N}_5$, or equivalently, `cfiret A2` is not a valid return target for `fn_b`. Furthermore, the hardware CFI subsystem enforces that a backward edge must target a `cfiret` instruction. Our design allows us to limit the number of return targets to only the last active call site. After a return target label has been validated, it is popped from the top of the LSS.

The functions `setjmp()` and `longjmp()` are cases using non-local gotos and would raise a false positive if instrumented using `cfibr lbl`/`cfiret lbl` instruction pairs. We did not include specific support for these functions; however, we could easily design two separate CFI instructions and simply introduce a new CFI register. One of these instructions would store the current LSS pointer into the newly added CFI register. This instruction would be issued as part of the `setjmp()` function. During execution of the `longjmp()` instruction, the register would be written to the LSS pointer using the other instruction, thus unwinding the LSS.

`cfiprc` **and** `cfiprj` **Instructions.** The `cfiprc` and `cfiprj` instructions are issued before any call or indirect jump instruction, including tail call jumps. The instruction is predicated with a label representing the valid branch target. This label is stored in the LSR and subsequently checked after branching to a `cfichk` instruction. This ensures that only valid members of the node's $\not\succ(\mathbb{N}_i)$ can be targeted. Only valid targets as determined by the CFG are encoded with matching labels. Following the example in Figure 7.4, prior to calling function `fn_b`, the `cfiprc B` instruction stores label B in the LSR. A check is performed once the branch executes and reaches the `cfichk B` instruction. The jump table in `fn_a` is validated in

a similar fashion, with the `cfiprj C` saving the label in the LSR and subsequent jump targets containing the corresponding `cfichk C` instruction. A mismatch in labels or the presence of any instruction other than `cfichk` results in a violation of control flow, and a fault is triggered.

`cfichk` Instruction. The `cfichk` instruction is issued at every function entry or indirect jump target. Predicated with a label, it checks the value stored in the LSR and performs a comparison with its predicate. This validates forward edges, which are restricted to targeting `cfichk` instructions. For instance, in Figure 7.4, when function `fn_a` calls `fn_b`, its forward-edge state is encoded as label B and captured by the LSR. The `cfichk B` encoding maps `fn_b` as a valid target for $\mathbb{N}_1$, where $\not\succ(\mathbb{N}_1) = \mathbb{N}_5$. Upon executing the `cfichk B` instruction, its label is matched against the current label stored in the LSR. Subsequent `cfichk` instructions are similarly handled.

Trampolines. A challenge with `cfiprc` and `cfichk` instructions is differentiating edges in divergent nodes that point to a convergent node. Consider the case where two divergent nodes resulting from indirect calls $\mathbb{N}_a$ and $\mathbb{N}_b$ with different $\not\succ$-functions target one common converging node $\mathbb{N}_c$. That is to say, $\mathbb{N}_c \in \not\succ(\mathbb{N}_a) \cap \not\succ(\mathbb{N}_b)$ and $\not\succ(\mathbb{N}_a) \neq \not\succ(\mathbb{N}_b)$. Instrumenting code alone with matching labels in `cfiprc`/`cfichk` to verify the edge would necessarily require instrumenting all target nodes in $\not\succ(\mathbb{N}_a) \cup \not\succ(\mathbb{N}_b)$ to share the same label. This results in breaking the inequality between both $\not\succ$-functions, effectively introducing erroneous edges in the CFG and therefore granularity. To preserve precision in the stateful CFG, trampolines are added to serve as unique bridges between these converging edges. Trampolines are instrumented with a `cfichk` instruction and a direct jump into the target function's body, bypassing the function's `cfichk` instruction. This instrumentation is illustrated in Figure 7.4, where function `fn_q` is assumed to be the target of multiple indirect calls. A trampoline `_tr_a_q` is added to serve as the indirect call target, thus precisely validating the function call.

AU: Please check the second sentence. Ok?

7.6.3 Runtime Environment

A modified runtime envinronment is needed to support both CFI-instrumented and non-CFI-instrumented software. As such, a mechanism is needed to track all CFI-instrumented processes and inform the hardware. Figure 7.5 shows a high-level overview of the software stack.

As the figure illustrates, the operating system kernel is capable of handling both CFI and non-CFI processes. This is accomplished by modifying the process

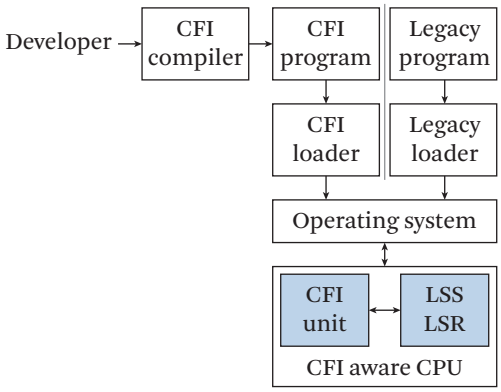

Figure 7.5 Software stack

control block in the operating system kernel to track CFI processes and signal the underlying hardware when a CFI process is scheduled. A custom CFI loader is added and utilized by CFI-instrumented software, which utilizes the kernel's syscall interface to activate CFI protection for the software in question.

7.6.4 CFI Hardware Infrastructure

As depicted in Figure 7.4, when the CFI-instrumented program executes, the newly added instructions control read/write operations on the LSR and LSS. However, to check that the CFI semantics are being precisely followed, we must supervise their execution. We propose a CFI finite-state machine (FSM) that supervises execution using the CFI instructions as input.

The primary design requirement for our dedicated CFI hardware infrastructure is to losslessly enforce any CFG with which we are provided and to efficiently enforce diverse CFI policies. Our platform does not constrict execution to a particular CFG or CFI policy. Instead, we propose a scalable, transparent subsystem capable of enforcing various CFG granularities, from precise to coarse. In this way, the vendor may choose the level of protection as determined by the security requirements of the application. Our hardware performs the necessary checks regardless of CFG coverage. The hardware will build a stateful CFG dynamically and enforce it based on the information provided during execution. It does so by recording the valid members for forward edges on the *label state register* and the valid backward edges on the *label state stack*.

It is necessary to maintain an LSS to tightly couple caller/callee pairs in order to ensure only the most recently executed forward edge is returned to. The LSR

records forward edges separate from the label state stack to handle false positives. For example, consider the jump table instrumented with precise, stateful CFI instructions in Figure 7.4, where the indirect jump stores its label `C` on the top of the label state stack. Upon reaching a valid `cfichk C` instruction, the label at the top of the LSS would be popped and matched. However, if fall-through were to occur, the next `cfichk C` instruction executed would similarly pop the label stored on the top of the LSS to be checked. This would of course trigger an error in our system, as the labels being checked would not match. It is therefore necessary to maintain separate forward- and backward-edge label state storage elements.

Label State Register. The LSR is a dedicated n-bit register accessible only by `cfiprc/j` and `cfichk` instructions. A `cfiprc/j lbl` triggers a write to the LSR. A `cfichk lbl` triggers a read from the LSR. The instruction label encodes valid targets as determined by the CFG for forward edges. Depicted in Figure 7.3, the valid forward-edge members are $\not\succ(\mathbb{N}_1) = \mathbb{N}_5$, $\not\succ(\mathbb{N}_2) = \mathbb{N}_3$, $\not\succ(\mathbb{N}_3) = \mathbb{N}_6$, and $\not\succ(\mathbb{N}_6) = \mathbb{N}_7$. These correspond to transitions ②, ④, ⑤, and ⑥, respectively, in Figure 7.4.

Label State Stack. The LSS is a dedicated $n \times m$ last-in-first-out buffer accessible only by `cfibr` and `cfiret` instructions. A `cfibr lbl` pushes the label to the top of the LSS. A `cfiret lbl` pops the label from the top of the LSS. The `cfibr lbl/cfiret lbl` pair encodes and checks stateful backward-edge targets. Backward edges are restricted to targeting valid members of the $\not\succ(\mathbb{N}_i)$ function based on the state obtained from a member of the $\not\lessdot(\mathbb{N}_j)$ function. Depicted in Figure 7.3, these states are S_1 and S_2. These correspond to transitions ③ and ⑦, respectively, in Figure 7.4. The depth of the LSS should be chosen to limit the occurence of overflowing the LSS when encountering nested functions. If an LSS overflow is detected, the contents may be written to a protected region of memory.

CFI Finite-State Machine. The CFI finite-state machine (FSM) is shown in Figure 7.6 and executes in parallel to the instruction commit stage of the processor. Placement in the pipeline at the commit stage ensures that the FSM follows the precise CPU state, i.e., all earlier exceptions/interrupts have been handled before performing CFI operations. Each transition in the FSM requires a single cycle, so the FSM state is synchronized with in-order program execution.

The initial state of the FSM assumes an arbitrary point in program execution. If the program is CFI enabled, then transitions in the FSM will occur upon encountering CFI instructions only after being notified by the OS if the process is CFI enabled. Otherwise, the FSM will remain in the initial state for the process's lifetime. Non-CFI instructions return the current FSM state to its initial state. If either `cfibr` or `cfiprc/j` are executed, then the FSM transitions to state LSS or state LSR, re-

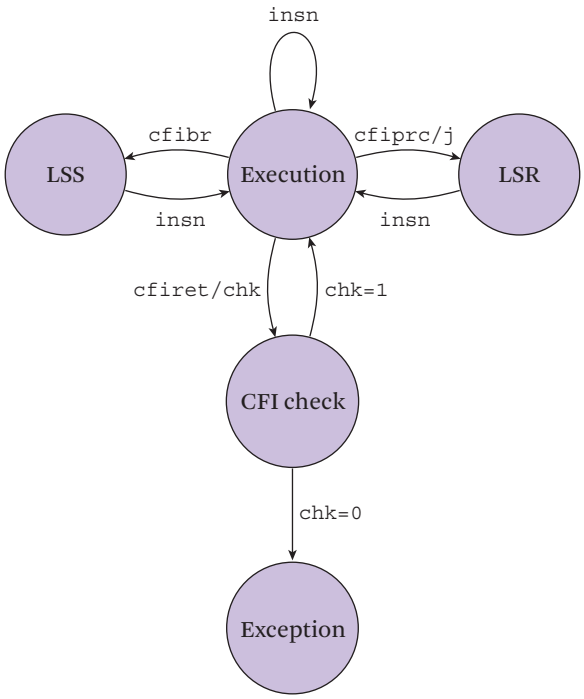

Figure 7.6 CFI FSM

spectively. This is functionally equivalent to a write because the semantics of both `cfibr` and `cfiprc/j` are to update the current label state. The state transitions to CFI check if either `cfiret` and `cfichk` instructions are executed. This is functionally equivalent to a read and compare because the semantics of both `cfiret` and `cfichk` is to check the current label state. If the label state check is validated, then the FSM state returns to the execution state; otherwise an exception is triggered.

Note that if the vendor provides any CFG then we losslessly enforce it. If in Figure 7.4 all `cfibr` labels match, or the labels at `cfichk` for `fn_b` and `fn_q` are grouped into an equivalence class, then the CFI FSM will not trigger an exception because the label state check will pass.

The CFI FSM generates control signals for reading from and writing to the LSS or LSR. It also monitors valid CFI transitions as determined by the stateful CFI semantics at runtime against the dynamically built CFG. Violations are detected if the intended execution flow, as given by the CFG, is not precisely executed. We group these violations into *execution-flow* and *logic-flow* violations. Any invalid transition in the FSM is considered a violation of `execution flow`. If a call/jump is executed,

then a `cfichk` must be targeted. Similarly, a return must target a `cfiret`. In addition, every call/jump instruction must be prefixed with a `cfiprc/j` instruction, and every call with `cfibr`. Logic-flow violations occur when an invalid label is encountered in either the LSS or LSR. For example, in Figure 7.4, if transition ② targets `cfichk Q` rather than `cfichk B`, then a logic-flow violation will be triggered.

7.7 Security Evaluation

7.7.1 Security Objectives and Requirements

The main goal of our hardware-enhanced CFI platform is to prevent code-reuse attacks. We must prevent runtime exploits that leverage either invalid backward edges, forward edges, or full functions. These include attacks that corrupt return addresses [Shacham 2007, Davi et al. 2014, Göktas et al. 2014a], corrupt code pointers used in indirect calls/jumps [Checkoway et al. 2010, Checkoway and Shacham 2010, Carlini and Wagner 2014], or reuse entire functions [Schuster et al. 2015, Tran et al. 2011]. Finally, we must prevent runtime attacks that bypass CFI while adhering to its policies [Carlini et al. 2015e].

AU: Please check the third sentence. Ok?

For our security discussion, we consider the adversary model and assumptions mentioned in Section 7.2. In particular, we assume that the application under protection has been provided with a precise CFG, and that the application has been instrumented with precise, stateful CFI instructions, as described in Section 7.6.2. We do not address the security of our hardware-enhanced CFI if given a *coarse* CFG.

7.7.2 Backward-Edge Code-Reuse Attacks

Conventional return-oriented programming attacks use backward edges (returns) to combine code sequences (gadgets) residing in the executable address space of an application to perform malicious actions. Traditionally, a memory write vulnerability is exploited allowing the attacker to inject a ROP payload, which is typically a number of return addresses each pointing to a gadget terminating in a return instruction [Shacham 2007]. Gadgets can be located at any arbitrary location in the application's program space. Recent ROP attacks [Davi et al. 2014, Göktas et al. 2014a] target only call-preceded code sequences, where a call-preceded code sequence is any instruction following a call.

Our hardware-enhanced CFI prevents backward-edge runtime attacks as described above, and in general, because they require redirection to invalid call-preceded instructions or arbitrary code locations. This is in direct violation of precise state preservation. Each call instruction is instrumented with a unique label that encodes the execution path's state information with a `cfibr lbl/cfiret lbl`

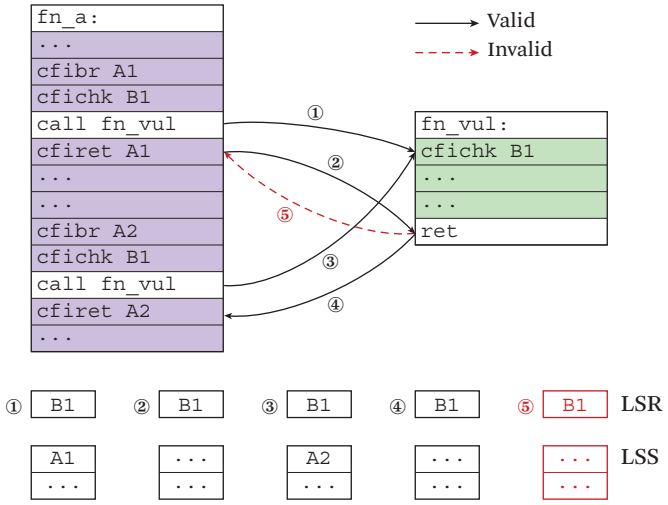

Figure 7.7 Illustrative backward-edge code-reuse attack.

instruction pair. A return instruction is only allowed to target a `cfiret` instruction if it is the most recent in the execution path history, i.e., it is a valid state. This is determined by checking the label at the top of the LSS against the `cfiret lbl` at the return target. Only `cfiret` instructions may be targeted by returns.

We use Figure 7.7 to discuss how our security requirements are fulfilled for runtime exploits that leverage these invalid backward edges. In Figure 7.7 we depict a function `fn_a` that consists of two direct function calls to `fn_vul`. The function `fn_vul` suffers from a memory corruption vulnerability that an attacker can exploit to corrupt a return address. Without CFI enforcement, the attacker can manipulate the return address to target any other instruction inside the program memory. However, in our CFI model, the return instruction must target the original caller. Our CFI state model also prevents an attacker from redirecting the control flow to a valid but currently inactive return place for `fn_vul`. As an example, assume an attacker attempts to redirect the control flow on edge ⑤. Since the valid return target, transition ④, is given by the precise state of control flow, an attacker is unable to exploit this backward edge. As described in Section 7.5.4, a return needs to target the *most recent forward-edge transition*. In our hardware-enhanced CFI, we encode the most recent forward-edge transition as a `cfibr A2`/`cfiret A2` instruction pair. We enforce precise, stateful CFI by pushing the label `A2` to the top of the LSS prior to any call and constraining the callee to returning to a `cfiret` instruction with a matching label.

7.7.3 Forward-Edge Code-Reuse Attacks

There are several variants of forward-edge runtime attacks, which typically use a corrupted code pointer to redirect control flow when dereferenced by an indirect call/jump. Jump-oriented programming attacks [Checkoway et al. 2010] use a dispatcher gadget, which acts as a virtual program counter (PC), to advance control flow through a dispatch table containing attacker-controlled addresses pointing to gadgets. These gadgets, rather than terminating with a return instruction, terminate with either an indirect call or jump instruction. Control is redirected back into the dispatcher to branch to the next gadget. ROP-without-return attacks [Checkoway and Shacham 2010] require a trampoline gadget that acts as a virtual PC to redirect control flow into gadgets terminating in an indirect call/jump. (Note that "trampoline" in Checkoway and Shacham [2010] has a different meaning than our usage.) Each terminating indirect call/jump instruction is used to point back into a trampoline wherein control flow can again be maliciously redirected. Variants of forward-edge runtime attacks exclusively target function entries [Göktas et al. 2014a, Carlini and Wagner 2014]. Typically, these are code sequences beginning at a function entry and terminating with an indirect call/jump.

Our hardware-enhanced CFI prevents forward-edge runtime attacks as described above, and in general, because they rely on either redirecting control flow to an invalid member or an arbitrary code location. Only valid indirect call/jump targets are allowed as given by their CFG members, per Section 7.5.2. This prevents the attacker from redirecting control flow to arbitrary locations in the application's progam space. Each benign call/jump target is instrumented with a `cfipr* lbl`/`cfichk lbl` pair that encodes its intended, benign target members. Redirection to an invalid member is prevented because its `cfichk lbl` state label encoding will not match the `cfipr* lbl` label in the LSR. Redirection to an arbitrary location in the application's code space will not target `cfichk` instructions.

We use Figure 7.8 to discuss how our security requirements are fulfilled for runtime exploits that leverage invalid forward edges. Within Figure 7.8 we depict a vulnerable function `fn_vuln` attempting to exploit a corrupted code pointer to redirect control flow, ③ and ④. The vulnerable function suffers from a memory vulnerability that allows the attacker to exploit a corrupted code pointer. Without CFI enforcement, the attacker can manipulate the code pointer to target either `fn_d` or an arbitrary location in `fn_e`. Our hardware-enhanced CFI prevents attackers from targeting invalid code locations via indirect calls/jumps. In our system, the valid targets for the indirect call/jump instructions are given by its valid members $\not>(\mathbb{N}_i)$ per its precise CFG. For example, valid members for the indirect call in the vulnerable function are encoded with the `cfiprc B1`/`cfichk B1` instruction pair.

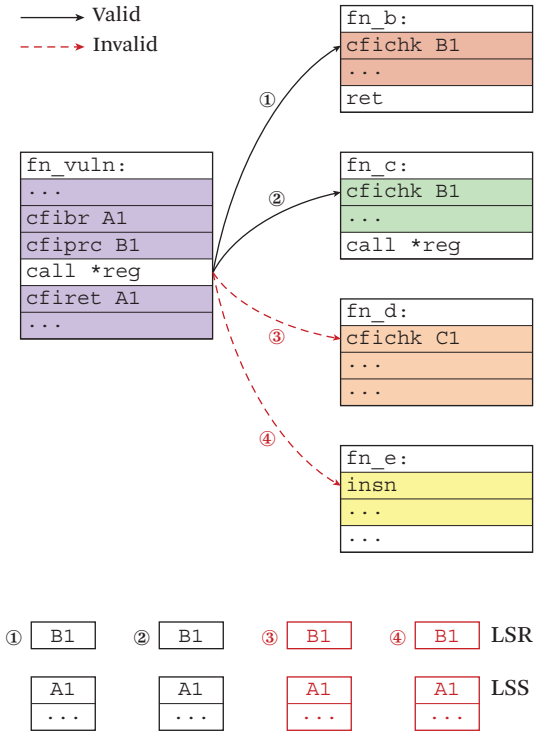

Figure 7.8 Illustrative forward-edge code-reuse attack.

We enforce precise, stateful CFI by storing the label `B1` to the LSR prior to executing the indirect call and checking that its target `cfichk B1` label matches. Only valid members, as given by the CFG, are encoded with matching labels.

Note that our hardware-enhanced CFI architecture also includes protection against dynamic code-reuse attacks, i.e., attacks such as JIT-ROP that dynamically determine gadgets on executable memory pages [Snow et al. 2013]. These attacks exploit backward and forward edges that we instrument with CFI checks based on the program's CFG.

7.7.4 Full-Function Code-Reuse Attacks

Conventional full-function reuse attacks use corrupted code pointers, along with attacker-controlled function arguments, to redirect control flow through a chain of existing `libc` functions [Solar Designer 1997a, Wojtczuk 1998, Solar Designer 1997b, Nergal 2001].

In a recent paper, Tran et al. [2011] were able to extend conventional Return-into-Libc (RILC) by demonstrating Turing-Complete RILC. Variants of full-function reuse attacks create counterfeit objects and fake virtual table pointers to redirect control flow to existing virtual methods in C++ programs [Schuster et al. 2015].

Our hardware-enhanced CFI prevents full-function reuse attacks because they rely on redirecting control flow to invalid indirect call/jump targets. Valid branch targets are instrumented with `cfipr* lbl/cfichk lbl` pairs. Only benign control-flow targets are encoded with matching labels. Redirection to an invalid control-flow target is prevented by checking that the label currently in the LSR matches the `cfichk lbl` label.

We use Figure 7.8 again to discuss how our security requirements are fulfilled for runtime exploits that leverage full functions. Within Figure 7.8 we depict a vulnerable function where the attacker may corrupt a code pointer and use it to redirect control flow to a function entry `fn_d`, ③. Without CFI enforcement, the attacker can freely manipulate the code pointer vulnerability to target any function entry in the executable address space of the application. Our hardware-enhanced CFI prevents attackers from targeting invalid functions. Valid functions are given by the precise CFG as a set of valid members $\not\succ(\mathbb{N}_i)$. For example, valid members for the indirect call in `fn_vuln` are encoded with the `cfiprc B1/cfichk B1` instruction pair. We enforce precise, stateful CFI by storing the label `B1` to the LSR prior to executing the indirect call. We check that its targeted `cfichk lbl` label matches what is currently stored in the LSR. Only valid members, as given by the CFG, are encoded with matching labels.

Note that COOP attacks can be prevented in our design if the class hierarchy is correctly and precisely covered in the CFG. For instance, Google compiler extensions can be leveraged to extract such precise CFG information [Tice et al. 2014].

7.7.5 Control-Flow Bending

A recent attack [Carlini et al. 2015e], called Control-Flow Bending (CFB), demonstrates code-reuse attacks are possible while adhering to fully precise static CFI. In a CFB attack, attackers may corrupt a code pointer to call a valid function entry where a vulnerability exists, allowing it to corrupt a return address. They then may use the corrupted return address to return to any call-preceded site. In particular, CFB exploits any function with a vulnerability that can overwrite its own return address and adheres to CFI by returning to any location where this function was called.

Our hardware-enhanced CFI prevents CFB attacks because it requires redirection to any call-preceded slot in a stateless CFI-protected system. We offer

precise, *stateful* CFI so that only the most recently executed forward-edge transition may be returned to. As described above, this is ensured with a unique `cfibr lbl/cfiret lbl` instruction pair. A return instruction is only allowed to target a call-preceded slot if it is the most recent in the execution path history. Using Figure 7.7, the invalid return ⑤ is prevented from returning to `cfiret A1` because it is not the most recent call-preceded slot.

7.7.6 Security of Label State Stack/Register

Even though it is not a strict security requirement, our design only allows CFI instructions to access the LSS and LSR and avoids CFI data being loaded to main memory. Recall the recent CFI attack that corrupts offset pointers referencing a CFI jump table spilled to the program's stack for efficiency reasons [Liebchen et al. 2015]. We prevent this attack, and similar attacks that corrupt or disclose CFI data, by storing CFI-related data, such as labels, in a dedicated memory, LSS, and LSR.

7.8 Performance Evaluation

To evaluate the support of our hardware-enhanced CFI protection, we generated custom build tools, a custom runtime environment, and custom hardware infrastructure. This enabled us to (i) issue newly added CFI instructions in proper code locations, (ii) create unique CFI labels for any arbitrary application, and (iii) support CFI services within a rich OS environment on a hardware platform.

7.8.1 Build Tools

A set of modified build tools are needed in order to issue the necessary CFI instructions in the proper places. As such, in order to test the performance of our system, we developed an instrumented toolchain based on the GNU Compiler Collection (`gcc`) version 4.9.2, the GNU Binary Utilities (`binutils`) version 2.23 and μClibc (`uClibc`) version 0.9.33.2.

The compiler, `gcc`, was made to issue the `cfibr` and `cfiprc` instructions before any function call, with a corresponding `cfiret` instruction at the return site. Similarly, a `cfiprj` was issued before indirect jumps with a `cfichk` at function entries and at indirect jump targets. The assembler, `gas`, was modified to recognize these new instructions and emit the necessary machine code. As the C library, `uClibc`, is built by the compiler, only those routines written in assembly need to be manually instrumented; the others can be directly compiled without issue.

For testing and CFG instrumentation purposes, we wrote an IDA Pro plugin that extends the SPARC processor module bundled with the program. This enabled us to

automatically instrument backward edges in our binary. Forward edges for indirect jumps were instrumented by manually extracting jump table information from the binary and feeding this information to the plugin. For indirect calls, a new section with trampolines was added to the binary at compile time using a custom linker script. The trampoline was instrumented with the proper check instruction and an indirect jump to the target function. Using the plugin, indirect calls were rewritten as direct calls to the proper trampoline. As this instrumentation is equivalent to the one presented in section 7.6.2, it is sufficient for performance-testing purposes. We should stress, however, that this chapter does not present a general solution to CFG generation and limits itself to providing a mechanism that can be used with manual analysis to generate an estimate of the CFG of a program.

7.8.2 Hardware Platform

To evaluate the overhead of our CFI implementation on a real system, we integrated it into the open-source LEON3 processor distributed by the European Space Research and Technology Centre [Gaisler Research 2017]. The LEON3 is a 32-bit processor that implements the SPARC V8 ISA [SPARC 2017]. The synthesizable LEON3 core is equipped with a 7-stage pipeline, separate instruction and data caches, memory management unit, hardware floating-point units, AMBA 2.0 AHB bus, and on-chip debug support.

Modifications were made to the processor pipeline to incorporate the CFI FSM, LSR, and LSS in the `iu3.vhd` module. The modified processor is implemented on the Xilinx Spartan-6 FPGA evaluation board. The FSM, LSR, and LSS are placed in parallel to the write-back stage of the LEON3 pipeline. Their operations include read/write access to the LSS/LSR and FSM operations, which are synchronized with the write-back stage so that they do not stall the pipeline. Our CFI instructions are decoded as `nop` instructions on the pipeline, which ensures single-cycle latency as determined by the SPARC V8 ISA [SPARC 2017].

7.8.3 Hardware-Enhanced CFI Evaluation Results

7.8.3.1 Performance

For the evaluation of our hardware-enhanced CFI, we used the industrial standard EEMBC's CoreMark benchmark suite [EEMBC 2017]. This benchmark suite is designed to test a processor core's functionality, namely, its pipeline, memory access, and functional unit operations. It is made up of small C programs containing read/write, integer, and control operations whose workload models several commonly used algorithms, e.g., matrix manipulation, linked-list manipulation, state-machine operation, and cycle redundancy check [EEMBC 2017]. This suite

covers usage of code pointers and frequent conditional/unconditional branching, which provides a representative class of CFI-instrumented code coverage and performance overhead comparison.

CoreMark programs are instrumented with precise, stateful CFI instructions. We follow the instrumentation described in Section 7.6.2 using our build tools.

We also evaluated several SPECInt2006 benchmarks, namely, `bzip2`, `libquantum`, and `h264ref`. These are representative example programs from the group of business, scientific, and problem-solving workloads. We did not evaluate full SPEC because of resource constraints on the FPGA evaluation board. The FPGA board provides 128 MiB of main memory, whereas full SPEC requires at least 1 GiB. Each of the programs evaluated could be run within the memory constraints imposed by the FPGA platform. Additionally, porting full SPEC2006 would require significant engineering effort in resolving all dependencies. The benchmarks we evaluated offered a reasonable trade-off in build time and coverage.

SPEC benchmarks are also instrumented with precise, stateful CFI instructions. We again follow the instrumentation described in Section 7.6.2 using our build tools.

The results are shown in Figure 7.9, where the performance overhead on average is 1.75%, with a worst-case overhead of 3.5% for SPEC benchmarks and 0.5% for CoreMark. The average code size overhead is 13.5% across both SPEC and CoreMark. We should note that this overhead is directly related to the number of calls and indirect jumps in the binaries. As more calls and indirect jumps are contained in the program, more CFI instructions need to be issued.

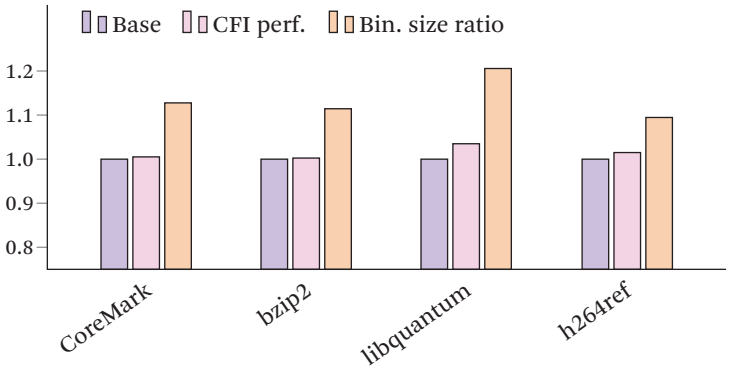

Figure 7.9 Normalized benchmark results.

AU: What does "Comb." stand for?

Table 7.2 Evaluation of Area Overhead with CFI Implemented on a LEON3 Processor

	LEON3	LEON3 CFI	Percent Change
Comb.	8,759.073	8,996.952	2.72
Sequential	16,921.416	17,143.284	1.31
Total	25,680.589	26,140.236	1.78

7.8.3.2 Area and Timing Overhead

We integrated the micro-architectural features to support our hardware-enhanced CFI design elements, as outlined in Section 7.6.4 into the 7-stage pipeline of the LEON3 processor. Our hardware-enhanced CFI LEON3 core was synthesized with Design Compiler H-2013.03-SP5-3 using the Synopsys 32/28 nm generic library, a teaching library created for micro-electronic design education. We evaluated both area overhead and maximum clock rate. In general, smaller area ensures better resource usage and lower cost requirements. A faster clock ensures our hardware will not be on the critical path or violate existing timing constraints. Table 7.2 displays the area overhead caused by extending the pipeline with full CFI protection. The total area overhead seen is negligible at 1.78%. We also evaluated the maximum frequency at which our CFI FSM, LSS, and LSR implementation could be clocked. Our hardware-enhanced CFI could be clocked up to 3 GHz without incurring timing violations.

7.9 Related Work

CFI defenses have been proposed to prevent code-reuse attacks [Abadi et al. 2009, Budiu et al. 2006, Wang and Jiang 2010, Davi et al. 2012, Zhang and Sekar 2013, Zhang et al. 2013, Bletsch et al. 2011, Tice et al. 2014, Arias et al. 2015]. In their seminal work on CFI, Abadi et al. [2009] propose a label-based mechanism. In particular, indirect branch targets are marked with unique labels. Before an indirect branch, CFI validates whether the branch targets a pre-defined label. Whereas the original CFI proposal targeted applications running on an x86-based desktop PC, CFI has been also adapted to mobile applications [Davi et al. 2012] and hypervisor code [Wang and Jiang 2010]. Unfortunately, software-based instrumentation induces too high performance penalties. Even a recent implementation utilizing an optimized shadow stack [Dang et al. 2015] adds significantly more performance overhead than our hardware-based CFI implementation and still leaves the shadow stack unprotected in main memory.

A number of coarse-grained CFI approaches aim at tackling the performance overhead. Several solutions build on behavioral-based heuristics to detect (i) the execution of short instruction sequences [Pappas et al. 2013, Cheng et al. 2014] or (ii) indirect branch counters [Yao et al. 2013, Kayaalp et al. 2013]. Other CFI schemes relax the CFI policies, most notably, they force returns to target *any* call site [Zhang and Sekar 2013, Zhang et al. 2013, Bletsch et al. 2011, Pappas et al. 2013]. However, a number of recent attacks against CFI demonstrate that neither behavioral-based heuristics nor relaxed CFI policies withstand advanced code-reuse attacks [Göktaş et al. 2014b, Davi et al. 2014, Carlini and Wagner 2014, Schuster et al. 2014]. In contrast, our hardware-based CFI scheme allows for finer-grained policies that resist these latest attacks, while being highly efficient.

Architectural fine-grained CFI support, as proposed by Budiu et al. [2006], introduced hardware support for fine-grained CFI protection via integrity checking of control-flow graph encoding. For forward-edge protection, Budiu et al. [2006] leverage a CFI label register similar to our LSR. However, for backward-edge protection, they assume a shadow stack, which incurs more performance overhead compared to our LSS. Similarly, Davi et al. [Davi et al. 2014, Arias et al. 2015] introduce hardware-assisted CFI instructions but focus only on CFI backward edges and bare metal code. In contrast to previous work on hardware-assisted CFI [Budiu et al. 2006, Davi et al. 2014, Arias et al. 2015], we support highly efficient CFI for shared libraries, multitasking, and support of legacy code [Sullivan et al. 2016].

Control-Flow Locking (CFL), as proposed by Bletsch et al. [2011], prevents CRAs by asserting a lock value before executing each indirect control-flow instruction, and de-asserting it upon entry into a valid destination. However, CFL checking is on the critical path, and applications with a larger number of control-flow instructions, such as XML parsers and interpreters, will suffer significant performance degradation. Compared with our approach, we can offer the same protection but in a very efficient way.

Branch regulation [Kayaalp et al. 2012] is a hardware-assisted CFI approach that restrains control-flow behavior dynamically. Indirect branches are forced to target function entries or function bounds, and a return should target a call-preceded instruction. A Secure Call Stack (SCS) is implemented to restrict backward-edge CFI, and each stack entry is augmented with function bounds to support forward-edge CFI. This approach shares many similarities with ours but fails to handle dynamically linked libraries, stack unwinding, tail call optimization, and compatibility with non-CFI-instrumented programs. In contrast, our approach does not need knowledge of function bounds to enforce a policy, as this is determined via the

CFI label. Branch regulation requires the storage of bounds data to be included in the program executable.

Onarlioglu et al. [2010] eliminate unaligned indirect control-flow instructions from a program with the insertion of `nop` sleds. The remaining indirect control-flow instructions are then secured by enforcing that they can only be executed by means of an aligned entry. Return address encryption is implemented to prevent backward-edge CRAs. In addition, per-function cookies are used to constrain indirect jumps to the function's bounds. This approach reports a large increase in both binary file size and performance overhead.

7.10 Conclusion

Within this chapter we present the formal underpinnings of a precise stateful CFI policy, which enabled the design and implementation of a lossless, scalable, and highly efficient hardware-enhanced CFI platform. The new framework leverages dedicated CFI instructions to losslessly enforce any CFG and diverse CFI policies within our model. Our hardware-enhanced CFI significantly lowers the performance overhead when applied to several SPECInt2006 and CoreMark benchmarks. Further, if provided with a precise CFG we show comprehensive protection from many traditional and recently proposed code-reuse attacks. The goal of our work is the design and implementation of a hardware-enhanced CFI framework that can losslessly support CFI policies with varying precision. Generation of precise CFGs for real-world applications remains an open challenge.

Acknowledgments

This work was partially supported by the U.S. Department of Energy (DE-FOA-0001386), the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP. Orlando Arias is also supported by the National Science Foundation through the Graduate Research Fellowship Program. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Energy or the National Science Foundation.