TU WIEN Informatics

Implementations, Improvements & Implications of Hardware-Assisted CFG-Based Control Flow Integrity Schemes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Mario Telesklav, BSc.
Matrikelnummer 01252288

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Univ.Ass. Dipl.-Ing. Stefan Tauner

Wien, 9. März 2021

Mario Telesklav	Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Implementations, Improvements & Implications of Hardware-Assisted CFG-Based Control Flow Integrity Schemes

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Mario Telesklav, BSc.
Registration Number 01252288

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Univ.Ass. Dipl.-Ing. Stefan Tauner

Vienna, 9th March, 2021

Mario Telesklav	Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Mario Telesklav, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. März 2021

Mario Telesklav

Danksagung

Ich möchte insbesondere meinem Betreuer Stefan Tauner für seine Anleitung bei der Erstellung dieser Arbeit und für all unsere bereichernden Diskussionen danken. Sein Feedback hat wesentlich zur Qualität beigetragen. Des Weiteren bin ich Prof. Steininger für die Beratung zum Thema der Arbeit und die Möglichkeit, sie in seinem Forschungsbereich schreiben zu können, sehr dankbar.

Außerdem möchte ich mich bei meinen Freunden und meiner Familie für ihre Unterstützung in jeder Hinsicht während meines Studiums bedanken, vor allem in herausfordernden und stressigen Zeiten. Sie haben diese Arbeit überhaupt erst möglich gemacht.

Acknowledgements

I would like to especially thank my supervisor Stefan Tauner for his guidance throughout creating this thesis and all of our enriching discussions. His feedback has contributed greatly to the quality of the final work. Another thank you goes to Prof. Steininger for advising on the topic and the possibility of writing it in his department.

Moreover I want to thank my dear friends and family for their support in all respects throughout my studies, particularly in challenging and stressful times. They made this work possible in the first place.

Kurzfassung

Die Möglichkeit, Speicherbereiche mit böser Absicht zu überschreiben, ist immer noch eine ernstzunehmende Sicherheitslücke vieler moderner Systeme. Diese Schwachstellen ermöglichen Code-Reuse Attacks (CRAs), bei denen bereits existierender Code für einen Angriff anstelle von eingeschleusten Anweisungen verwendet wird. CRAs sind in den letzten Jahren besonders relevant geworden. In CRAs leiten Angreifer den Kontrollfluss eines Programms so um, dass er vom beabsichtigten Control Flow Graph (CFG) abweicht, der alle gültigen Sprünge, Funktionsaufrufe und Rücksprünge in einem Programm darstellt. Eine weit verbreitete Maßnahme gegen solche Angriffe ist es, Control Flow Integrity (CFI) sicherzustellen. In den letzten 15 Jahren hat es dazu viele Vorschläge für Implementierungen sowohl in Software als auch in Hardware gegeben. Das zugrunde liegende Ziel ist immer zu überprüfen, ob die Programmausführung dem vorgesehenen CFG entspricht, indem die Gültigkeit der Ziele von indirekten Sprüngen zur Laufzeit sichergestellt wird. Hardware-unterstützte Ansätze weisen normalerweise eine bessere Performance auf als Implementierungen in Software. Die meisten existierenden Ansätze wurden jedoch auf unterschiedlichen Plattformen und mit unterschiedlichen Benchmark-Anwendungen evaluiert, was quantitative Vergleiche der vorgestellten Ergebnisse kaum aussagekräftig macht.

In dieser Arbeit gehen wir zunächst auf die Grundlagen von CRAs ein und geben einen Überblick über CFI im Allgemeinen. Wir konzentrieren uns anschließend auf Hardware-unterstützte CFI-Ansätze und beschreiben fünf solcher Schemata im Detail, die verschiedene Schutzkonzepte einsetzen. Unser Hauptbeitrag besteht in der Implementierung der verschiedenen Ansätze auf einer gemeinsamen Plattform, um einen aussagekräftigen quantitativen Vergleich zu ermöglichen. Außerdem präsentieren wir ein neues Schema, das auf den unserer Meinung nach besten Konzepten aufbaut und diese verbessert.

Unsere Ergebnisse zeigen, dass CFG-basiertes Durchsetzen von CFI in Hardware mit geringeren Kosten in Bezug auf Laufzeit und Codegröße erzielt werden kann, als zuvor in akademischen Konzepten gezeigt wurde. Zugleich werden anderswo vernachlässigte Features berücksichtigt und die Hardwareanforderungen auf einem ähnlichen Niveau gehalten.

Schlüsselwörter — *CRA, ROP, JOP, CFG, CFI, RISC-V, Software Security, Shadow Stack, Control Flow Transfer, EXCEC*

Abstract

Potential memory corruption errors are still a serious vulnerability of many modern systems and can be maliciously exploited. Code-Reuse Attacks (CRAs), where code already present in some application is used for an attack instead of injected instructions, are enabled by such errors and have become particularly relevant in recent years. In CRAs attackers redirect the control flow of a program in a way that it deviates from its intended Control Flow Graph (CFG), that represents all valid jumps, function calls and returns in a program. The enforcement of Control Flow Integrity (CFI) is a well-established mitigation technique against such CRAs, backed by many proposals for implementations both in software and hardware in the past 15 years. Their underlying goal is always to check whether program execution follows its intended CFG by checking the validity of control flow transfer targets at runtime. Hardware-assisted approaches usually have a better performance compared to implementations in software. However, most schemes proposed by academia were evaluated on different platforms and different benchmark applications were used, which makes quantitative comparisons of the presented results hardly meaningful.

In this work we first reprise the basics of CRAs and give an overview of CFI in general. We then focus on hardware-assisted CFI approaches and describe five concrete schemes in detail, that cover a wide range of different protection concepts. The main contribution of this work is a meaningful quantitative comparison based on our implementations of these approaches on a common platform. In addition, we present a novel hardware-assisted CFI scheme called EXtensive CFI Enforcement Concept (EXCEC), which is based on what we consider the best concepts and ideas of the existing approaches. EXCEC improves some details and offers a more sound protection than most other schemes.

Our final results show that hardware-enforced CFG-based CFI can be achieved with significantly lower overheads on runtime and code size than previously presented in academic concepts, while still implementing features that have been neglected elsewhere and keeping hardware utilization at a similar level.

Keywords *— CRA, ROP, JOP, CFG, CFI, RISC-V, Software Security, Shadow Stack, Control Flow Transfer, EXCEC*

Contents

CHAPTER 1

Introduction

To this day, errors in programs can still be exploited on most processors. Vulnerabilities, which enable the manipulation of memory contents, are particularly common. We refer to such errors as *memory errors* in the remainder of this work. An attacker typically wants to change program behavior, e.g., in order to execute malicious code with the privileges of some application under attack. Memory errors can be exploited to that end. The 2020 Common Weakness Enumeration (CWE) by MITRE[1] reports such vulnerabilities as the second most common family of software errors [MIT20]. Recent statistics released by Microsoft and Google show similar results. According to their statistics, about 70 % of all security bugs are related to memory errors [ZDN20].

Especially in the embedded world with the Internet of Things (IoT) gaining more and more attention, where Central Processing Units (CPUs) need to be small, efficient and cheap and sometimes no operating system (OS) support for advanced security features is available, this becomes an increasingly serious threat.

While it is not always easily possible to prevent memory errors, concepts for detecting and limiting the consequences evolved in recent years. Their main goal is to detect (intentional and malicious) deviations from normal program behavior enabled by software flaws. This thesis deals with Control Flow Integrity (CFI) and more specifically with hardware-based approaches thereof, which are considered an effective and promising approach for enforcing correct program execution.

We first describe the underlying problem in this chapter. In Chapter 2, the concept of CFI is explained in general with a later specialization on hardware-based CFI in Chapter 3, where some state-of-the-art approaches are described. We then present implementations thereof and the design of a novel and extensive CFI scheme, which is based on joined concepts and ideas of existing approaches, in Chapter 4. Chapter 5

[1] http://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

describes how programs can be enriched with additional instructions required for CFI. An extensive quantitative analysis of multiple CFI implementations is presented in Chapter 6. Chapter 7 follows with a final conclusion on the topics covered in this thesis.

1.1 Motivation

Memory unsafe languages like C and C++ potentially entail memory errors because there is no automatic bound-checking on arrays and pointer references. A particularly relevant error is *memory corruption*, where an attacker is able to overwrite memory locations with arbitrary contents. Such errors are present in a vast number of applications [dCV17].

Table 1.1 lists some unsafe but frequently used functions of the Standard C Library, which can be exploited by an attacker. It is up to the programmer to handle these functions with care [BST+00].

Function prototype	**Potential problem**
`strcpy(char *dest, const char *src)`	May overflow the `dest` buffer
`strcat(char *dest, const char *src)`	May overflow the `dest` buffer
`getwd(char *buf)`	May overflow the `buf` buffer
`fscanf(FILE *f, const char *format, ...)`	May overflow its arguments
`scanf(const char *format, ...)`	May overflow its arguments
`sprintf(char *str, const char *format ...)`	May overflow the `str` buffer

Table 1.1: Some unsafe functions in the Standard C Library [BST+00]

A very simple usage of such an unsafe function is shown in Listing 1.1. A buffer of limited size is allocated on the stack and later filled with user input from the `scanf` function. If the string read by `scanf` is longer than the buffer size, data is written beyond the allocated memory resulting in what is called a *buffer overflow*. This overwrites subsequent memory locations on the stack because stacks usually grow downwards.

```
1    char buffer[8];
2    scanf("%s, buffer);
```

Listing 1.1: Potential buffer overflow

We show an exemplary stack layout during an arbitrary function call in Figure 1.1. First, some registers of the caller are saved - including the return address where the function should jump to after finishing. Parameters for the callee follow. Local variables are pushed to the top of the stack (meaning to the lowest address) during function execution.

When a fixed-size buffer is declared (as previously shown in Listing 1.1), its size is allocated on top of the stack as a local array. Functions like `scanf` read data into that buffer, but don't check bounds. A buffer overflow occurs when the buffer size is exceeded, leading to overwriting data written on the stack before (e.g., the function's return address).

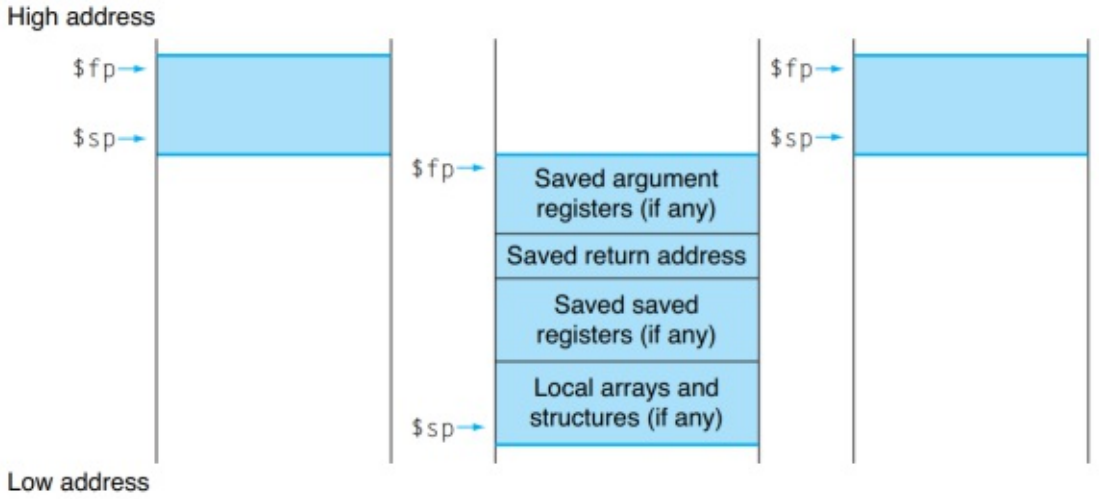


Figure 1.1: RISC-V stack layout during a function call [PH]

Stack-based buffer overflows, such as shown in the example above, are arguably the most prominent example for memory corruption vulnerabilities. But there exist other forms as well. Literature lists heap-based buffer overflows, format string vulnerabilities, use-after-free vulnerabilities and numeric problems such as integer overflows as some other potentially exploitable weaknesses [SWK+16, SPWS13].

Many of such errors could be prevented by attentive programmers. For example, there exist safe alternatives for the functions listed in Table 1.1 and manual bound checks could often be applied. An essential problem here is that it is not feasible to update and rework huge legacy code bases. In addition, some compilers offer features such as *address sanitation* for detecting some sorts of memory corruption. These can usually only be used with considerable run-time overhead though [SBPV12]. Unfortunately, there exists no silver bullet for immediate prevention of memory corruption errors.

A great deal of academic and industrial effort therefore went into techniques for detecting the *consequences* of attacks rather than preventing them in the first place. In [CBP+15], defense methods are classified into *prevent-the-corruption* and *prevent-the-exploit* approaches. We will present different techniques for the latter in the subsequent chapters of this thesis.

Memory errors are often triggered by mistake, e.g., by overwriting the return address and causing the program to jump to an unintended position or accessing some portion of memory after it has been freed. This typically leads to unexpected behavior and, most prominently, program crashes. But memory errors can also be exploited.

Figure 1.2 gives an overview of attacks enabled by such vulnerabilities. These attacks can be roughly classified into four families where most of them require different mitigation concepts: *Code corruption attacks* try to modify existing code. *Control-flow hijack attacks*

want to alter program behavior by changing the control flow. *Data-only attacks* aim to change program logic, e.g., by manipulating flags used for branches. And finally *information leaks* are used to gain access to data otherwise never shown to the user, such as passwords [SPWS13].

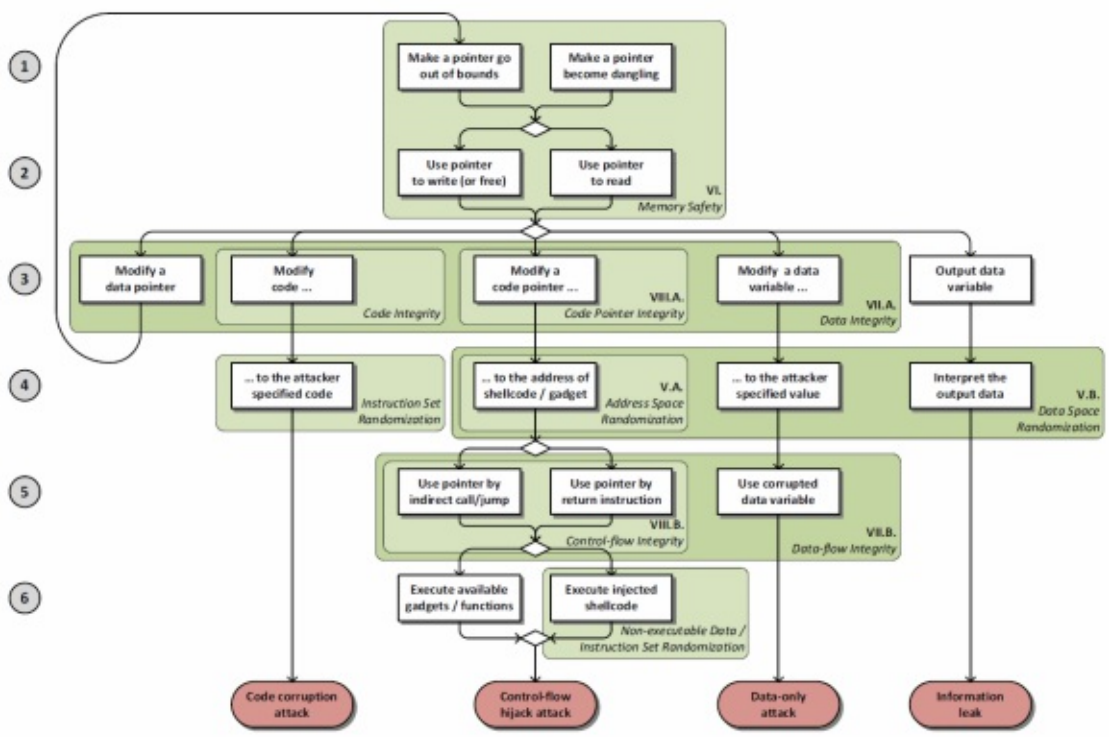


Figure 1.2: Attack model with different kinds of exploits [SPWS13]

We focus on the second of beforementioned attack families in this work, i.e., on attacks aiming to hijack an application's control flow. Such attacks are not an end in themselves though but usually aim to execute malicious code of the adversary's choice. In a common memory corruption attack, two goals need to be achieved. The attacker has to inject malicious code and must manipulate the legit control flow in a way that it is redirected to the payload [CPM+98]. Mitigation techniques for code injection attacks exist and are widely used in commercially available computers. Such are, for example, Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) and stack canaries, which are all briefly described in Section 1.3.

There is, however, a family of attacks called Code-Reuse Attacks (CRAs) where no code needs to be injected but instructions already present in the program are used.

1.2 Code-Reuse Attacks (CRAs)

In contrast to attacks where malicious code is injected as part of the attack, e.g., with the help of a buffer overflow, CRAs exploit legitimate code that is already present in a program. In a CRA, the attacker tries to hijack the program's control flow by redirecting it to a valid address of his or her choice, thereby changing program behavior. Such code can reside in the application's sources itself but also in shared libraries like *libc*.

A typical attack consists of two tasks: First, an attacker must manipulate a program's control flow such that it leaves its intended path. This redirection can be achieved by manipulating the target address of some indirect control flow transfer in the program. In general, indirect jumps, indirect function calls (e.g., when using function pointers) and function returns can be exploited to that end. We briefly listed attacks based on memory corruption errors in the previous section. These attacks can be used to manipulate the target of an indirect control flow transfer.

As a second step, the adversary must find a way to trick the program into executing some malicious code [Sha07]. CRAs build chains of small code snippets already present in the program called *gadgets* in order to build malicious functionality. Early exploits used whole functions in *libc*, e.g., for system calls, instead of small snippets. Such attacks are called return-into-libc (RILC) attacks [TEB+11].

Three techniques have evolved in recent years for launching CRAs: Return-Oriented Programming (ROP), Jump-Oriented Programming (JOP) and Call-Oriented Programming (COP). They are not necessarily used in an isolated way but are sometimes combined [CW14]. We describe the three forms of CRAs in the following sections.

1.2.1 Return-Oriented Programming (ROP)

ROP is a special form of CRA where the adversary creates a chain of gadgets, each with a designated functionality, which all end in return statements. They can potentially start at arbitrary positions and don't necessarily begin at function entries. Listing 1.2 shows a simple ROP gadget for adding two register values.

```
1 add a4,a4,a3
2 sw  a5,0(a0)
3 sw  a4,4(a0)
4 ret
```

Listing 1.2: ROP gadget for adding two register values

The principle behind ROP was first presented by Shacham in [Sha07]. In contrast to previous RILC attacks, this approach does not require full functions as gadgets. Short code sequences ending in return statements are identified, typically only two or three instructions long.

Shacham's initial approach was limited to x86 and exploited the *geometry* of the Instruction Set Architecture (ISA), meaning the fact that x86 uses variable-length instructions.

Also, x86 is considered a very *dense* ISA: There is a high probability that any random byte stream can be interpreted as a sequence of valid instructions. Listing 1.3 shows an example thereof which is interpreted as two valid instructions on x86. If the same stream is interpreted starting only one byte later, the code has a completely different meaning, as shown in Listing 1.4 [Sha07].

```
1 f7 c7 07 00 00 00        test $0x00000007, %edi
2 0f 95 45 c3              setnzb -61(%ebp)
```

Listing 1.3: Two valid x86 instructions [Sha07]

```
1 c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
2 95                       xchg %ebp, %eax
3 45                       inc %ebp
4 c3                       ret
```

Listing 1.4: Sequence without the first byte has a different meaning on x86 [Sha07]

In order to be useful for a ROP attack, such a sequence only needs to end in a return statement. Usually this is the `RET` instruction (`C3` in Listing 1.4) on x86, but other return-like instructions are also possible. Shacham claimed that any sufficiently large body of x86 instructions, such as *libc*, contains enough gadgets for doing arbitrary computations. He identified a catalog of required gadgets, each with a well-defined operation. These are linked to a chain by using manipulated return addresses for jumping from one gadget to the next one. The memory corruption exploited for creating such a chain is also used for enriching it with register values, acting as parameters to the gadgets, at the right positions. Shacham acknowledged that there exist multiple solutions for getting a fully-functional set of gadgets, but the one he described is as follows [Sha07]:

- **Load and Store**: Loading and storing register values from/to the memory and loading constants into a register
- **Arithmetic and Logic**: Addition, negation, logic and/or, exclusive or, shifts etc.
- **Control Flow**: Conditional and unconditional jumps
- **System and Function Calls**

A group of researchers including Shacham himself generalized the concept in 2008 and showed that ROP attacks are not limited to x86 and don't require an ISA with instructions of variable length [BRSS08]. They showed that it is possible to find a Turing-complete library of gadgets in a program compiled for the SPARC ISA, which only has instructions of fixed length, and concluded that the attack affects all kinds of OSes and platforms.

A very basic example for how a ROP attack can be started is shown in Listing 1.5. The vulnerable code is in line 3, where writing to a bounded buffer is simulated with the `memcpy` function. This would often be the reading of some user input in a real-world application.

```
1 int foo() {
2     char buff[4] = {0};
3     memcpy(buff, "\x00\x00...\x00\x00\x00\x00\x00\x30\x84\x00\x1c", 20);
4     printf("input: %s\r\n", buff);
5
6     ...
7 }
```

Listing 1.5: Simple ROP attack example

When executing `foo`, at first the return address, the frame pointer and potentially other registers are pushed to the stack for later recovery. The corresponding stack layout for an exemplary machine is indicated in Figure 1.3. When entering the function, the stack pointer is first increased and then the registers, which are modified in this function, are saved to the stack. This is especially relevant for register `$ra`, which holds the return address. Then local variables are created on the stack. In this example, memory for the local `buff` array is allocated.

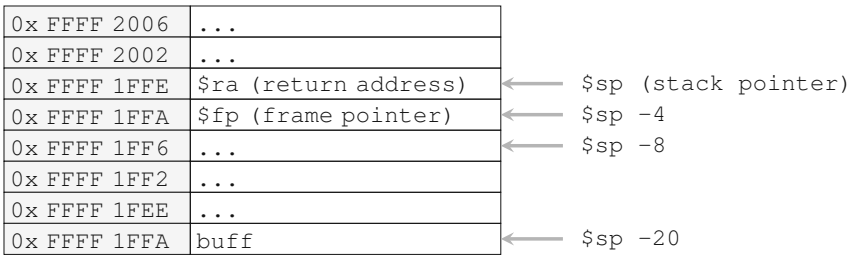


Figure 1.3: Stack layout for the C code of Listing 1.5

As `memcpy` is executed, not only 4 characters are written to `buff`, but also adjacent values on the stack are overwritten, namely the saved frame pointer and return address. Later when `foo` returns to the address saved in the return address register, it does not transfer control back to its original call site but to the address written to the register with the manipulated argument for `memcpy`.

Real-world ROP attacks are more complex and use chains of gadgets in order to perform some task. However, the basic principle - overwriting the return address - is always the same.

Figure 1.4 shows how a more advanced ROP attack based on chaining of code gadgets with the intent of making a system call with certain parameters works. First, a buffer overflow (as for example previously shown in Listing 1.5) is used to overwrite legit stack contents with the addresses of the gadgets used in the attack, the address of the final system call and the parameters required for it. The resulting stack layout after the buffer overflow is shown in the right part of the figure.

Instead of causing a return to the function's call site, the first return instruction now transfers control to `gadget 1`, which pops `parameter 1` from the stack, stores it in

register `ecx` and further jumps to `gadget 2`. The second gadget then moves this value to register `eax` and jumps back to `gadget 1`, which pops `parameter 2` from the stack and writes it to register `ebx` using `gadget 3`. Now both parameters are available in the required registers `eax` and `ebx` for the final system call, which is the target of the final jump [ZQQQ18].

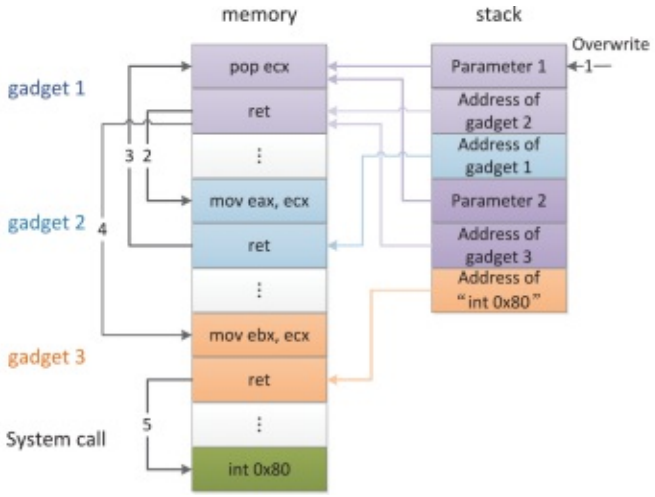


Figure 1.4: Example of a gadget chain for a ROP attack [ZQQQ18]

Both [Sha07] and [BRSS08] build their proofs of concept on gadgets found in *libc*, which offers a very extensive body of code and is present in most Unix systems. Statically-linked applications, e.g. on bare-metal systems, are less vulnerable because the availability of gadgets is limited. Given a program large enough to find a sufficient set of gadgets, such attacks are still possible though.

1.2.2 Jump-Oriented Programming (JOP)

Similarly to return statements jumping to targets based on the return address register, also indirect jumps transfer control to the address contained in some register. JOP exploits theses indirect jumps. In contrast to ROP, where gadgets must end in return instructions, the gadgets used for JOP attacks have to end in indirect jumps. This concept was first described by Bletsch et al. in [BJFL11].

The JOP approach does not rely on stack-based buffer overflows and is therefore immune to most defense mechanisms for ROP attacks. Instead of using a manipulated stack for chaining gadgets, a dispatch table (sometimes called dispatcher gadget) is required, which orchestrates jumps between gadgets [KOGP12].

The basic workings of a JOP attack are shown in Figure 1.5. At the beginning, an *initializer* is required to break out of the intended control flow of the program. The initializer manipulates the target of some indirect control flow transfer and loads values to registers required for the attack. To that end, memory vulnerabilities - such as previously

discussed - can be used. The gadget then redirects the control flow to a *dispatcher*, which is responsible for chaining the gadgets together. All code snippets end in indirect jump instructions, where their target addresses point back to the dispatcher [SAS15].

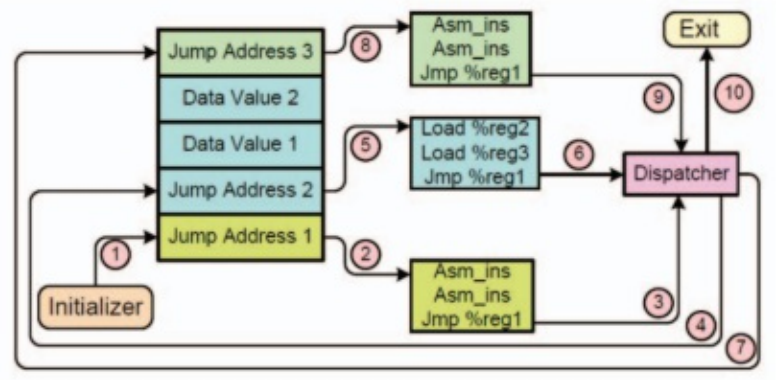


Figure 1.5: Workings of a JOP attack [SAS15]

The dispatcher gadget holds a *virtual program counter* in memory or some register, which is used for indexing the gadget addresses in a dispatch table. Note that there is no connection between the CPU's Program Counter (PC) and this counter. When control flow reaches the dispatcher, it advances the program counter. This can be as easy as making it point to the next adjacent word in memory, but also advanced functions are possible, depending on the dispatcher gadget and its associated table. The basic principle of a dispatcher is shown in Figure 1.6. The program counter is then used to index a dispatch table, which is provided by the attacker (e.g., by exploiting a buffer overflow), and holds the addresses of all gadgets required for the attack [BJFL11].

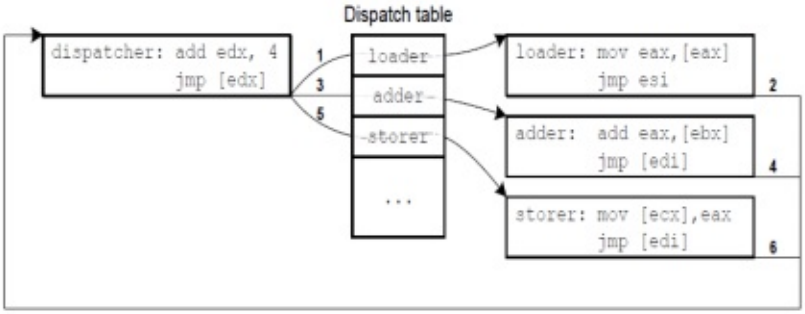


Figure 1.6: Functionality of a dispatcher gadget [BJFL11]

In C and C++ labels are used for direct jumps, which are checked by the compiler so that they transfer control only within the function. This is not always possible for indirect jumps. Gadgets used for a JOP attack are able to transfer control to arbitrary addresses also outside of the function they themselves are located in.

1.2.3 Call-Oriented Programming (COP)

Yet another form of CRA is COP, where indirect function calls are exploited. Although JOP and COP both exploit forward control flow transfers (in contrast to function returns used in ROP), COP stands out for two reasons. First and more obvious, indirect function calls are used for the attack instead of indirect jumps. Second, no dispatcher gadget is required in many cases. This is because indirect calls are memory-indirect on some ISAs, most prominently x86, meaning that the target of some indirect function call is directly determined from memory and not from some register [CW14].

Such calls appear very frequently on affected ISAs because indirect calls are also used for functions in dynamically linked libraries and *vtables* in C++ [CW14].

1.3 Defense Concepts

In order to tackle the possible attacks mentioned in Section 1.1 and Section 1.2, many mitigation techniques have been proposed by industry and academia. Each technique comes at some cost, like a performance penalty or required hardware support. Furthermore, a single technique usually doesn't offer universal protection: Each of them aims at a very specific threat model [ABEL09].

The relevance and usability of defense concepts also depends on the platform they are used in. For example, CPUs using the *Harvard* architecture don't suffer from code injection attacks in the same extent as *Von Neumann*-based CPUs because data and instruction memory are separated by design. It has been shown tough that special forms of code injection are possible as well [FC08]. Also, OS support is required for some of the approaches. We briefly list various prominent protection concepts:

- **Data Execution Prevention (DEP)** is an approach found in processors using the *Von Neumann* architecture (i.e., processors, which have a shared memory for data and instructions). The idea is to mark each memory location to be either writable (for data segments) or executable (for instructions), but never both at the same time. This way, code injected by exploiting some memory error cannot be executed. The technique is also known as Execute-Never (XN), Non-Executable (NX) Memory and W $\oplus$ X (Write XOR eXecute) [ARM15, dCV17]. DEP is supported by both AMD and Intel CPUs and nowadays commonly used by modern OSes [ZQQQ18]. In the ARM ISA, the XN bit can be used to define whether some memory region is executable [ARM15]. In principle, concepts similar to DEP also exist for bare metal systems which have a Memory Protection Unit (MPU) for enforcing certain permissions on physical memory regions [CAS+17]. CRAs cannot be prevented with DEP though because only existing code is used for the attacks.
- **Address Space Layout Randomization (ASLR)** randomly positions program segments like `.data`, `.text` and the stack in memory in order to prevent an adversary from knowing certain addresses required for attacks. Every time the

program restarts the address-space layout changes, thereby making attacks based on static addresses very unlikely. However, it has been shown that ASLR is ineffective on 32-bit architectures because addresses can be determined using brute force attacks. Also, knowledge about a single address is sufficient to deduct the segment's base address since only the starting positions of segments are randomized [SPP+04].

- **Dynamic Information Flow Tracking (DIFT)** marks data from insecure sources (e.g., user inputs) and tracks flagged values throughout the CPU pipeline and memory. Data from such untrusted inputs is considered *tainted*. The idea is to follow tainted data in order to see how it propagates (i.e., taints other data) and, most importantly, to know when it is used in dangerous ways [NS05]. DIFT can be implemented in hardware, which requires significant changes to the datapath and memory, or in software, which introduces heavy performance overhead. The method is also vulnerable to many false positives [KOGP12].

- **Stack canaries** are specific words placed on the stack next to the return address. Before returning from a function, this word is checked against an expected value. Using a buffer overflow in order to overwrite the return address changes this canary word too in most cases and leads to a detection of the attack. It is very important to choose canary words in a way they cannot be guessed, otherwise the word could be included in a buffer overflow and an attack would not be detected [CPM+98].

- **Gadget Checking** dynamically tracks the targets of indirect control flow transfers and monitors call frequency and length of the targets. Exceeding certain thresholds is considered a violation. The approach is vulnerable to a high false positive rate though, especially in programs with many small functions [ZQQQ18, CXS+09].

- **Control Flow Integrity (CFI)** makes sure that the execution flow of a program follows a certain path called Control Flow Graph (CFG), which is often determined ahead of execution (e.g., through source code analysis, binary analysis or execution profiling) [ABEL09]. To that end, run-time checks are added to the program in order to compare the actual control flow with the intended one. That affects all kinds of control flow transfers, most importantly indirect function calls, returns from functions and indirect jumps. Chapter 2 goes into detail on CFI in general and Chapter 3 describes some concrete implementations.

1.4 Aims and Contributions

The most important aim of this thesis is to establish a meaningful quantitative comparison of various existing hardware-assisted CFG-based CFI approaches, which were originally evaluated on different hardware platforms and with different sets of benchmark applications in their respective original proposals.

A further goal is designing a novel CFI scheme and comparing it to existing approaches in terms of performance, code size and hardware utilization.

The following steps are taken to achieve these goals:

1. Five existing CFI schemes are implemented and evaluated on a common hardware platform. This involves, among other things, the consideration of hardware constraints and the (un-)availability of platform characteristics like branch delay slots.

2. The best concepts and ideas of existing schemes are identified and combined in a novel design called EXtensive CFI Enforcement Concept (EXCEC), that is also implemented in hardware. These used concepts are distributed across multiple independent approaches and a combination results in an extensive and powerful CFI protection. EXCEC also improves several implementation details, comes with dedicated recursion handling, support for `setjmp` calls and also enforces CFI during Interrupt Service Routines (ISRs).

3. A concept for *instrumentation*, which is required for all hardware-assisted CFI schemes, is realized by means of a GNU Compiler Collection (GCC) plugin. Custom CFI instructions are injected at compile time based on source code. It is particularly relevant to identify the correct and precise positions for CFI instructions.

4. Almost 40 benchmarks for embedded systems tests are ported to our development platform, resulting in a representative and comprehensive set of applications. Timing of the benchmark body execution is unified for all programs and limited hardware resources on the development platform are respected. CFG information for indirect control flow transfers, which is required for instrumentation, is extracted for all benchmark applications.

Our evaluation results show that CFG-based CFI can be achieved with significantly lower overheads on runtime and code size than previously presented in other academic concepts while hardware utilization can be kept at a similar level and elsewhere neglected features can be implemented. The evaluation also hints at various weaknesses and strengths of existing schemes and allows for obtaining an optimal combination of CFI enforcement techniques.

To the best of our knowledge, we are the first to establish such comparisons. Previous works focused on qualitative evaluations and only used the overheads stated in the evaluations of the original proposals of the respective CFI schemes for comparisons. Those are of course not particularly meaningful because the evaluations were performed using completely different setups.

CHAPTER 2

Control Flow Integrity (CFI)

In recent years many approaches were presented for dealing with Code-Reuse Attacks (CRAs). A large part of them are based on the concept of Control Flow Integrity (CFI), which can be implemented on software side, with dedicated hardware support or as a mixture of both.

CFI does not prevent attacks based on flaws in software. The concept rather deals with the possible consequences thereof by detecting and consequently preventing CRAs.

This chapter first explains the underlying threat model used for CFI in Section 2.1. We then go into detail on the concept of CFI itself in Section 2.2 and also describe properties and challenges connected to the necessary Control Flow Graphs (CFGs). As CFI is an abstract concept and not limited to either software or hardware approaches, different concepts are eventually discussed in Section 2.4. The subsequent Chapter 3 later goes into detail on hardware-enforced solutions.

2.1 Threat Model

The commonly used threat model assumes that an adversary is in full control of the entire data memory of an application under attack when working on CFI [ABEL09]. This specifically applies to (but is not limited to) buffer overflows (as described in Section 1.2), which can be used to manipulate the targets of indirect jumps, indirect calls and function returns.

This threat model is mainly relevant for programs written in memory unsafe languages like C and C++. High-level languages, such as Java and C#, try to prevent this kind of directly exploitable memory errors by design due to their type safety. In case their type checking system or their interpreter are attacked, even such languages can lead to vulnerable programs though [CPM+98, dCV17].

De Clercq et al. list trusted hardware, full knowledge of memory and code layout by the adversary and the absence of other attack vectors than the possibility for usual memory corruption (e.g., buffer overflows) as additional assumptions [dCV17].

2.2 Introduction to Control Flow Integrity (CFI)

The original CFI concept referred to throughout this thesis was published by Abadi et al. in 2005 in [ABEL05] and later extended in [ABEL09]. The authors described the whole purpose of their concept as follows: "The CFI security policy dictates that software execution must follow a path of a CFG determined ahead of time". This section first clarifies some terminology and later goes into detail on the concept proposed by Abadi and his colleagues.

A Control Flow Graph (CFG) is an abstract representation of the allowed control flow of a program. A very basic example is shown in Figure 2.1. Nodes in the graph are basic blocks, which are continuous sequences of instructions. Basic blocks have exactly one entry point and one exit point, where only the instruction at the exit can change the program's control flow [dCV17]. Section 2.3 covers CFGs in more detail.

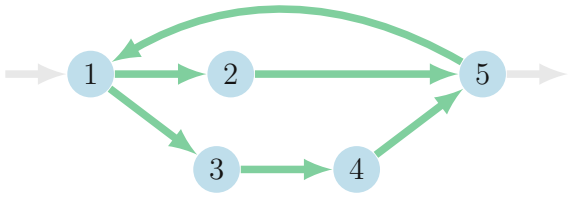


Figure 2.1: Abstract CFG example

The edges in a CFG can be classified into forward facing- and backward facing edges. Function calls and jumps, regardless whether they are direct or indirect, are considered forward edges, returns from functions are backward edges. The identifying property of the latter is that these return to the call site where the function call originated from. Loops, such as the one in the CFG example in Figure 2.1, or other forms of jumps are not considered backward edges because they are not preceded by a function call.

Burow et al. state that most CFI mechanisms are based on a two-stage process [BCN+17]. As a first step, the CFG of a program has to be generated during an *analysis* phase, which is often done in a static and offline way. The process of generating the underlying CFG is highly complex. Section 2.3 addresses this topic and points out some limitations. Then, the *enforcement* phase ensures that the actual control flow during execution is in sync with what the CFG created in the first phase allows.

We distinguish between coarse-grained and fine-grained CFI, where De Clercq et al. define fine-grained CFI as "a policy which only allows control flow along valid edges of a CFG" and coarse-grained CFI as a relaxed policy, where only certain rules are enforced, such as that the target of function calls always has to be the first instruction of a function

[dCV17]. Implementations for coarse-grained policies are usually more efficient in terms of performance overhead and significantly less complex because no CFG information is required. It has been shown though that such policies not always provide effective protection [GABP14].

Every program contains instructions which transfer control to some other position: jumps, function calls and returns. Conditional branches are considered as jumps here. Targets of such instructions can either be determined statically or dynamically. Explicit, direct jumps and calls have static targets. Those are hard-coded into the program, already known at compile time and need no further run-time checks when it comes to forward edge CFI enforcement. This is because the targets cannot be manipulated under the assumptions of a certain threat model, as presented in Section 2.1. Things get more complicated with dynamic targets, which are only determined at run-time, such as when using function pointers and returning from functions.

The CFI concept by Abadi and his colleagues suggests to add run-time checks for all of the beforementioned variants of indirect control flow transfers. The main idea is to enforce that every jump, call and return only targets one of its allowed destinations, i.e., nodes within the bounds of the program's CFG, by adding dynamic checks to the application. These checks are based on the three instructions shown in Table 2.1, which are to be understood as abstract and implementation-independent at that point [ABEL09].

Instruction	**Description**
`label ID`	Inserted at function entries. Only announces the `ID`
`call ID, DST`	Transfers control to the address in register `DST` and checks whether the `ID` matches the `ID` announced by `label ID`
`ret ID`	Transfers control back to the calling function and checks whether the `ID` matches the `ID` announced at the target

Table 2.1: CFI instructions proposed by Abadi et al. [ABEL09]

Figure 2.2 shows a small example program along with its CFG and the required CFI instructions from Table 2.1. `sort2` calls a `sort` function twice, each time with a different function pointer to some comparator function. The code of `sort` itself is omitted in the figure. This function can be any sort algorithm. The comparators form an equivalence class, indicated by sharing the same `label 17` at their entry, meaning that calling any of them is allowed from the same call instruction. At compile time, it is only known that `sort` calls any function of that equivalence class. Therefore the dynamic run-time check `call 17,R` at the function call is added, which compares the `ID` contained in the call (`17` in this example) with the `ID` at the target of the dynamic call (`label 17`).

Checking CFG-compliant function returns works similarly, but *every* function return is instrumented with the proposed `ret ID` instruction. This is due to the fact that every return is basically an indirect jump, where the jump target is popped from the stack.

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

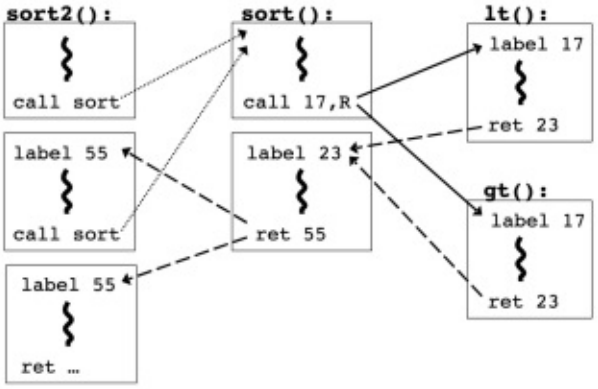


Figure 2.2: CFI example with imprecise CFG [ABEL09]

The example in Figure 2.2 also shows that CFG-based CFI still allows some control flow transfers, which are actually not valid at run-time: `sort` could return to any of its two call sites in `sort2` and could call any if the comparator methods in the equivalence class `17`. Even though only one is actually valid during execution, both are part of the legitimate CFG. This imprecision allows exploits within a valid CFG.

Figure 2.3 presents a similar CFG with different labels for the two comparator functions. However, this introduces more complexity because CFI now needs to be able to handle multiple labels for the same control flow transfer.

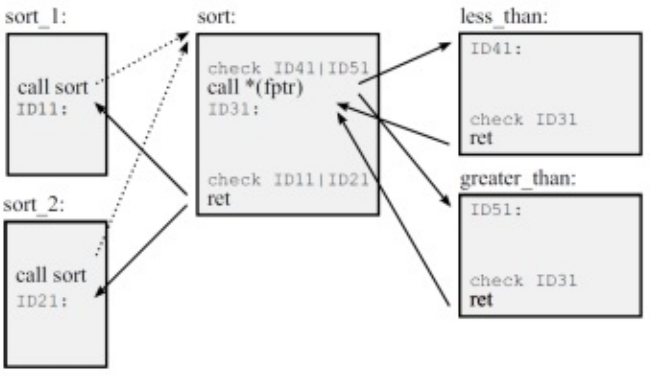


Figure 2.3: Example with imprecise CFG [GABP14]

Even the most precise CFG cannot guarantee that a function, which is called from multiple call sites, returns to the call site last used for its invocation. A CFG as the one shown in Figure 2.3 is considered being *fully-precise*, i.e., removing a single edge from the graph would break functionality. Attacks based on control flow transfers within the bounds of a CFG are called *Control-Flow Bending (CFB)*. Many defense concepts

cannot prevent CFB, and even CFI based on a fully-precise CFG does not offer perfect protection in all cases [CBP+15].

Abadi et al. proposed to complement the label-based forward edge CFI with a *shadow stack*, which is a stack only for return addresses in parallel to the actual stack. Integrity of the addresses can be checked by comparing the values on both stacks [ABEL09]. This improves CFI protection significantly [CBP+15]. We discuss both concepts, i.e., label-based checks as well as shadow stacks, in Section 3.1.

Towards a concrete implementation of the abstract instructions, the authors only discuss CFI implementations in software because hardware support for new instructions was considered unrealistic at the time of writing the paper. Listing 2.1 shows an exemplary software implementation of the `label ID` instruction for the 86 Instruction Set Architecture (ISA). The new instruction is added right at the beginning of indirectly called functions.

```
1 mov    eax, [esp+4]       ; first instruction of destination code
2
3 ---------------------------- is changed to --------------------------------
4
5 DD     12345678h          ; label ID, as data
6 mov    eax, [esp+4]       ; destination instruction
```

Listing 2.1: CFI instrumentation of a valid destination (x86) [ABEL09]

The changes required for an indirect function call are presented in Listing 2.2. Before actually transferring control to the destination, the label at beginning of the target function is loaded and compared with the label assigned to the call site. The call is only executed when no mismatch is detected, otherwise control is transferred to some error handler.

```
1 call    [ebx+8]           ; call a function pointer
2
3 ---------------------------- is changed to --------------------------------
4
5 mov    eax, [ebx+8]       ; load pointer into register
6 cmp    [eax+4], 12345678h ; compare opcodes at destination
7 jne    error_label        ; if not ID value, then fail
8 call   eax                ; call function pointer
9 prefetchnta [AABBCCDDh]   ; label ID, used upon return
```

Listing 2.2: CFI instrumentation of an indirect function call (x86) [ABEL09]

In addition, another label is defined right after the call at the position to which the callee returns. This label is used for checking validity of a return target, as shown in Listing 2.3. Instead of simply transferring control back to the return address, labels are compared again - following the same principle as before.

```
1 ret    10h               ; return, and pop 16 extra bytes
2
3 ----------------------------- is changed to --------------------------------
4
5 mov    ecx, [esp]         ; load address into register
6 add    esp, 14h           ; pop 20 bytes off the stack
7 cmp    [ecx+4], AABBCCDDh ; compare opcodes at destination
8 jne    error_label        ; if not ID value, then fail
9 jmp    ecx                ; jump to return address
```

Listing 2.3: CFI instrumentation of a function return (x86) [ABEL09]

Ideal forms of CFI, along with other security mechanisms such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), offer strong protection against a wide range of attacks [GABP14].

The CFI concept also supports security mechanisms other than the protection against CRAs. [ABEL09] lists protection against the circumvention of Inlined Reference Monitors (IRMs), Software Fault Isolation (SFI) and Software Memory Access Control (SMAC) as examples thereof. An IRM enforces some security policy (e.g., limited access to the stack) by injecting checking code directly into an application [ES00]. Similarly, SFI protects memory access operations by checking whether the respective address lies within an allowed range. The concept basically emulates a hardware-based Memory Management Unit (MMU) and thereby enables certain protection mechanisms on systems without dedicated hardware support for this feature. SMAC is a more advanced SFI approach which can enforce a wider range of security policies, such as limited access to certain memory regions for specific pieces of code [ABEL09].

The abstract idea of CFI is also used for reasons not necessarily related to malicious deviations from the intended control flow. [VTVW+17] describes how to inject faults into software for fault tolerance testing using CFG information of a program. [Van20] applies CFI techniques for detecting invalid control flow transfers due to external disturbances. For instance, electromagnetic interference can lead to bit flips, which potentially impact branch decisions and targets.

2.3 Control Flow Graph (CFG)

CFI enforcement for an application is based on its CFG. We already briefly introduced such graphs before, but to be more precise: CFGs in the context of program execution are directed and potentially cyclic graphs where the set of nodes is formed by all basic blocks of the application and the set of edges by all control flow transfers between these blocks. Exemplary CFGs were shown in Figure 2.1, Figure 2.2 and Figure 2.3.

A CFG is needed in order to perform CFI run-time checks. These checks make sure that any branch only targets nodes that are consistent with the CFG in order to detect and prevent a manipulated control flow. The principle is pictured in Figure 2.4, where CFI has to make sure that program execution only follows the CFG depicted with its

blue nodes and edges. CFG-based checks for CFI during program execution can be implemented in many ways. Some approaches are presented in the following section and in the subsequent Chapter 3.

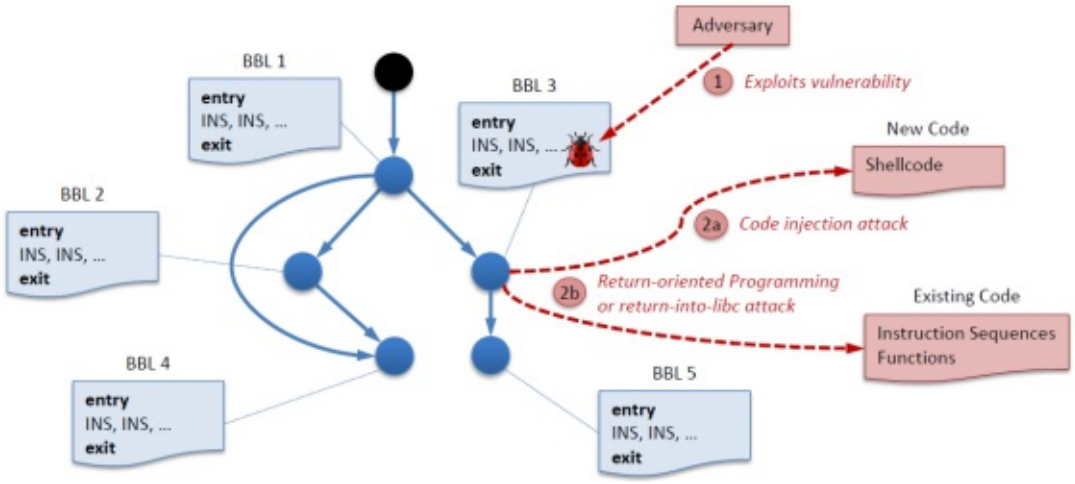


Figure 2.4: Exemplary flow for breaking out of the CFG [DDE+12]

In a preparatory step, however, the CFG must first be generated by extracting basic blocks, control flow transfers and the potential targets of these. Identification of blocks and control flow transfer is usually trivial and only depends on the instruction encoding of the used ISA.

Target nodes of direct branches can be easily derived: Those are hard coded into the application and need no CFI protection on forward edges. Also the targets of all function returns are implicitly known at run-time for most cases, even though they are indirect branches. This is because returns target their the original call site of their function in most cases. Rare exceptions to this statement are briefly described in Section 3.2. Protection concepts, such as the beforementioned shadow stack, can be effectively applied for these backward edges [ABEL09].

Compiling a set of targets for indirect jumps and calls is very complex though and not possible in all cases. Indirect calls are not only used explicitly like in the example from Figure 2.2, but also have some important implicit uses. Calls to functions in dynamically linked libraries are implemented as indirect control flow transfers since the target address is not known at compile time. Also the resolution of virtual calls in C++ with *vtables* is based on indirect function calls.

Depending on the effort made it is often only possible to generate equivalence classes for the set of valid jump or call targets [BCN+17]. One simple approach is to match the type of a function pointer with the types of all available functions. However, this often leads to a vastly over-approximated sets of potential targets.

Listing 2.4 shows a simple example for this problem. Static analysis solely based on function types is not able to find the exact call targets of the indirect control flow transfers in lines 14 and 17, although the reader intuitively knows that either of the two calls only has one possible target.

```
1  void foo(int a) {
2      return;
3  }
4
5  void bar(int a) {
6      return;
7  }
8
9  void baz(void) {
10     int a = input();
11     void (*fptr)(int);
12     if (a) {
13         fptr = foo;
14         fptr();
15     } else {
16         fptr = bar;
17         fptr();
18     }
19 }
```

Listing 2.4: Example for problems with static source code analysis [BCN+17]

More advanced analysis approaches consider the execution flow leading to an indirect jump or call and can thereby limit the set of call targets more precisely [BCN+17].

The efficiency of CFI directly depends on the precision of the underlying CFG but static analysis is not always able to derive a precise graph. However, this restriction is often accepted in the interests of simplicity [GABP14].

It is suggested to use only *one* label for CFI protection of all indirect control flow transfers in [ABEL09]. This of course introduces an extreme relaxation of the CFG, but does not require complex static analysis. Practical implementations of this idea, such as Intel Control-Flow Enforcement Technology (CET) [Int19] or Branch Regulation (BR) approaches as in [KOGP12] even go one step further and mark *all* functions with the same label, thereby ensuring that some indirect branch at least targets the very beginning of a function and not some arbitrary position. We describe the two concepts in Chapter 3.

There also exist slight improvements of this simplified CFG assumption in literature: Using different labels for function returns, indirect function calls and indirect jumps at least creates three equivalency classes. Another method for simplifying the CFG is code duplication, i.e., inlining methods in order to prevent indirect control flow transfers in the first place. The drawbacks are apparent though and make this approach not practicable in many cases [ABEL09, GABP14].

2.4 Implementing CFI

CFI can be implemented in software, with hardware enforcement or as a combination of both. After the original CFI proposal in [ABEL09], support for CFI has also been added to some compilers, tools and operating systems (OSes) in recent years [BCN+17].

Microsoft added a feature called Control Flow Guard (CFG) to its Visual C++ compiler with Visual Studio 2015 for basic forward edge protection. CFG runs on Windows 8.1 and Windows 10 and can be enabled by setting a build option, which is disabled by default. The compiler then prepares a set of valid targets for indirect function calls, which are written to a dedicated section of the binary header. During execution, the kernel checks whether indirect calls adhere to this statically determined set [msc18].

Also Clang offers optional CFI enforcement options for forward edges. Different checks can be enabled, e.g. for type-checking of pointers and some casts, but require Link Time Optimization (LTO) to be active [claa]. Indirect calls are secured by replacing the calls with jump tables, which contain only valid targets, and lead to an exception in case none of the table entries matches the dynamic call target at run-time [clab]. Clang's support for CFI is used in parts of the Android OS [and20].

Backward edge protection is neither provided by Microsoft's CFG nor by Clang's CFI concept. GNU Compiler Collection (GCC) does not come with similar concepts at all but requires third-party extensions or hardware mechanisms, such as Intel CET or ARM Pointer Authentification Code (PAC) [Cor19, Ben20, QT17].

The academic community also proposed many software implementations with different approaches. We only briefly list some distinguished techniques here. To the best of our knowledge, none of these concepts achieved recognition beyond the scientific field. The reader is referred to works such as [BCN+17] for a detailed look on software-based CFI.

HyperSafe [WJ10], which was presented in 2010, replaces indirect branch addresses with indices to a jump table and strictly safeguards the table. The concept introduces an overhead of less than 5 % on average. The **CF-Locking** [BJF11] approach from 2011 marks all indirect control flow transfers with *locks* (i.e., simple flags). The locks are set right before an indirect branch and reset immediately afterwards. Detection of CFI violation is based on these locks always being unset before an indirect control flow transfer occurs. The **bin-CFI** scheme [ZS13] proposed in 2013 is also based on jump tables and instruments indirect control flow transfers in binaries in a way that the original branch instruction is replaced with a jump table lookup. An overhead of about 4 % is entailed for C programs, respectively 8.5 % for C++ applications.

In the same year **CCFIR** [ZWC+13] was presented, which replaces indirect control flow transfers with direct jumps wherever possible, specifically for function returns, by adding *springboards* to the code. Indirect branches are only allowed within the address range of these springboards. Every outside branch, including those jumping into the springboard and from there to an external address, is a direct one. **CCFIR** has a performance decrease of less than 4 %.

MoCfi [DDE+12] follows a completely different approach by replacing branch instructions with jumps to a shared library, which is dynamically linked and performs CFI checks. The concept both protects backward edges by means of a *shadow stack* as well as forward edges by checking each indirect branch target against a set of allowed targets. The **ROPecker** [CZM+14] scheme from 2014 is implemented as an OS kernel module and analyzes control flow by triggering events at run-time. The module reacts to these events and detects CFI violations by looking for anomalies such as a high number of small, chained gadgets with a reported performance overhead of about 2.6 %.

In 2014 **MCFI** [NT14] was presented as an approach for indirect branch protection with labels similar to the original CFI concept from [ABEL09], but with the significant advantage of support for separately compiled code modules. The concept introduces about 5 % overhead.

Most software-side CFI implementations degrade performance significantly. Different studies report an average performance overhead of up to 20 % [KOGP12, CXS+09, ZWC+13] for some concepts, which is not accepted in most production environments [dCV17]. The functional scope and protection precision of course also impact performance. Meaningful comparisons for this aspect are difficult though, because different platforms and different methodologies for benchmarks are used and some of the proposals entail strong dependencies.

Other limitations introduced by some of the concepts listed above are considerably increased binary size and required OS support (e.g., for kernel modules). In addition, all CFI-related memory elements reside in the normal memory, but need to be protected form outside manipulation in some way. This is a typical task for the OS, e.g. for kernel-level protection [KOGP12].

Practical implementations of CFI in commercially available hardware are not widely used yet. The concept for Intel's highly anticipated CET was already specified in 2016, but the *Tiger Lake* Central Processing Unit (CPU) family released in late 2020 is the first to be equipped with the CFI protection technique [Gar20]. Intel Control-Flow Enforcement Technology (CET) is discussed in Section 3.3.5. ARM introduced a feature called Pointer Authentification Code (PAC) with its ARMv8.3 addition in 2016, which authenticates targets of indirect control flow transfers before using them. To that end, the upper bits of 64-bit addresses are used to store a PAC for detecting corrupted addresses. The ISA is extended with instructions for computing and checking the codes. The feature is only available for the ARM 64-bit extension though, because the standard 32-bit variant does not offer enough available bits in the registers for storing the PAC [Bra16, QT17]

Hardware-enforced CFI is covered in the following Chapter 3 and multiple approaches thereof are listed, some of which in more detail. The remainder of this work focuses on fine-grained and hardware-enforced variants of CFI because this is still a very active research area.

CHAPTER 3

Hardware-Based CFI

Literature concerning hardware-based CFI describes many different approaches. For example, De Clercq et al. compiled a survey of 21 techniques in [dCV17] with both very coarse- and fine-grained implementations.

This chapter sums up some advanced academic approaches for fine-grained CFI enforcement, both from the perspective of abstract concepts in Section 3.1 as well as embedded in concrete implementations in Section 3.3. Section 3.2 also lists challenges arising when working on hardware-enforced CFI.

3.1 Concepts

Concepts for hardware-assisted CFI protection can be classified into two groups based on the direction of control flow transfers they aim to protect.

Forward edge protection is about CFI enforcement on indirect jumps and calls. While the target is statically known for direct control flow transfers, it is only determined at run-time for indirect ones. This makes some sort of check against the intended CFG so important. Techniques for forward edge protection need to make sure that a jump or call only targets an allowed address, where the set of allowed addresses is determined at compile-time or even before.

Backward edges occur whenever some function returns. From a technical point of view, every backward edge is an indirect jump because the target is only determined at run-time and read from a register. Ideally, backward edge protection makes sure that a function return may only target the call site where the function call originated from.

It is sometimes argued that protection of forward edges is less relevant compared to securing backward edges, simply due to the fact that indirect control flow transfers on forward edges are rare in comparison to function returns [DHP+15]. In all of the

benchmark applications used for evaluations in this thesis (see Section 6.2 for a list), only about 3 % of all control flow transfers are indirect jumps or calls and 15 % are function returns. The rest are direct function calls and branches. The evaluation was performed on a RISC-V CPU with only statically linked libraries and source code only written in C. Using C++ and dynamic linking can increase the shares of indirect control flow transfers. The dominance of their direct variants is, however, not in question.

Another important property of CFI concepts is their *statefulness*. Many functions are invoked by multiple callers, which leads to backward edges to each of these call sites in the CFG. A *stateful* policy only allows returning to the most recent call site and triggers a CFI violation for all other cases, even for targets valid with respect to the CFG. Figure 3.1 depicts such a case, where the function `foo` can be called from two functions. The invocation of `foo` from function `bar`, represented by the green arrow, must be followed by a return instruction targeting `bar` again. The other backward edge constitutes a CFI violation in this case at run-time, although being present in the statically determined CFG.

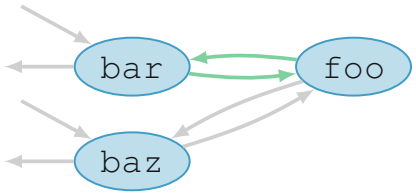


Figure 3.1: Example for stateful CFI

Some prominent concepts for CFI enforcement on forward and backward edges are listed in the following sections. Note that none of the approaches is capable of enforcing full CFI on its own.

3.1.1 Active Set

The *active-set* approach for backward edge protection keeps track of all currently active functions by marking them active in an internal array, as indicated in Figure 3.2. This can be done by assigning a unique label to each function, which is used to index the active-set. No explicit CFG is required for this approach. Every time the execution of a function starts, its flag is marked active in the active-set. Upon returning from a function, its corresponding flag is marked inactive. It is only allowed to return to active functions: Whenever a function returns to some return address, it is immediately checked whether the corresponding function at the target is currently active [dCV17, DHP+15].

Recursion needs to be specifically considered for active-sets because when the most deeply nested call returns, it would unset the flag of its function in the set. In this case, all subsequent returns of the same function fail because their respective flag is not set anymore. Counters for recursion-depth, as used in [DHP+15], are one possible solution.

Figure 3.2: Active-set example

It has been shown though that keeping a set of active call sites does not provide sufficient backward edge CFI protection. The approach is only partially *stateful* because while the set of active call sites is known, the order in which these are traversed is not. Since returning to any active call site is possible within the active-set policy, exploits like skipping functions when returning or directly jumping back to `main` are possible [TW17].

3.1.2 Shadow Stack

A shadow stack (or return address stack) is an additional memory beside the usual stack for redundantly keeping track of return addresses. This approach is used for protecting backward edges and introduces a *stateful* CFI protection. Whenever a function is called directly or indirectly, the address this function should return to is pushed to the usual stack and additionally to the shadow stack. Figure 3.3 shows the connection between the two stack elements. Right before returning from the function, the topmost values are popped off both stacks in order to get the return address and the results are compared. The return is only valid if the two addresses match [dCV17]. The basic concept of shadow stacks was already proposed two decades ago in [CH01].

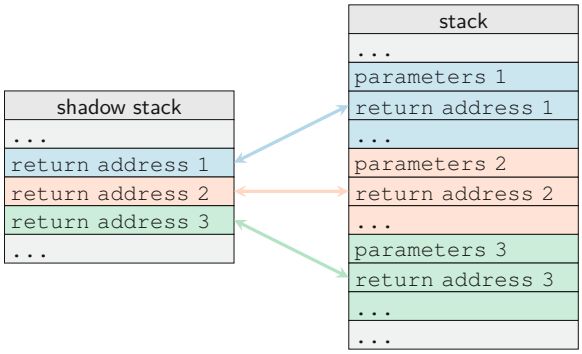


Figure 3.3: Shadow stack example

In other words, a shadow stack makes sure that a function can only return to its original call site. This is often referred to as a *call-ret* pair and constitutes perfect statefulness. It has been established that a shadow stack (or any other *stateful* approach) is required for an effective CFI protection [CCAI16]. Another advantage of this concept is that no CFG

information is needed beforehand because the allowed values and the sequence of return addresses is implicitly given by the last in - first out (LIFO) property of the stack.

However, there are constructs in C and C++ which break the strict concept of *call-ret* pairs. For example, C++ exceptions, `setjmp/longjmp` and *tail-call elimination* introduce special cases, which have to be handled separately when using a shadow stack [CCAI16]. Such challenges for CFI are listed in Section 3.2.

In addition, shadow stacks require significant amounts of memory. This can be implemented using registers, which comes with additional costs in terms of hardware utilization but memory access does not noticeably affect performance. The other possibility is some fenced off segment of the main memory, which is cheap but access is slower and it must be guaranteed that the shadow stack's memory section is protected from unauthorized modification [DBGJ19, DHP+15].

3.1.3 Labels

Label-based approaches use unique identifiers for checking whether the target of an indirect jump or call matches its intended destination. To that end, labels are added to the very beginning of functions and code blocks, which are the target of indirect control flow transfers. These labels are checked when performing the transfer. In case an attacker manipulated the control flow of the program, a label mismatch is detected [dCV17]. Depending on the implementation, the label check is performed right before jumping (as in the initial CFI concept [ABEL09]), or immediately after (like in [CCAI16]).

Listing 3.1 and Listing 3.2 show pseudo-assembly code for checking a label right after an indirect jump. The calling function sets the expected label of the callee. It depends on the concrete implementation how the indicated `setlabel` functionality is realized. The callee compares this label with the one at its entry point (indicated by `checklabel`). If the call was within the intended control flow, the two labels match.

```
1 <foo>:
2 ...
3 setlabel 0x42
4 call $reg
5 ...
```

Listing (3.1) Setting label in caller

```
1 <bar>:
2 checklabel 0x42
3 ...
4 ...
5 ...
```

Listing (3.2) Label check in callee

In principle, label checks can be used for both forward and backward edge protection. Securing a function return can be implemented in the same way as the exemplary indirect call in Listing 3.1 and Listing 3.2. There is one issue though: Backward edge protection using labels is stateless, meaning it can only check whether the return to a specific label is legitimate according to the CFG, but not if the return targets the most recent call site. This is a problem for functions, which are called from multiple call sites because a return to any of them appears to be valid according to the CFG [dCV17].

Functions, which are indirectly called from multiple different callers, are difficult for protection using labels because the callee can only carry one label check in its entry. This introduces a coarse grained CFG since all callers must use this very same label too, thus rendering them indistinguishable for the CFI check at the callee. Consider the example in Listing 3.1 and Listing 3.2 again. If `bar` is to be called not only from `foo`, all other callers must use the same label.

However, there exist solutions for this problem, such as the *trampolines* used by Sullivan et al. in their hardware-assisted CFI scheme presented in [SAD+16]. When using trampolines, indirect calls are redirected to an injected piece of code unique for each call, which performs the CFI check and then forwards control flow to the intended target, if it is considered valid. The trampoline code checks whether the target address is in accordance with the allowed CFG and triggers an exception if a violation is detected. Figure 3.5 illustrates the basic operation of a trampoline. Section 4.5 later goes into more detail on trampolines.

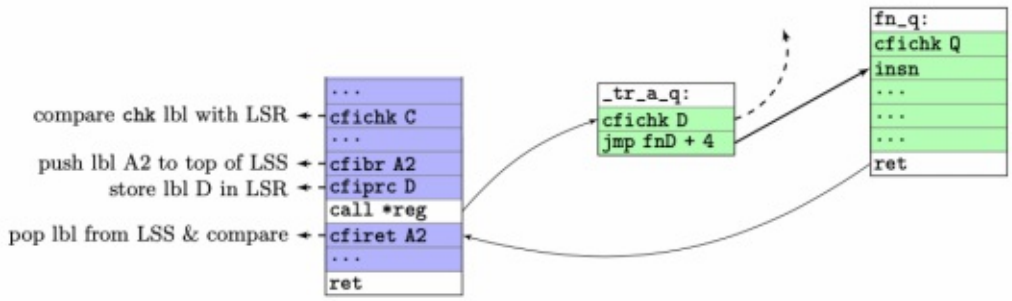


Figure 3.5: Operation of a trampoline for indirect function calls [SAD+16]

3.1.4 Policy Matrix

A policy matrix is a two-dimensional array, where one dimension corresponds to the addresses of all indirect function calls of a program and the other dimension to the start addresses of all indirectly called functions. Figure 3.6 shows an exemplary policy matrix. Each entry contains a flag indicating whether the function call at the respective address is allowed to target a certain callee. Direct calls are not part of the matrix [DBGJ19].

The matrix needs to be filled with CFG information on indirect calls at run-time, e.g. at the very start of program execution. Given the usual address width of 32- or 64 bits, neither caller- nor callee address can be used as indices to the matrix directly but need to be translated to (smaller) indices in some way. This computation can be very expensive for large matrix dimensions in terms of hardware requirements and critical path length.

Instead of using the decoded caller address as index for rows, labels can be used. These need to be known in advance, i.e., when the matrix is loaded into memory, and indirect control flow transfers are to be annotated with the labels.

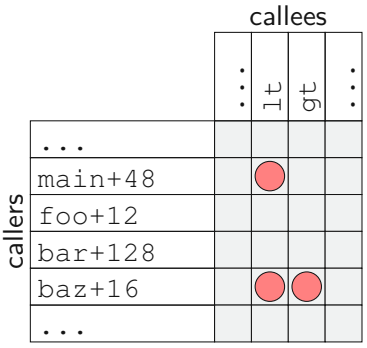


Figure 3.6: Policy matrix example

Similarly to labels, a policy matrix can be used for both forward and backward edge protection in principle. However, the same limitation, namely the statelessness of backward edge CFI checks, applies.

3.1.5 Further concepts

For the sake of completeness we briefly list some further hardware-based CFI concepts in this section, which are either considered being too coarse grained or do not use CFG information. We make this limitation given the title of this thesis.

Branch Regulation (BR)

BR defends against CRA attacks by checking if control flow transfers target an addresses within the same function (i.e., for local jumps) or the very beginning of another function (i.e., function calls). The concepts thereby prevents jumps to arbitrary addresses, making CRA attacks with chains of gadgets significantly harder. BR does not require any static analysis steps or knowledge about the program's CFG but only enforces coarse-grained CFI [KOGP12].

The concept blurs with the original label-approach described in Chapter 2 when only a small number of different labels is used.

Cryptographic approaches

Cryptographic approaches use encryption or signing techniques for CFI enforcement and not necessarily rely on CFGs. The main idea is to detect manipulated control flow target addresses by means of cryptography. Such concepts are able to provide very fine grained CFI but involve challenges like a significant performance- and hardware overhead. In addition, the protection of keys needs to be considered [ZQQQ18].

Two examples for this class of CFI enforcement techniques are HCIC and SOFIA: HCIC uses cryptography for detecting manipulated branch targets and solves the problem of protecting encryption keys by using physical unclonable functions (PUFs) in a very

innovative way [ZQQQ18]. SOFIA encrypts instructions with information dependent on the control flow. When executing the program, control flow information is again used for decryption and only paths are able to perform a valid decryption. Although introducing a very fine-grained CFI, SOFIA entails more than 100 % runtime overhead [DCGÜ+16].

Finite State Machine (FSM)

Concepts based on Finite State Machines (FSMs) check for correct system behavior by only allowing certain sequences of execution, which are represented by internal states. Implementations thereof can be very coarse-grained, i.e., might only check specific properties like the fact that function calls are only allowed to target the first instruction of some function, quite similar to BR.

More fine-grained approaches like the one presented in [RKHB12] enforce a secure sequence of function calls by comparing the call order at runtime with a pre-computed model of a correct execution.

3.2 Challenges

Implementing the concepts described in Section 3.1 entails to consider and solve some challenges, which are briefly described here.

- **Recursion** introduces problems for CFI [CCAI16, DHP+15]. Even in small programs, recursive function calls can easily stack up very high and thereby overflow a shadow stack. The active-set approach on the other hand is not vulnerable to overflows, but does not keep track on recursion depth. This introduces some imprecision for returns from recursively called functions.

- **Tail-Call Elimination** is a compiler optimization, which introduces an exception to the common principle that every function returns to the site it was called from. When enabled, the optimization generates code for certain function calls, where a function does not return to its own call site but to the call site of its caller. Naturally, this breaks backward edge CFI with shadow stacks [CCAI16]. Ways to handle tail-call elimination when enforcing CFI are, for example, disabling this compiler optimization (e.g., with the GCC flag `-fno-optimize-sibling-calls`) or excluding the affected code from instrumentation.

- **C++ exceptions** also don't follow the principle of *call-ret* pairs. When an exception occurs, the function causing it jumps to the end of the `try` block by unwinding the stack until the address of the last instruction in this block is found without executing the function returns in between. The simplified control flow thereof is depicted in Figure 3.7. If not specifically handled, this leads to inconsistencies between shadow stack and the common stack [DZL16].

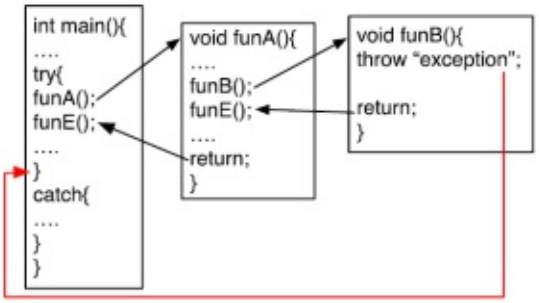


Figure 3.7: Simplified execution flow with a C++ exception [DZL16]

- **`setjmp/longjmp`** is a language feature of C for performing jumps across function boundaries. A call to `setjmp` stores the current program state, e.g., important registers, in a buffer. This state can then be restored with `longjmp`. The two functions allow to jump to a previous position in the program without unwinding the stack step by step [JTC18], which again introduces a problem for CFI enforcement with shadow stacks.

- The **`exit`** function available in C entails a similar problem. It can be called from arbitrary positions in the program, terminates execution immediately and only calls functions registered with `atexit` before. This leaves CFI components, such as a shadow stack, in a potentially dirty state and might lead to problems for subsequent executions. The `_Exit` function poses a special form of this problem and leads to termination without even calling `atexit` functions [JTC18]. However, CFI-related components can be reset upon an `exit` because the program terminates anyways at this point.

- **Multithreading** needs to be considered on some platforms and requires the duplication of CFI-related components in the CPU or more advanced solutions. This can be problematic for memory-intense elements like shadow stacks [CCAI16].

- **CFG generation** is a very complex topic. The edges for function returns, direct calls and jumps can be easily deduced. For their indirect variants on the other hand it is very hard, and sometimes even not possible, to find out the set of possible branch targets. Section 2.3 described some basics of CFG generation.

- **Instrumentation of source code or programs** is required for all concepts presented in this chapter. This is the process of injecting (custom) instructions into the program in order to control CFI enforcement. Instrumentation is particularly challenging for shared libraries and programs, where no source code is available. Also issues like dynamic linking and separate compilation of the required Compilation Units (CUs) needs to be considered. Chapter 5 covers the process of instrumentation in detail.

- **Access to required run-time information**, e.g., pipeline internals or memory elements, might be limited and can also be problematic. Internal signals like the Program Counter (PC) might not be accessible and changes to the CPU core are not always possible.
- **Hardware constraints** limit the possibilities for additional elements for CFI (e.g., shadow stack, policy matrix or controllers), which have to be carefully dimensioned. Both the requirements for proper CFI as well as the available hardware resources establish lower, respectively upper limits. Storage elements can be implemented in multiple ways. While, for instance, using existing main memory implemented with Block RAMs (BRAMs) blocks for a shadow stack is very efficient in terms of hardware requirements, it introduces new challenges like access latency and security concerns, which have to be handled specifically. Flip Flops (FFs) on the other hand are very fast but expensive and particularly limited.

3.3 Hardware-Based CFI Implementations

This section summarizes a selection of concrete hardware-based CFI implementations of the concepts listed in Section 3.1 and also describes an already specified and released implementation by Intel. Most of the approaches listed here use multiple concepts in order to support both forward and backward edge protection, while others focus on either. We selected four academic implementation (FIXER [DBGJ19], HAFIX [DHP+15], HCFI [CCAI16] and HECFI [SAD+16]) and one industry approach (Intel CET [Gar20]) for a more detailed evaluation based on the mix of CFI concepts and implementation details they use. These schemes pose a good sample of different concepts.

However, there exist more hardware enforced CFI concepts than the subset covered in this section. Especially BR and cryptographic approaches, which were both briefly mentioned in Section 3.1.5, are highly active research topics. Given the title of this thesis, we limit our focus on fine-grained CFG-based CFI concepts here though and refer the reader to further literature on other approaches, e.g., the extensive survey of hardware-based CFI by de Clercq and Verbauwhede in [dCV17].

It is worth noting at this point that software-side forward edge CFI, as described in Section 2.4, is sometimes considered to be sufficiently efficient because function returns appear way more frequent than indirect jumps and calls [TRC+14]. A focus on backward edges therefore seems to be a reasonable restriction.

The implementations described here share a common underlying principle: All of them extend some CPU with a module for CFI enforcement, either by modifying the core itself or using some kind of co-processor, and introduce ISA extensions with custom instructions for annotating a program with CFG information.

While extending core internals offers a great deal of possibilities, its practicality might be limited because existing Intellectual Property (IP) soft-core CPUs are often used in real-world applications. Modifying these is complex, expensive or not possible at all in

many cases. Another approach for enforcing CFI in hardware is utilizing the core's debug interface. Not all required signals might be available for some interfaces though [dCV17].

A direct comparison between the implementations seems tempting at this point, but cannot be established in a meaningful way yet. The implementations are partly based on different platforms and the authors each use different benchmarks for evaluation. Chapter 4 presents implementations for the five concepts of this section on a common platform, namely a 32-bit RISC-V System-on-a-Chip (SoC). Based on those, meaningful comparisons in terms of performance, code size, hardware utilization and security are possible, which are eventually discussed in Chapter 6.

3.3.1 FIXER: Flow Integrity Extension for Embedded RISC-V

De et. al proposed **FIXER** in [DBGJ19], which protects both forward and backward edges by combining the well established concepts of a shadow stack (as described in Section 3.1.2) and a policy matrix (Section 3.1.4). They implemented their approach on RocketChip [AAB+16], a 64-bit RISC-V SoC platform, and benchmarked on a Xilinx Zynq Field Programmable Gate Array (FPGA). RocketChip offers the possibility to extend the processor's functionality without modifying the core itself by using a co-processor called Rocket Custom Coprocessor (RoCC) for custom instructions. All CFI elements, including the required memory modules, reside there.

For backward edge protection, the instructions shown in Listing 3.3 (lines 3-5) are inserted in front of every function call. First, the current PC is written to register `t0` and incremented in order to point to the return address of the call using standard RISC-V instructions. Then the RoCC custom instruction in line 5 pushes the content of register `t0` to the shadow stack.

In addition, the instructions shown in Listing 3.4 (lines 5-6) are inserted in front of every function return. The RoCC instruction in line 5 pops the topmost element off the shadow stack and writes it to register `t0`. The conditional branch in line 6 then compares `t0` to the return address stored in register `ra` and jumps to some error handling code in case a mismatch is detected.

```
1 <foo>:
2 ...
3 auipc       t0,0
4 add         t0,t0,14
5 CFI_CALL
6 call        bar
7 ...
```

Listing (3.3) FIXER: Direct function call [DBGJ19]

```
1 <bar>:
2 ...
3 ...
4 ...
5 CFI_RET
6 bne         t0,ra,_cfi_error
7 jr          ra
```

Listing (3.4) FIXER: Function return [DBGJ19]

Forward edge protection is performed by using a policy matrix, which is populated before executing the program and queried at every indirect function call. The matrix contains a

bit for each caller-callee pair indicating whether the respective call is allowed. To that end, a custom `CFI_FWD` instruction containing both the address of the call as well as the dereferenced function pointer is inserted right before an indirect call. The CFI module in the RoCC then checks and returns the corresponding bit in the policy matrix.

Although the selected CPU platform with the RoCC appears to offer a very convenient way of extending core functionality, it comes at significant performance costs. The CFI instrumentation for this approach shown in the above listings requires multiple instructions in each case. The overhead could be reduced to a single instruction in each case if access to the core itself is available.

3.3.2 HAFIX: Hardware-Assisted Flow Integrity Extension

Davi et al. presented a highly noticed approach called **HAFIX** [DHP+15], which uses the concept of *active-sets* (as described in Section 3.1.1). They focus on backward edge protection and argue that forward edge CFI is less challenging in terms of performance and can therefore be efficiently implemented in software. This is because function returns occur significantly more often than indirect calls and jumps.

The paper describes the implementation of two different concepts for backward edge CFI: A shadow stack approach implemented on a LEON3 microprocessor with the SPARC V8 instruction set and an active-set approach on some Intel Siskiyou Peak x86 platform. We focus on the latter in this summary because the first does not significantly differ from other shadow stack implementations.

In contrast to most other (hardware-based) CFI approaches, HAFIX does not required a pre-computed CFG. Each function is assigned a unique label, which can be as trivial as a strictly increasing counter. At the beginning of each function, a `CFIBR` instruction with that label is inserted at the very first position. The CFI module contains a memory element directly indexed by this label, which holds one bit for each possible function indicating whether the function corresponding to the label is currently active. An internal state machine make sure that each function call is immediately followed by such an instruction, otherwise a control flow violation is detected. Listing 3.5 and Listing 3.6 show the required additional instructions.

```
1 <foo>:
2 CFIBR    0x10
3 ...
4 call     bar
5 CFIRET   0x10
6 ...
7 CFIDEL   0x10
8 ret
```

Listing (3.5) HAFIX: Function call

```
1 <bar>:
2 CFIBR   0x14
3 ...
4 ...
5 ...
6 ...
7 CFIDEL   0x14
8 ret
```

Listing (3.6) HAFIX: Called function

Right before a function returns, it is flagged inactive with a `CFIDEL` instruction using the same label as the initial `CFIBR`. When a function returns, the backward edge has to target a `CFIRET` instruction. This is again enforced by the state machine internal to the CFI module. `CFIRET` checks if the function corresponding to its label is currently flagged active and raises an exception if the return targets an inactive call site.

As discussed in Section 3.2, recursion is challenging when it comes to CFI. In implementations with shadow stacks, recursion can overflow the stack and thereby break CFI checks. However, active-set approaches have a different problem: They flag some recursively called function inactive when the first return of that function occurs, e.g. with the `CFIDEL` used in HAFIX. When the next return for the same function happens, a CFI violation is wrongly detected because the associated flag was already reset.

Functions called recursively are instrumented differently in HAFIX for that very reason: Instead of using a `CFIBR` instructions at the function entry, a `CFIREC` containing the usual label is issued. The CFI module increases an internal counter associated to the label of this function on every recursive call, and decreases it on every return. The counter is implemented as a register for fast access. It is argued that using a separate instruction for recursive calls prevents many memory operations on the active-set, which is implemented using BRAMs, because only the counter register is accessed instead.

The implementation only supports non-nested recursive functions because it uses a single counter for keeping track of recursion depth. One of the benchmark applications used in our evaluation, namely `slre`[1], contains such a form of recursion where function calls are nested like `foo-bar-bar-foo-bar-bar-foo ...`, which is not compatible with HAFIX for the reason described above.

The active-set concept used here has been proven to be ineffective in some cases. Theodorides et al. showed that multiple different attacks on the control flow are sill possible despite having CFI backed with an active-set [TW17]. It is also possible to bypass `CFIDEL` instructions using stack unwinding, which eventually leads to all functions being flagged active [CCAI16].

3.3.3 HCFI: Hardware-enforced Control-Flow Integrity

Christoulakis et al. presented an approach called **HCFI** in [CCAI16]. They used the concept of labels (as described in Section 3.1.3) for protecting forward edges and a shadow stack (Section 3.1.2) for backward edges. To that end, they implemented their approach on a Leon3 32-bit SoC with the SparcV8 ISA and evaluated it on an FPGA.

Listing 3.7 and Listing 3.8 show example code for how caller and callee functions are instrumented for indirect function calls in pseudo assembly. Note that the platform used here supports a branch delay slot, which in many cases allows for a very efficient CFI implementation because the compiler is not always able to fill this slot with meaningful instructions and would only issue a `NOP` instead. Right after calling a function, a

[1] `https://github.com/embench/embench-iot/tree/master/src/slre`

SETPCLABEL instruction carrying a label is executed. This instruction not only sets the label used for CHECKLABEL at function entries, but also pushes the current PC to the shadow stack.

The delay slot is also used for backward edge protection, i.e., for the CHECKPC instruction at function returns, which pops off the shadow stack and compares the value with the return target from the actual stack.

```
1 <foo>:
2 ...
3 call        $s2
4 SETPCLABEL  0x13
5 ...
```

Listing (3.7) HCFI: Indirect function call

```
1 <bar>:
2 CHECKLABEL 0x13
3 ...
4 ret
5 CHECKPC
```

Listing (3.8) HCFI: Called function

Similarly to indirect function calls, all direct ones need to be instrumented too. Only now a SETPC instruction is used, which works the same way as SETPCLABEL but does not carry a label and also suppresses the CHECKLABEL check at the entry of callee. In addition, an internal FSM makes sure only valid sequences of CFI instructions are executed. For instance, SETPCLABEL must immediately be followed by CHECKLABEL. Mismatches cause exceptions in any case.

The authors also added support for `setjmp` / `longjmp` constructs and specific recursion handling to show that these known challenges, which we described in Section 3.2, can be overcome. Additional instructions announce that a `setjmp`, respectively a `longjmp`, is on the way. While the former stores the current shadow stack pointer into a separate register, the latter unwinds the shadow stack down to that position.

Recursive function calls are flagged on the shadow stack in order to prevent a stack overflow. When a SETPC instruction arrives and its PC matches the top of the shadow stack, the recursion flag for this position is set and the PC is not pushed to the shadow stack. Upon a CHECKPC instruction, the top of the shadow stack is compared again and in case its recursion flag is set, it is not popped from the stack. To detect the end of recursion, also the second address on the stack is compared if comparison with the topmost address resulted in a mismatch.

Consider the recursive factorial calculation shown in Listing 3.9 as a basic example. The stack never grows throughout the complete execution of the recursive function, which is the intended behavior and very effective. HCFI does not keep track of recursion depth though, which introduces a problem for certain kinds of applications. The problem is that the concept only allows for simple forms of recursion.

Some more complex recursive functions seem to break the concept for recursion handling suggested by Christoulakis and his colleagues. For instance, the recursive `towers` implementation from the RISC-V Benchmark suite[2] apparently cannot be handled by

[2] https://github.com/ucb-bar/riscv-benchmarks

```
1 int factorial(int n) {
2   if (n >= 1) {
3     return n * factorial(n-1);
4   } else {
5     return 1;
6   }
7 }
```

Listing 3.9: Recursive factorial function

HCFI and wrongly leads to an exception. Listing 3.10 shows the simplified C code of the affected function. The problem here is that the end of recursion cannot be detected properly because of the multiple nested recursive function calls within the `towers_solve_h` function, which all have different return addresses. Since there is no counter of recursion-depth associated with each item on the shadow stack, there is no way of knowing when a recursion is finished. The authors suggestion to also compare the second item on the stack does not solve this problem here.

```
1  void towers_solve_h( ... ) {
2    int val;
3    ...
4
5    if ( n == 1 ) {
6      ...
7      val = list_pop(...);
8      list_push(...);
9      ...
10   } else {
11     towers_solve_h( n-1, ... );
12     towers_solve_h( 1,   ... );
13     towers_solve_h( n-1, ... );
14   }
15 }
```

Listing 3.10: Simplified recursive `towers` function

3.3.4 Strategy Without Tactics - Policy Agnostic Hardware-Enhanced Control-Flow Integrity

Sullivan et al. suggested a very extensive approach in [SAD+16], which follows the same basic principle as HCFI [CCAI16] but also protects indirect jumps and uses the concept of *trampolines* for handling indirect calls of functions used by multiple callers. This allows for greater CFG precision. The authors also extended a LEON3 core and the SPARCV8 ISA with custom instructions for marking indirect calls and jumps, function entries, and the targets of indirect jumps and backward edges. Sullivan et al.'s concept is called *HECFI* in the remainder of this thesis for easier reference.

Both direct and indirect function calls are extended with a preceding `CFIBR` instruction containing a label, which is pushed onto a Label State Stack (LSS). The LSS is basically a shadow stack for labels. This approach is a major difference to other concepts using a shadow stack because most of them push full return addresses. The first instruction executed after a function return has to be a `CFIRET` containing the same label as the preceding `CFIBR` instruction of the corresponding function call. Everything else constitutes a CFI violation. Listing 3.11 and Listing 3.12 show exemplary functions.

```
1 <foo>:
2 ...
3 CFIBR     0x2
4 CFIPRC    0x3
5 CALL      $s2
6 CFIRET    0x2
7 ...
```

Listing (3.11) HECFI: Indirect call

```
1 <bar>:
2 CFICHK    0x3
3 ...
4 ...
5 ...
6 ...
7 RET
```

Listing (3.12) HECFI: Callee function

For securing indirect calls, additional `CFIPRC` instructions containing a different label are also inserted right before the call instruction. This label must match the label of a `CFICHK` instruction right at the entry of the callee. The check performed there is only active immediately after `CFIPRC` instructions, but suppressed for direct function calls.

Sullivan et al. argue that using two instructions with different labels, i.e. one each for the forward- and backward edge policy, benefits security. This is necessary because the callees might be called from multiple sites, which would otherwise need to use the same label for backward edge protection. However, similar concepts (such as HCFI, see Section 3.3.3) push the current PC to the shadow stack instead of labels. This enables them use the same instruction for both announcing an indirect call and push the PC to the stack, thereby saving one instruction in comparison to the concept by Sullivan et al.

Indirect jumps use the very same concept with the only difference that a `CFIPRJ` instruction is used instead of `CFIPRC` and no stack is involved because jumps don't imply a backward edge. This is important because the implementation uses an internal state machine to ensure the correct order of CFI instructions.

When multiple indirect calls legitimately target the same function, all `CFIPRC` - `CFICHK` pairs would have to use the same label. This would thereby introduce a coarse-grained CFG, potentially enabling exploits on the control flow. Sullivan et al. proposed to use trampolines to solve this problem, such as briefly introduced in Section 3.1.3.

The authors only describe a very basic use case though, namely one where an indirect call only has one potential target. The more general case for indirect calls is having a set of possible targets. In order to support this case, trampolines need to be organized as a jump table, i.e. by comparing the actual call target with a list of allowed ones and redirect the jump accordingly. A failed lookup in this table leads to a CFI exception.

3.3.5 Intel Control-Flow Enforcement Technology (CET)

Intel published the specification of an upcoming commercial security feature called CET for its x86 CPUs in 2016 [Int19, Gar20]. This represented a milestone because most other hardware-enforced CFI approaches never left the academic stage. As depicted in Figure 3.12, Intel's concept consists of two components directly integrated into the core.

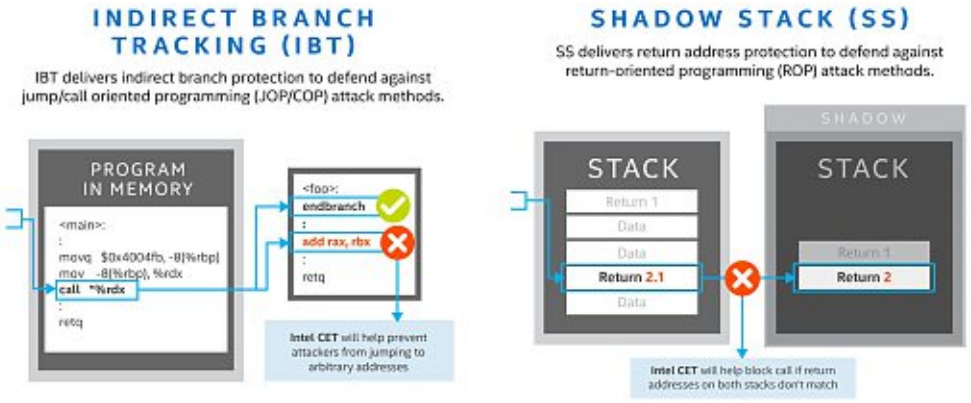


Figure 3.12: Intel Control-Flow Enforcement Technology [Gar20]

An ordinary shadow stack is used for backward edge protection. Its basic workings are very similar to the shadow stack implementations of the academic CFI approaches presented above. There is one significant advantage though: Intel uses the common ISA instructions for function calls and returns for interfacing with the shadow stack and can thereby save a lot of otherwise additionally required instructions.

For forward edges, Intel specified a concept called Indirect Branch Tracking (IBT), which makes sure that all sorts of indirect control flow transfers are only allowed to jump to some legit target address. This is very similar to Microsoft's CFG, which was briefly discussed in Section 2.4 [msc18, grs16], and the general concept of BR, as described in Section 3.1.5.

There are no labels involved, meaning IBT only uses a single equivalency class (similarly to using a single label, as presented in [ABEL09]) for *all* indirect branch targets. To that end, the entries of functions potentially called in an indirect way as well as indirect jump targets are instrumented with an `ENDBRANCH` instruction. This new instruction is normally interpreted as a `NOP` on the x86 ISA, ensuring backward-compatibility for CET-enabled applications on older machines. Only new Intel CPUs interpret `ENDBRANCH` as the only legit target of indirect control flow transfers and thereby ensure CFI with an internal FSM [Int19].

The *Tiger Lake* mobile CPU family released in late 2020 is the first to actually implement CET in hardware after the announcement in 2016 [Gar20].

Adding the ENDBRANCH instruction to all possible indirect branch targets remains a task for compiler makers. Naive implementations simply add ENDBRANCH to *all* function entries and *all* jump targets, thereby sacrificing performance. More intelligent solution would involve complex control flow analysis during compilation. We compiled simple test programs with ICC 19.01, GCC 10.2 and Clang 11.0.0 having full CET enabled. Disassembled binaries hint that ENDBRANCH is added to all functions entries and all labels in the user code, regardless whether they are used as targets of indirect control flow transfers. This is because the compilers cannot be sure whether some function is indirectly called across different CUs. Only when functions are declared `static` or LTO is enabled for compilation, the compilers could derive which functions are actually the targets of indirect calls in our tests. In these cases, instrumentation with ENDBRANCH is done in a more fine-grained way. No hardware with CET was available for testing yet at the time of writing this thesis.

CHAPTER 4

Implementation

The hardware-enforced CFI approaches previously described in this thesis are not fit for a direct comparison because their implementation environments and evaluations differ too much. We therefore implemented a selection of these schemes on a common platform, namely a 32-bit RISC-V SoC, to enable a better comparison. In order to cover and examine different CFG-based concepts, this set consists of FIXER [DBGJ19], HAFIX [DHP+15], HCFI [CCAI16], HECFI [SAD+16], which are all academic proposals, as well as Intel's commercially available Control-Flow Enforcement Technology (CET).

This chapter first gives an overview of the software- and hardware platform used for our implementations in Section 4.1. Section 4.2 describes how the ISA extension required for custom CFI instructions was implemented. Then some remarks on the original concepts are listed and necessary changes for an implementation on our platform are discussed in Section 4.3. Section 4.4 defines and explains a common set of parameters, which is needed for a meaningful comparison.

Eventually, a novel scheme for hardware-enforced CFI called EXtensive CFI Enforcement Concept (EXCEC) is presented in Section 4.5, which is based on what we consider the best concepts and ideas found in the existing approaches from Chapter 3 and is enriched with some custom enhancements.

4.1 Development Platform & Environment

The PULP platform[1] and its PULPissimo microcontroller architecture[2] are used as a base for all implementations presented in this thesis. The platform provides open-source hardware designs as well as the required software-side components.

[1] https://pulp-platform.org
[2] https://github.com/pulp-platform/pulpissimo

PULPissimo is an open-source SoC platform developed by ETH Zürich and the University of Bologna. Its implementation in the Verilog hardware description language (HDL) is fully synthesizable and operates with a single core [pul]. PULPissimo allows developers to select either of the CV32E40P[3] (formerly known as RI5CY) or Ibex[4] (previously Zero-riscy) CPU cores for the SoC. We decided to use CV32E40P, which is a 32-bit in-order core with four pipeline stages and implements the RISC-V base ISA, including the extensions C (compressed), M (multiplication) and optionally F (floating point) [cv3].

While the platform can be fully simulated, all evaluations were performed on a Xilinx ZedBoard[5] and simulation was mainly used for testing purposes. The Zynq Z-7020 SoC on the board also contains an ARM Cortex-A9 with considerable hardware resources, but only its programmable logic (PL) part was used, i.e., the board's FPGA, which offers 53,200 Lookup Tables (LUTs) and 106,400 Flip Flops (FFs). The PULPissimo platform operates with 16 MHz and has a total of 512 KiB of memory on the ZedBoard.

CFI-related functionality was implemented for all schemes directly within this core, e.g., by adding CFI modules to the *execute*-stage of the pipeline, because PULPissimo's full source code is available. All of the five before mentioned CFI approaches and a novel scheme proposed by us (later described in Section 4.5) were implemented in a modular way so that they can be exchanged in the SoC with minimal effort.

Other changes to CV32E40P only affect the *instruction decoder*, where support for various custom instructions was added, and the *pipeline controller* for exception handling. The latter receives exceptions from the CFI modules and reacts upon them.

On software-side, the PULP platform provides a modified RISC-V toolchain[6] with many PULP-specifics, including forks of RISC-V GCC and the respective GNU C Library. Another important component is the PULP software development kit (SDK)[7], which contains platform-specific configurations and function implementations. Both the PULP RISC-V GNU toolchain and the PULP SDK are required for cross-compilation of applications for the RISC-V PULPissimo platform and execution on the FPGA.

The PULP platform also comes with a custom RISC-V ISA extension and adds, for example, hardware loops. Applications can be compiled with and without this extension, which affects runtime in some cases. Many programs show a better performance when being compiled *with* PULP extensions.

At the time of writing this thesis, the PULP SDK only supported compilation for programs written in C, but not C++. However, all of the applications used for evaluation and testing are written in C anyways. The official set of PULP examples[8] also only contains C programs. In addition, usage of C++ concepts like exceptions and inheritance

[3] https://github.com/openhwgroup/cv32e40p
[4] https://github.com/lowRISC/ibex
[5] http://zedboard.org/product/zedboard
[6] https://github.com/pulp-platform/pulp-riscv-gnu-toolchain
[7] https://github.com/pulp-platform/pulp-sdk
[8] https://github.com/pulp-platform/pulp-rt-examples

results in larger binaries. This can become a problem on platforms with limited hardware resources. As already mentioned above, our development platform only comes with a total of 512 KiB memory, which constraints the size of executable programs.

4.2 Instruction Set Architecture (ISA) Extension

In general, most hardware-based CFI proposals require custom instructions, i.e., extensions to the respective ISA. Such instructions must be added to the toolchain on software side, but also hardware support is needed.

The RISC-V base instruction set supports 32-bit instructions with the six formats shown in Figure 4.1. This simplifies decoding as compared to complex ISAs like x86 [AW17].

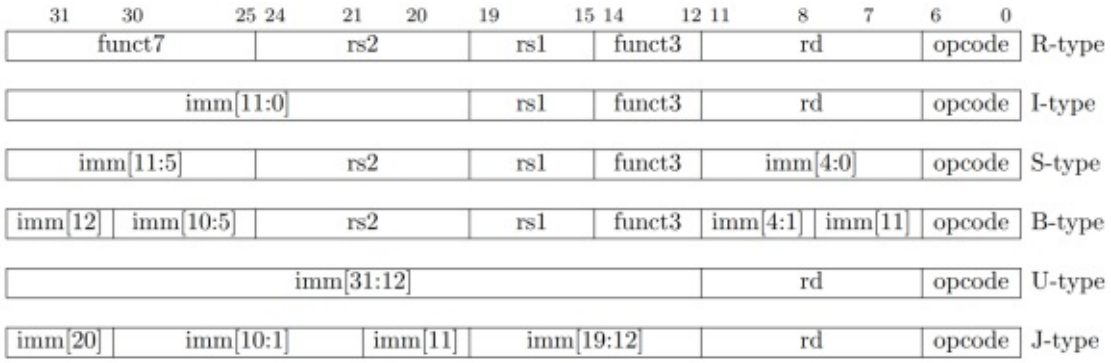


Figure 4.1: RISC-V base instruction formats [AW17]

The RISC-V GNU toolchain used in our setup can be easily extended with custom instructions. To that end, the format of new instructions needs to be defined as exemplary shown in Listing 4.1 for two instructions similar to the ones used for HCFI in [CCAI16].

```
1 SETPC          31..20=0 19..15=0 14..12=0 11..7=0x00 6..2=0x1D 1..0=3
2 SETPCLABEL     imm20                      11..7=0x01 6..2=0x1D 1..0=3
```

Listing 4.1: Examples for custom instruction format

Here a still available `opcode` (bits [6:0], see Figure 4.1) is used to identify custom CFI instructions in general and the function code in bits `[11:7]` is set unique for each instruction. In accordance to Figure 4.1, these bits are usually used for the `rd` register or for immediate values in other instructions. We preferred to use an instruction format able to transport a 20-bit immediate value though, so the respective bits are used as a function code instead. Note that the `SETPCLABEL` instruction uses the `imm20` field for the label, while `SETPC` does not use a parameter. These instructions can then be used just as any other assembly instruction, as for instance depicted in Listing 4.2.

```
...
SETPC
call <foo>
...
SETPCLABEL  0x42
call s3
...
```

Listing 4.2: Examples for using custom instructions

However, these definitions only allow for injecting custom assembly instructions and are not sufficient to implement an extension to the programming language. Section 5.3 later describes how instructions are internally represented in GCC. In simple words, when adding instructions only like described here, the compiler knows about the instruction's syntax, but not about its semantics. This is sufficient for our application.

Changes to the toolchain are only necessary for compiler-based instrumentation approaches. If no tool support is required and instructions are only added after compiling and linking, i.e., in some binary post-processing step, it is not necessarily needed to modify the toolchain at all because only the hardware is affected.

On hardware-side, the core's decoder must be extended in order to support custom instructions. This component is usually implemented as some sort of switch, which first identifies the instruction format (as shown in Figure 4.1) and den extracts the distinct fields based the that format.

4.3 Implementation Details

Key components of this thesis are implementations of existing hardware-based CFI concepts on a common RISC-V platform, which were presented by their authors for different ISAs. We picked FIXER [DBGJ19], HAFIX [DHP+15], HCFI [CCAI16], HECFI [SAD+16] and Intel's CET [Int19] and implemented the schemes as close to their original proposals as possible.

However, some implementation details are not disclosed by the authors and thereby leave room for interpretation. Such ambiguities are noted here wherever necessary. In addition, porting to our RISC-V platform required certain changes due to differences resulting from the used hardware platforms and ISAs, which are also documented in this section.

All CFI schemes were implemented as separate modules, which are simply attached to the core's pipeline. The pipeline itself only requires minor changes in our setup: Merely the additional custom instructions need to be handled in the decoder and are then forwarded to the respective CFI module. Furthermore, all exception signals for CFI violations are wired to the pipeline controller. A dedicated exception handler for such errors logs the exception cause, resets the core and terminates program execution.

Performance impacts are considered a problem of shadow stacks, which are only mapped to the main memory [CCAI16, TW17]. All memory elements were implemented with

FFs instead of BRAM elements for this reason. Implementations based on FFs hardly introduce any performance overhead, but FFs are a rather limited and expensive resource.

The functional scope of an implementation not only impacts parameters like security and hardware utilization overhead, but also strongly affects the set of supported applications. For example, CFI concepts without specific recursion handling can exceed their shadow stack capacity more easily, but recursion handling is hard for complex cases. Other challenging language concepts like C's `setjmp` or `exit` functions and compiler optimizations like *call-tail elimination* can break CFI. Such problems were described in Section 3.2.

Most of the previously presented CFI approaches depend on custom instructions, partly without any parameters. This raises the questions as to whether such instructions are actually needed. Consider, for example, the `SETPC` instruction used in HCFI. Its purpose is to push the return address of its associated function call to the internal shadow stack and to temporarily disable forward edge protection for indirect branches. The same task can be achieved by extending semantics of actual `call` instructions, which are already part of the respective ISA, if access to internal pipeline signals is possible.

We added custom instructions with general CFI management commands to all of the implemented schemes. `CFI_ENABLE` and `CFI_DISABLE` are used to turn CFI enforcement *on* before entering `main`, respectively *off* after returning from `main`. This way, startup and exit code is excluded from CFI enforcement because active protection is not required there. An additional `CFI_RESET` instruction is used to reset the internal state of the CFI modules before a program enters `main`. This is a precautionary measure for the case where the state of the CFI module is left dirty because of improper program termination. `CFI_RESET` and `CFI_DISABLE` are also explicitly used for the same reason when `exit` calls occur.

The following paragraphs list some findings, required modifications and ideas for improvements in the various CFI implementations.

FIXER

The concept for FIXER was originally proposed as a coprocessor to a 64-bit RISC-V SoC [DBGJ19]. Our development platform (as described in Section 4.1) is a 32-bit RISC-V SoC though, where the CFI module can be integrated directly into the core.

A main consequence of implementing the CFI module as a coprocessor is that multiple instructions are needed for getting the current PC, which is required for backward edge CFI. This is because the coprocessor does not have access to the core's internal signals. Figure 4.2 shows how a direct function call and its associated return are instrumented in the original FIXER concept. Other approaches are able to achieve the same functionality with only one additional instruction each instead of two or three.

In order to enable a fair comparison to other CFI concepts, which are integrated directly into the pipeline, we improved this behavior in our implementation so that also only one

```
1 <foo>:
2 ...
3 auipc       t0,0
4 add         t0,t0,14
5 CFI_CALL
6 call        bar
7 ...
```

Listing (4.3) Caller instrumentation

```
1 <bar>:
2 ...
3 ...
4 ...
5 CFI_RET
6 bne         t0,ra,_cfi_error
7 jr          ra
```

Listing (4.4) Callee instrumentation

Figure 4.2: Instrumented code for direct function call and return in FIXER [DBGJ19]

instruction is needed for each action. Naturally, the number of instructions required for the original FIXER has a considerable negative effect on performance and code size.

The paper is rather vague on how the policy matrix for forward edge protection is implemented. Initially filling the matrix requires knowledge about both the addresses of indirectly called functions as well as the addresses of the indirect function calls. The latter cannot be easily determined across multiple CUs at compile time without significant changes to the source code. We therefore decided to not use the caller-addresses to index the matrix, but simple labels which act as index in their place. This tweak only helps when filling the matrix and looking up flags therein, but does not change the approach in terms of instruction count or performance overhead.

However, it is still necessary to transform 32-bit callee addresses to matrix indices because the latter have a significantly smaller value range, so some sort of address space translation is required. The authors of FIXER briefly mention *address decoders* for this job. Our implementation uses an additional array for decoding, which holds some unique part of indirectly called addresses. When looking up a flag in the matrix, the CFI module queries this array in order to get an index for matrix access.

This is expensive in terms of critical path length and hardware utilization. We note that this approach for address decoding does not scale well and offers room for improvement. For instance, a pipelined implementation of the lookup could solve problems with the length of the critical path. Many other concepts, such as set-associative memories like they are used in caches, are not applicable here though.

HAFIX

The proposal for HAFIX suggests to use either a shadow stack or an active-set for backward edge CFI [DHP+15]. Only the latter approach is considered in this thesis since shadow stacks already occur in almost all other hardware-based CFI proposals.

Our implementation of the active-set is as described in the original paper, with the only difference that our set is smaller because less memory is available on our development

platform and also typical programs contain less functions in the embedded world. The selected values for such parameters are explained in Section 4.4.

The approach for recursion handling in HAFIX does not support nested recursion, as explicitly stated by the authors. The problem is that only one counter for recursion depth is used, which cannot be shared for nested recursive calls. Our implementation of HAFIX does not try to fix this shortcoming. The `slre` benchmark, which is one of the applications used for evaluation (see Section 6.2), can not be executed with HAFIX for that reason.

HCFI

The concept for HCFI was implemented for the SPARC ISA [CCAI16], which allowed the authors to make use of branch delay slots. Our RISC-V platform does not support this concept, so it was necessary to change the position and accompanying semantics of some instructions in our HCFI implementation. This conceptual difference might have a small negative performance impact. Other changes were not necessary for this approach.

The recursion concept of HCFI fails in our implementation and triggers an CFI exception in the `towers` application from our set of benchmarks. The reason for that was explained in Section 3.3.3.

HECFI

The proposal for HECFI was also implemented for SPARC, but does not use branch delay slots [SAD+16]. Our implementation largely follows the approach from the paper, only *trampolines* require additional explanation here because only a very basic case with just one potential call target is described by the authors. We implemented trampolines as jump tables containing all valid jump targets and added the respective code at the end of the function where the corresponding indirect call is located. During execution, the actual target address is compared with the table entries and only transfers to valid targets are allowed.

Intel Control-Flow Enforcement Technology (CET)

Intel CET is the only non-academic approach considered in this thesis. We focus on the basics in our implementation because the original concept targets commercial full-scale CPUs. Backward edge protection with an internal shadow stack follows established concepts, but forward edge CFI involves some details to consider.

Protection of indirect function calls and jumps only requires the custom `ENDBRANCH` instruction (without any label or index as parameter) at callee side. CFI for indirect forward edges is triggered implicitly by the respective instructions and does not require any instrumentation on side of the caller. This entails a very low performance overhead and the absence of a custom instruction at caller side also means that *every* indirect forward edge is automatically CFI enforced. Our CET module expects any `JALR` and

JR instruction on forward edges to be immediately followed by ENDBRANCH. However, this is a problem for portions of code, which cannot be instrumented with ENDBRANCH. The consequence is that it was necessary to disable CFI enforcement for programs with indirect control flow transfers in affected functions.

Basic tests in a setup with direct and indirect function calls showed that Intel's ICC 19.0.1, GCC 10.2 and CLANG 11.0.0 add ENDBRANCH to the entry of *every* function, including `main`. Only when functions are declared `static` or LTO is enabled, the compilers inject ENDBRANCH in a more fine-grained way. It remains unclear to us whether this is the common compiler behaviour or more advanced concepts are used for detecting the targets of indirect control flow transfers. We slightly changed this concept in our implementation to enable a fair comparison with other CFI schemes and only instrument entries of functions, which actually are indirectly called.

4.4 Common Implementation Parameters

In order to achieve a good comparability, a common set of parameters is used for all hardware-based CFI implementations. These depend on actual requirements of the applications to be executed but are also limited by the available hardware resources of our development platform (see Section 4.1). The values assigned to each of the parameters are based on the set of benchmark applications used in this thesis (later described in Section 6.2).

Parameter	Value
ADDRESS_WIDTH	32
SHADOW_STACK_SIZE	128
INDIRECT_CALLS	64
INDIRECT_JUMPS	64
INDIRECTLY_CALLED	64
SETJMP_CALLS	8
NUM_FUNCTIONS	1024
RECURSION_DEPTH	128

Table 4.1: Common implementation parameters

The SHADOW_STACK_SIZE is required by all implementations using a shadow stack. The size allows for 128 nested function calls, which is sufficient for our set of benchmark applications. In most cases, 32-bit instruction addresses (defined by ADDRESS_WIDTH) are stored on the stack. As later described in Section 4.5, we challenge the assumption that full addresses need to be stored and present an alternative approach where only 18 bits are pushed.

HECFI does not store return addresses on its shadow stack, but labels unique to the caller functions. The NUM_FUNCTIONS parameter is used to derive the required stack width

for this case, which resolves to `log2(NUM_FUNCTIONS) = 10`. The `NUM_FUNCTIONS` parameter is also used for the size of the active-set used in HAFIX. Only flags are stored in this memory element, so no width parameter is required.

The numbers of indirect function calls and of indirectly called functions are especially relevant for FIXER. `INDIRECT_CALLS` and `INDIRECTLY_CALLED` define the dimensions of the policy matrix in this approach. Doubling these two parameters quadruples the required memory for FIXER's matrix. Considering available hardware resources, both are set to 64. This is a rather optimistic value and should be increased if hardware resources allow. These two values also define the width of labels used for forward edge protection in other CFI concepts. Label-based approaches, such as used in HCFI, therefore only need to handle 6-bit labels (`log2(INDIRECT_CALLS) = 6`). `INDIRECT_JUMPS` follows the same principle and describes the number of supported indirect jumps.

Given the rare occurrences of `setjmp` calls, `SETJMP_CALLS` is set to 8. This allows for 8 unique appearances of `setjmp`.

The `RECURSION_DEPTH` parameter defines the maximal supported recursion depth, which is required in approaches using recursion counters. In the context of this work it makes no sense to use a value larger than `SHADOW_STACK_SIZE` because the ultimate goal is to achieve comparability and some other CFI schemes have no dedicated recursion handling but simply push all recursive calls to their shadow stack.

We note that this selection of values for common parameters is not universally applicable but only reflects the requirements for our set of benchmark applications and the available resources in our hardware setup.

4.5 EXtensive CFI Enforcement Concept (EXCEC)

When analyzing the existing hardware-based CFI schemes previously described in Section 3.3, one comes across many promising concepts and ideas. Unfortunately, those are distributed across multiple approaches. This section presents a novel hardware-based CFI scheme called EXCEC. It is based on existing approaches, combines their best components and adds some enhancements.

EXCEC represents an improvement in terms of at least one of the aspects security, performance- or code size overhead in comparison to each of the existing implementations while keeping hardware utilization at a similar level. Such comparisons are extensively covered in Chapter 6.

EXCEC is designed for CFI enforcement on forward and backward edges. As opposed to most other CFI schemes, also indirect jumps are protected. In addition, support for `setjmp` calls and dedicated recursion handling is added and interrupts as well as CFI enforcement in Interrupt Service Routines (ISRs) are considered.

CFI Enforcement on Backward Edges

We decided for a shadow stack for the protection of function returns because it was established that other concepts like active-sets do not provide sufficient protection for function returns [CCAI16, TW17]. The required memory for the stack resides directly in the CFI module and is implemented with FFs instead of BRAM elements. This way, the memory is both secure from outside modification and also does not introduce any performance impact because registers can be accessed within one clock cycle.

There are some important details involved when using shadow stacks though. First, the position of the respective CFI instructions has great impact on checking the integrity of backward edges. Some approaches perform the check right when a function returns, i.e., by the function's second to last instruction before the actual return or in any branch delay slot of the return. Others do so at the target of the return and additionally make sure that the backward edge is immediately followed by such a check using some sort of FSM. We consider the first concept, i.e. checking the return address before executing the return instruction, favorable because it is less complex.

As a second issue the question arises as to which values should actually be pushed to the shadow stack. For instance, HECFI uses simple labels while HCFI pushes full 32-bit addresses to the stack. An enhanced version of the latter approach is used in EXCEC because it reduces complexity when instrumenting code with the underlying CFG since no labels are to be managed. Custom instructions are not used because it is possible to utilize the existing `CALL`, respectively `RET`, instructions of the RISC-V ISA for pushing to and popping from the stack in our setup.

As another tweak for the shadow stack only the bits [18:1] of 32-bit addresses are stored, as depicted in Figure 4.3, which cuts memory demand almost in half. The base address of PULPissimo's 512 KiB L2 memory is `0x1c000000` for applications built with the PULP SDK. This means that the 13 most significant bits (MSBs) of all instruction addresses have a constant value, which makes storing and comparing them meaningless. In addition, instruction addresses are always even and instructions have a fixed length of 4 bytes in the base RISC-V ISA [PW17], so the two least significant bits (LSBs) carry no information. However, PULPissimo supports the RISC-V extension for compressed (i.e., 16-bit) instructions, so only the LSB can be ignored. These tweaks are very platform-dependent though and might not be applicable for others.

Figure 4.3: Part of address used for shadow stack in EXCEC

CFI Enforcement on Forward Edges

While a policy matrix for forward edge protection, such as used in FIXER, is very intuitive and offers security at an optimal granularity, its memory demands outweigh the benefits of the concept. We therefore opted for using labels in a way similar to HCFI instead of a policy matrix. The labels are compared between caller and callee upon every indirect function call and indirect jump.

In addition, *Trampolines* are used in order to achieve fine-grained CFI for functions, which are indirectly called from more than one call site. This concept was introduced in [SAD+16]. The workings of trampolines in EXCEC are shown for an exemplary indirect function call in Figure 4.4. Trampoline code is added to the very end of the function, where the trampoline is used. Normal control flow never reaches this code, but the indirect function call redirects control flow there. The custom CFI instructions used in Figure 4.4 are explained in detail in the next section.

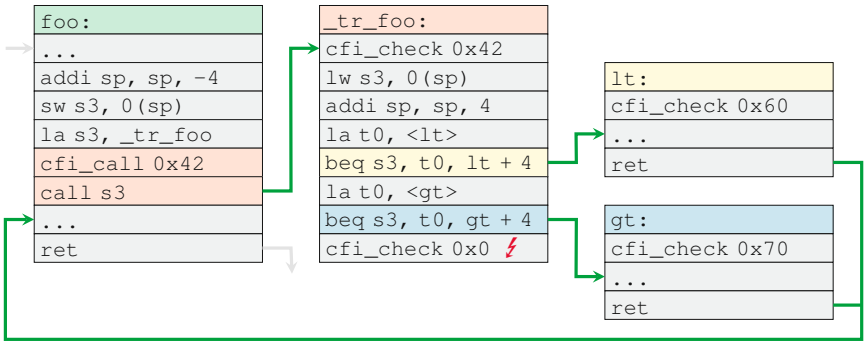


Figure 4.4: Workings of a trampoline for indirect function calls in EXCEC

When an indirect function call occurs, its target address is first saved to the normal stack. This is depicted by the first two instructions on the left side in Figure 4.4. Then the address (register `s3` in this example) is overwritten with the trampoline's address so that the indirect call transfers control to trampoline code instead of its actual target. A `CFI_CALL` instruction carrying a unique label is added right before the indirect control flow transfer. The call itself is then redirected to a trampoline, which first checks the label to ensures that the trampoline is unique for each call and cannot be reused from others. Afterwards the trampoline retrieves to original call target from the stack again and compares it to the addresses in the jump table (`lt` and `gt` in the example in Figure 4.4). In case the 12-bit offsets encoded in the table's `beq` instructions are not sufficient, direct jumps, which allow larger offsets, could be used in combination with conditional branches instead. Only if a matching table entry is found, control flow is directed to the respective function with a direct branch, bypassing the CFI check at its entry. This way, the indirectly called functions (e.g., `lt` and `gt`) can still be invoked from other callers without using the same label for all of their call sites. The `CFI_CHECK 0x0` instruction at the end of the trampoline explicitly triggers an exception in case the table contains no matching entry.

ISA Extension

The RISC-V toolchain offers to add custom instructions as an easy way to extend the base instruction set. This is done by adding the 32-bit custom instructions for CFI enforcement shown in Table 4.2.

Instruction	instr[31:12]	instr[11:7]	instr[6:0]
`CFI_CALL` *label*	imm20	0x11	0x77
`CFI_JUMP` *label*	imm20	0x12	0x77
`CFI_CHECK` *label*	imm20	0x13	0x77
`CFI_SETJMP` *index*	imm20	0x14	0x77
`CFI_LONGJUMP`	0x0	0x15	0x77

Table 4.2: CFI instructions for EXCEC

Custom instructions for announcing function calls, returns and jumps, such as the `SETPC` and `CHECKPC` instructions of HCFI, are not needed because full access to pipeline internals is available and already existing signals for the various control transfer types can be used. It is possible to directly use the branch signals from the core's decoder as commands for the CFI module and thereby significantly decrease both performance and code size overhead. The `imm20` field of RISC-V instructions is used for `label` and `index` values. CFI semantics of the changed RISC-V `JAL`, `JALR` and `RET` instructions, as well as of the instructions listed above in Table 4.2, are defined as follows:

- **`JAL`** and **`JALR`** are the existing instructions for direct and indirect function calls in the RISC-V ISA. When CFI enforcement is enabled, they (also) announce the call to the CFI module, which pushes the address of the subsequent instruction to the shadow stack for backward edge protection.
- **`RET`** is the existing RISC-V instruction for function returns. If CFI enforcement is enabled, this instruction additionally pops the topmost value off the shadow stack and checks whether it matches the address used for the function return. In case a mismatch is detected here, the CFI module issues an exception.
- **`CFI_CALL label`** and **`CFI_JUMP label`** announce an indirect function call, respectively an indirect jump, for safeguarding of forward edges. To that end, the *`label`* encoded as `imm20` in the instruction is stored within the CFI module. These instructions must be followed by a `CFI_CHECK label` with the same label.
- **`CFI_CHECK label`** is placed at the very first position of functions, which are called indirectly, and also at the target of indirect jumps. This instruction checks whether the `label` encoded within a preceding `CFI_CALL` or `CFI_JUMP` instruction matches its own `label`. In case a mismatch is detected here, the CFI module issues an exception. The instruction is usually suppressed and only enabled by

preceding CFI_CALL and CFI_JUMP instructions because forward edge protection is not required for direct calls and jumps. However, any CFI_CHECK with label 0x0 is considered a CFI violation. This allows for triggering explicit exceptions.

- **CFI_SETJMP index** is placed immediately after a setjmp call. Depending on whether there was a preceding CFI_LONGJMP, this instruction either stores the current shadow stack pointer in EXCEC's setjump memory, whereby index is used as an index to this memory in order to support multiple different setjmp calls, or restores this very pointer to a previously stored value.
- **CFI_LONGJMP** sets an internal flag such that a succeeding CFI_SETJMP call unwinds the shadow stack to a previously stored position.

Although the imm20 field of CFI_CALL, CFI_JUMP, CFI_CHECK and CFI_SETJMP instructions allows labels with up to 20 bits, smaller labels are used internally in order to save memory. The selected values for such parameters are explained in Section 4.4.

The correct sequence of CFI instructions, which is depicted in Figure 4.5, is enforced with an FSM in the CFI module. CFI_CALL and CFI_JUMP must be immediately followed by indirect branch instructions and CFI_CHECK. The CFI_CHECK itself can appear anywhere, but is suppressed unless it is enabled by a preceding CFI_CALL or CFI_JUMP or carries the label 0x0. This can be used to explicitly trigger exceptions when reaching invalid positions like the end of *trampolines*. Moreover, the RET instruction, which pops the topmost address from the shadow stack, is only valid if the last JAL instruction had pushed the very same value.

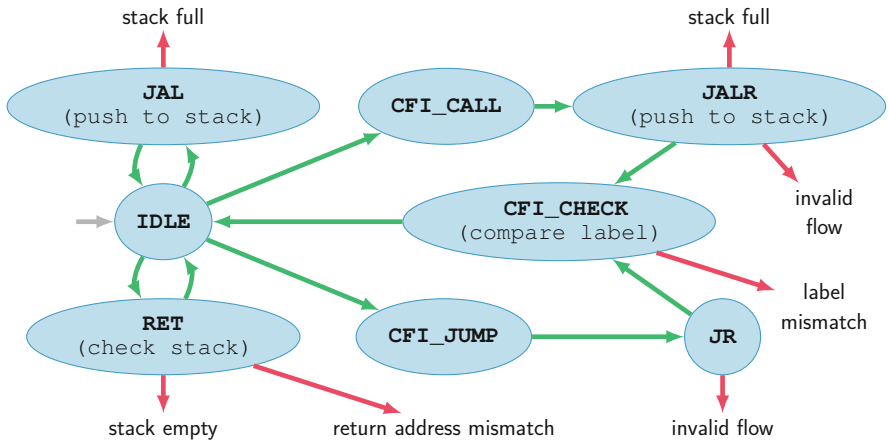


Figure 4.5: Valid (CFI) instruction sequences in EXCEC

Instrumentation makes sure that CFI instructions only appear at valid positions: CFI_CALL and CFI_JUMP, for example, only occur in in front of JALR and JR instructions. Other sequences, that don't comply with what is shown in Figure 4.5, are considered to be CFI violations and implicitly trigger exceptions.

Architecture & Interface

Figure 4.6 shows how EXCEC is included into the CV32E40P core's pipeline. All of the CFI enforcement functionality is packed into a separate module. The only modifications to the original core itself are extensions to the *Decoder* in order to support the additional CFI instructions and processing of CFI exceptions in the pipeline *Controller*, which acts upon CFI violations and transfers control to an exception handler.

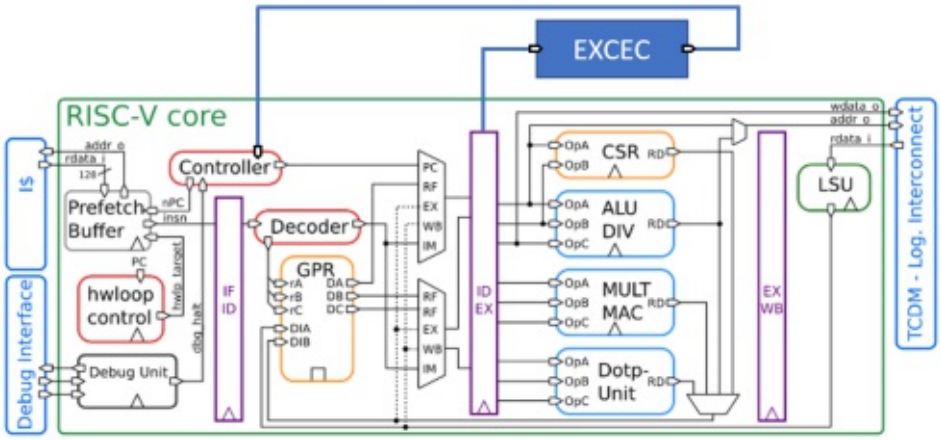


Figure 4.6: CV32E40P pipeline with EXCEC (modified from [pul])

An important feature of any CFI module is separation and safeguarding of all CFI related registers and memory elements in a way that instructions unrelated to CFI cannot modify their state. Only memory elements implemented with FFs internal to the EXCEC module are used instead of main memory. The parameters defined in Section 4.4 are used for dimensioning theses elements.

Table 4.3 describes the interface of our EXCEC implementation. The instructions for forward and backward edge protection as well as for the support of `setjmp` calls were described above. Signals with identical names correspond to these instructions. The `cfi_enable_i`, `cfi_disable_i` and `cfi_reset_i` signals are used for enabling CFI enforcement before entering `main`, respectively disabling CFI after returning from main and for resetting all registers in the EXCEC module. This way, platform setup code can be excluded and program execution always starts with a clean CFI state.

The outbound exception signals in Table 4.3 are routed to PULPissimo's pipeline controller and inform about the cause of some CFI violation. The `invalid_branch_o` signal is asserted when a `CFI_CHECK` instruction results in a label mismatch. This is the case when the label used with `CFI_CHECK` is different from the label in a preceding `CFI_CALL` or `CFI_JUMP` instruction. Deviations from the expected sequence of CFI instructions, i.e., a difference to the flow shown in Figure 4.5, are indicated by the `invalid_flow_o` signal. An example thereof is that indirect function calls, as announced with `CFI_CALL`, must be followed by `JALR` and `CFI_CHECK`. The `invalid_ret_o` signal shows whether

Signal	Dir.	Type	Description
clk	in	logic	clock signal
rst_n	in	logic	reset signal
jump_done_i	in	logic	for stalling
pc_if_i	in	logic [31:0]	PC in fetch stage
ra_i	in	logic [31:0]	return address for current call
jal_i	in	logic	current instruction is a JAL
ret_i	in	logic	current instruction is a RET
cfi_call_i	in	logic	upcoming instruction is a JALR
cfi_jump_i	in	logic	upcoming instruction is a JR
cfi_check_i	in	logic	check for indirect branches
cfi_sj_i	in	logic	setjmp call upcoming
cfi_lj_i	in	logic	longjmp call upcoming
cfi_label_i	in	logic [19:0]	label for forward edge CFI
cfi_index_i	in	logic [19:0]	index for setjmp
cfi_enable_i	in	logic	enable CFI
cfi_disable_i	in	logic	disable CFI
cfi_reset_i	in	logic	reset CFI
invalid_branch_o	out	logic	exception for label mismatch
invalid_flow_o	out	logic	exception for invalid instr. sequence
invalid_ret_o	out	logic	exception for return address mismatch
stack_empty_o	out	logic	exception for empty shadow stack
stack_full_o	out	logic	exception for full shadow stack

Table 4.3: EXCEC interface

the return address on the shadow stack matches the one used within some function return. And finally, stack_empty_o and stack_full_o indicate an empty, respectively full, shadow stack.

Strictly speaking, the latter is not a CFI error but rather an indicator that further checks won't work because no more return addresses can be stored on the shadow stack. In some other approaches this case is handled by moving the oldest parts of the shadow stack to main memory in order to free some space. The stack_full_o signal is also asserted in case one of the recursion counters overflow.

A dedicated exception handler, which prints a CFI-related error message, resets the internal state so that the CFI module is not left dirty and then terminates program execution, is implemented for CFI exceptions.

The proposed scheme also considers interrupts, which are particularly important in embedded systems but mostly neglected in other academic CFI proposals. The EXCEC module in the CPU core makes sure that all sequences of CFI instructions, such as CFI_–

CALL and the succeeding JALR and CFI_CHECK, are executed atomically by delaying interrupts if need be. This ensures a valid state in the CFI module and only adds interrupt latency in the worst case. Also, EXCEC is able to enforce CFI during ISRs: All control flow transfers executed there are protected in the same way as transfers occurring during normal program execution. This is possible because CFI enforcement can be kept active the whole time since non-standard control flow transfers are used to jump to the ISR, respectively back, and neither affects the CFI state. When an interrupt occurs, the CPU core implicitly sets the PC to the ISR and no return address is pushed to the shadow stack. At the end of the ISR a dedicated mret instruction is used for returning, which also entails no shadow stack operations.

EXCEC is based on concepts of various other CFI proposals. It uses a shadow stack similar to the one in HCFI [CCAI16] and FIXER [DBGJ19], but does not need dedicated CFI instructions for interfacing the shadow stack. The concept does not require instrumentation of function returns and direct calls because existing RISC-V ISA instructions can be used to announce these operations. This implicit approach is also used in Intel's CET [Int19]. Furthermore, EXCEC only stores a reduced number of bits in order to reduce memory demands. For forward edge protection, the scheme uses HCFI's label approach, but extend it with an advanced version of the trampolines used in HECFI [SAD+16]. Dedicated recursion handling is implemented with counters for each shadow stack entry, which is an extended variation of the single recursion counter used in HAFIX [DHP+15]. Support for setjmp is implemented as in HCFI.

We believe that EXCEC offers a sound and efficient protection against CRAs. Basic attack examples are immediately detected by the hardware module. A quantitative evaluation of EXCEC in terms of its introduced overheads is presented in Chapter 6 along with evaluations of other CFI implementations described in this thesis.

CHAPTER 5

Instrumentation

Most hardware-enforced CFI concepts require custom instructions, meaning that the respective ISA needs to be extended. These new instructions must be added at specific positions in the code. Software-based approaches also require additional instructions, only here already existing ones are used. The general process of enriching programs with additional instructions is called *instrumentation*.

This chapter first describes the instrumentation procedure and possible methods thereof in Section 5.1. Then some challenges are discussed in Section 5.2 and a GCC plugin developed for all instrumentation tasks of this thesis is presented in Section 5.3.

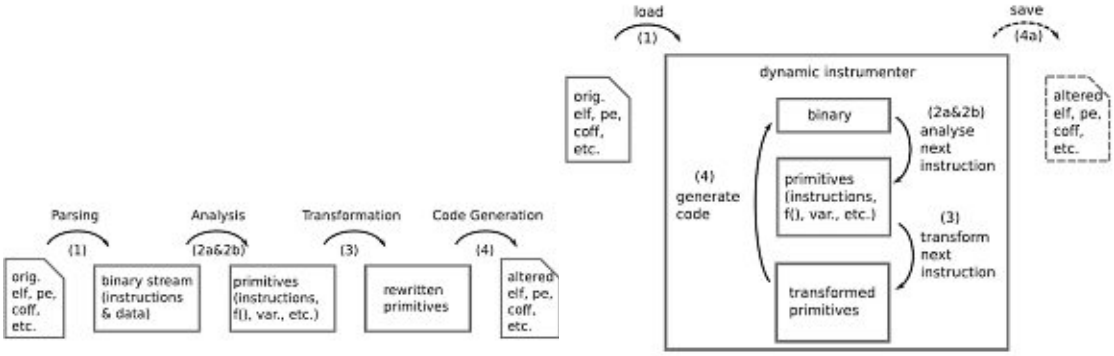
5.1 Introduction

Inserting or changing instructions is required in several areas, which are not limited to instrumentation for CFI. Other applications include emulation (translation of instructions between two ISAs), optimization (e.g., for runtime patching to avoid down times) and observation (profiling and tracing) [WMUW19]. In the context of emulation the basic concept is also called *dynamic binary translation* [Pro02]. Depending on the use case and the available artifacts, instructions can be added or modified during compilation, i.e., based on source code, or by manipulating some already compiled binary in the absence of its associated code.

In principle, some sort of instrumentation is required for most kinds of CFI concepts, regardless of whether they are software- or hardware-based. For example, the CFI schemes described in Chapter 3 follow very different approaches: In [DHP+15] instrumentation is performed fully automated by a compiler plugin, but the concept only adds CFI protection on backward edges. [CCAI16] intercepts the compiler at assembly level and instruments the code with a Python script before the linker step is executed. [SAD+16] uses an IDA Pro plugin for automatically instrumenting backward edges. Forward edges

require manual steps though. This last example is solely based on a binary and doesn't require its source code.

Using a disassembler like IDA Pro for injecting instructions is one approach of an overall instrumentation concept called *binary rewriting*. The main idea is to modify semantics of a program by adding, changing or removing instructions without having the program's source code at hand. Also, recompilation of modified applications is not necessary. There exist static and dynamic variants: Static rewriting describes the process of instrumentation in an offline way before actually executing an application. Dynamic rewriting changes the instructions at runtime. The availability of debug symbols or unstripped binaries is a great help here [WMUW19].



(a) static

(b) dynamic

Figure 5.1: Steps for binary rewriting [WMUW19]

Figure 5.1 shows the required steps for binary rewriting. The abstract principle is the same for static and dynamic cases. At first, the binary needs to be *parsed*, which involves translating the raw binary stream to mnemonics. This is particularly important for ISAs with dynamic instruction lengths because there the binary stream cannot be simply split into chunks of the same size for determining distinct instructions. The goal of the second phase is to *analyze* the instructions in order to recover the structure of the program, which includes identifying its basic blocks, functions and the program's CFG. As a third phase, the program is *transformed*, meaning that new instructions are added at specific locations or existing ones are altered. Transformation can be done in a *minimal-invasive* way by adding new parts to the program and redirecting the control flow there or by performing a *full-translation*, which allows manipulation at any position but requires additional steps. The last phase *generates code* again based on the previously transformed program [WMUW19].

The *parsing*- and *analysis* phases, i.e., steps 1 and 2 from Figure 5.1a, are not explicitly required when performing source code-based instrumentation during the compilation process because the outputs of the analysis phase (basic blocks, function boundaries and CFG information) are already implicitly available in the compiler. *Transformation* and *code generation* are of course needed anyways.

Techniques, which don't have access to source code or debug symbols, are not always able to recover all kinds of control flow transfers [BCN+17]. Also, more basic tasks like distinguishing code from in-lined data and detecting function boundaries are hard problems, which need to be considered [WMUW19]. On the other hand, concepts depending on source code or debug symbols have a limited scope of application because these artifacts are not always available [GABP14].

Furthermore, it is not possible to extract all symbol names in stripped binaries. This limitation becomes relevant when trying to match a CFG built from function names and labels with a CFG based on basic blocks generated from such a binary.

A typical use case for instrumentation without symbol information is replacing certain instructions with other instructions of the same length. Offsets and addresses don't have be adapted in this case [WMUW19].

Positions for Instrumentation

Depending on the respective CFI concept, programs have to be instrumented with additional instructions at some very specific positions. These are independent of the technique used for instrumentation. The exact positions also depend on the ISA, because, for instance, the (non-) existence of branch delay slots can determine whether instructions need to be placed right before or after certain positions in the code. The following enumeration contains relevant positions used in many existing CFI schemes:

- At **function entries**, still before any other instruction. Recursively called functions need special instrumentation in some approaches.
- At **the end of functions**. In some cases it can also be useful to write instructions *after* the last instruction of some function. Those are never executed within the normal control flow, but constitute a practical position for placing additional code like *trampolines*.
- At **return statements**. This is the common place for checking whether the return address is in accordance to the CFG and/or the shadow stack. Note that returns can occur more than once within a function.
- At **direct function calls**. Usually this is used for shadow stack operations for backward edge protection. A special form are recursive function calls because recursion can potentially overflow a (shadow) stack and might introduce uncertainty in the control flow. Some concepts therefore support special annotation of recursive calls, for instance in order to keep track of recursion-depth.
- At **indirect function calls**. In addition to the steps required for backward edge protection, also instructions for forward edges can be inserted here. Some approaches also add CFI instructions right after a function call at the address the callee returns to as an alternative approach for checking backward edges.

- At **indirect jumps**, similarly to indirect function calls, but without any operations related to backward edges because jumps are unidirectional. Note that switch statements in C are sometimes translated to indirect jumps by the compiler.
- At **targets of indirect jumps**, e.g. labels in a function, for checking function-local forward edges similarly to CFI at function entries. If the compiler generates indirect jumps for switches, their case statements are also considered targets.
- At **`setjmp/longjmp`** calls, which lead to control flow transfers outside of the strict *call-ret* pair concept of usual function calls. This problem was described in Section 3.2.
- Before the call of **`main`** and after its return. These positions are used to enable CFI enforcement, respectively disable it again. In addition, CFI can be reset before entering `main` such that every program execution starts from a clean CFI state.
- At **`exit`** calls, similarly to what happens when returning from `main`.

5.2 Challenges for CFI Instrumentation

The difficulty of instrumentation depends on the nature of the respective CFI concept. In principle, the backward edge protection approaches described in Section 3.1 are easier to implement because there are no dependencies between source and target of control flow transfers. Consider, for example, the backward edge CFI instrumentation used for HCFI [CCAI16] shown in Listing 5.1 and Listing 5.2. The injected instructions are not bound to certain function calls or returns in any way because they don't carry any parameters. Note that the CFI instructions in Listing 5.1 and Listing 5.2 are placed *after* the branch instructions they belong to. This is possible because the ISA used for the original evaluation of HCFI supports branch delay slots.

```
1 <foo>:
2 ...
3 call    <bar>
4 SETPC
5 ...
```

Listing (5.1) HCFI: Instrumentation of a direct function call

```
1 <bar>:
2 ...
3 ...
4 ret
5 CHECKPC
```

Listing (5.2) HCFI: Instrumentation of a directly called function

Forward edge protection with labels on the other hand, as described in Section 3.1.3, has dependencies: Such label checks require instrumentation of caller and callee with the same label. Listing 5.3 and Listing 5.4 show injected CFI instructions for HCFI, only now for an indirect function call. In this case, labels specific to the caller and the callee are encoded with the respective instructions.

However, there are more issues to consider apart from the position of instructions. Some other challenges are listed in the following paragraphs:

```
1 <foo>:
2 ...
3 call        $s2
4 SETPCLABEL  0x13
5 ...
```

Listing (5.3) HCFI: Instrumentation of indirect function call

```
1 <bar>:
2 CHECKLABEL 0x13
3 ...
4 ret
5 CHECKPC
```

Listing (5.4) HCFI: Instrumentation of indirectly called function

Availability of artifacts enables or limits the various approaches for instrumentation and also impacts the possible CFI precision. If no source code is available, compiler-based approaches are not viable at all. On the other hand, instrumentation difficulty inversely correlates with the degree of available information: Access to symbols, debug information or even source code allows for finer analysis and simplifies instrumentation.

Ideally, every CU is available as source code and all CUs are compiled and linked at the same time. Unfortunately, many libraries are only linked to applications statically or dynamically but compiled earlier.

Shared libraries can be instrumented at their own compile time, during a linker phase with LTO or by performing static or dynamic binary rewriting. For some CFI concepts the former approach introduces either a very coarse CFG or a lot of overhead in terms of unnecessary instructions because the required call information from the programs which use the library is not available yet.

Consider, for example, the demand for instrumenting programs with label checks during compilation (i.e., without binary rewriting) with static linking. When caller and callee are to be found in CUs, which are compiled at different points in time, fine-grained CFI is usually much harder. In such cases, the resulting CFG can loose its precision because the same label has to be used in many places. One possible solution for this case is LTO, which allows to perform instrumentation for all CUs previously compiled with LTO enabled during linking.

Extraction of CFG information, which is required for instrumentation, is a complex problem. While function returns and direct jumps and calls can be easily translated to their respective edges in a CFG, finding the targets of their indirect variants is way more complex. A frequently used method for finding possible indirect call targets is matching the type of function pointers with the signatures of functions, whose addresses are assigned to function pointers anywhere in the program. This introduces a very high false-positive rate though [LH19]. When neither source code nor any symbols are available, even extracting the CFG for branches is not trivial [WMUW19].

5.3 GCC Plugin for CFI Instrumentation

All of the hardware-based CFI schemes implemented for this thesis require instrumentation. Given the extent of benchmark applications used for evaluation, it is not feasible to manually add the required instructions for each concept. An automated and scalable method for injecting CFI instructions is required. We therefore perform all instrumentation tasks for this thesis in a compiler-based way. Our toolchain, as described in Section 4.1, comes with GCC - which can be extended with a plugin. This requires the availability of source code for all CUs where CFI enforcement is needed, so the approach is not applicable for commercial-off-the-shelf (COTS) applications.

However, we believe that this decision is legit within the context of this thesis for two reasons. First, instrumentation is a very important and necessary tool towards CFI enforcement but no focus in the strict sense of the topic of our work. The interested reader is referred to further literature on other instrumentation concepts like the extensive survey of binary rewriting techniques by Wenzl et al. in [WMUW19]. The second reason is that compiler-based code injection allows for very fine-grained CFI enforcement and does not suffer from the same problems as the analysis phases of binary rewriting, which were explained before in Section 5.1 and Section 5.2.

GCC Compilation Phases

GCC offers a large subset if its internal Application Programming Interface (API) to plugins and thereby allows to perform analysis, optimization and even modification tasks [StGDC20]. The latter is required for CFI enforcement, where additional instructions need to be injected at specific positions into some program.

When compiling an application, GCC runs through multiple phases called *passes*. At the very beginning the **Parsing** pass is executed, which reads and parses the input files and transforms them to a first intermediate internal representation. Then, the **Gimplification** pass is invoked, which further converts to another representation called *GIMPLE*. Then multiple **Inter-Procedural Optimization (IPA)** passes follow and perform transformations across function boundaries, such as constant propagation, function inlining and symbol cleanup. After that, **Tree Static Single Assignment (SSA)** passes perform local optimizations like removal of useless statements, tail call elimination and loop unrolling. Finally **Register Transfer Level (RTL)** passes are executed, which perform more optimizations and generate code at RTL [StGDC20].

Plugin Registration & Execution

GCC plugins must be registered for the desired phase in which they are to be executed. Our plugin was implemented as a RTL pass because GCC's internal representation of the code at this point is already close to the final assembly output. The basic code required for registering a GCC plugin is shown in Listing 5.5, where `GCC_PLUGIN` is the class of the actual plugin and must be derived from `rtl_opt_pass`. GCC runs through all of

these passes separately for each CU and only links the resulting object files together at the very end.

```
static struct plugin_info info = { "1.0", "CFI instrumentation};

int plugin_init(struct plugin_name_args *plugin_info,
        struct plugin_gcc_version *version) {
    register_callback(plugin_info->base_name, PLUGIN_INFO, NULL, &info);

    GCC_PLUGIN *gcc_plugin = new GCC_PLUGIN (context);
    struct register_pass_info pass_info;
    pass_info.pass = gcc_plugin;
    pass_info.reference_pass_name = "*free_cfg";
    pass_info.ref_pass_instance_number = 1;
    pass_info.pos_op = PASS_POS_INSERT_AFTER;

    register_callback(plugin_info->base_name, PLUGIN_PASS_MANAGER_SETUP,
        NULL, &pass_info);

    return 0;
}
```

Listing 5.5: Exemplary registration of a GCC plugin

When the respective optimization pass is executed, GCC calls the `execute` function of registered plugins for each function of the current CU. `execute` contains the actual plugin functionality and allows to iterate all basic blocks and instructions of the currently optimized function. Listing 5.6 shows a simple example of `execute` where first the names of the current function and the file in which it resides are extracted. Then the code iterates all instructions of all basic blocks with the `FOR_BB_INSNS` and `FOR_EACH_BB_FN` macros and prints debug representations.

```
GCC_PLUGIN::GCC_PLUGIN(gcc::context *ctxt)
        : rtl_opt_pass(cfi_plugin_pass_data, ctxt) { ... }

unsigned int GCC_PLUGIN::execute(function * fun) {
    char *function_name = (char*)IDENTIFIER_POINTER (DECL_NAME (
        current_function_decl) );
    char *file_name = (char*)DECL_SOURCE_FILE (current_function_decl);
    basic_block bb;
    bool recursiveFunction = false;

    FOR_EACH_BB_FN(bb, cfun){
        rtx_insn* insn;
        FOR_BB_INSNS (bb, insn) {
            debug_rtx (insn);
        }
    }

    return 0;
}
```

Listing 5.6: Exemplary `execute` function for a GCC plugin

Insertion of Instructions

Instructions are objects of the class `rtx_insn`, that represents nodes in GCC's internal code tree in general. All of the specific positions described before in Section 5.1 can be identified using fields of the `rtx_insn` instances and the macros provided by GCC. Such instances are typically nested and contain, for example, sub-nodes for the registers required for their respective functionality. The types of each node can be checked. It is possible to identify various different instruction types like function calls and returns, but also more advanced nodes like jump tables generated for switch statements. Listing 5.7 shows an example for how common return instructions are composed from `rtx_insn` nodes during the RTL phase of GCC. Here the outer node also contains a sub-node for the register `ra`, which contains the return address.

```
1 (jump_insn 466 465 469 (parallel [
2             (simple_return)
3             (use (reg:SI 1 ra))
4         ]) "kernel/init.c":184 974 {simple_return_internal}
5     (nil)
6 -> simple_return)
```

Listing 5.7: Debug representation of a return statement

Once the suitable positions are identified, new instructions can be injected. For simplicity assembly instructions are added in string format and not built from distinct `rtx_insn` nodes. Support for the latter would require deeper changes to GCC itself. When adding instructions only as strings, it is sufficient to define their formats, as shown in Section 4.2. GCC knows nothing about the semantics of such instructions.

Assembly instructions inlined in C code are also represented as string nodes, similarly to the instructions injected during instrumentation. This is a problem for our detection approach though because any of the relevant positions identified with `rtx_insn` node types can also occur as an inline assembly node, that cannot be identified in the same way. It would be possible to parse such string-format instructions and manually extract the required information. However, this was not implemented for this thesis.

It is possible to perform certain GCC passes in a unified way for all statically linked CUs by using Link Time Optimization (LTO) instead of executing the plugin separately for each CU. LTO moves some optimization passes to the linker phase, which allows for joint optimizations of the whole program instead of isolated ones for every CU. This feature can be enabled for GCC by compiling and linking with the `-flto` option.

By using LTO it is also possible to instrument libraries, that are not compiled at the same time as the user code of some program but only linked to the application. LTO must also be enabled for all shared libraries where instrumentation is required. This is particularly relevant for common libraries like *libgcc* and runtime libraries such as the ones of the PULP SDK in our setup.

Limitations

Unfortunately, it was not possible to compile *libgcc* with LTO. Our GCC plugin therefore only instruments the user code of applications and all functions of the PULP SDK. Exclusions are added to the plugin so that calls of *libgcc* functions are specifically handled. Otherwise calls of such functions would push their return address to the shadow stack, but this address is never popped because of the missing instrumentation of function returns in `libgcc`. However, `NOP` instructions are used instead wherever possible in order to achieve the same overhead of run time and code size.

In principle, the GCC plugin also works with C++ code. Indirect function calls used for *vtable* implementations, which are a very important concept of object-oriented programming in C++, are detected and instrumented as intended. However, the PULP SDK only supported compilation of C code at the time of writing this thesis, so instrumentation tests were only possible with the normal RISC-V GNU Toolchain[1], i.e., not with its PULP fork, without actually executing the resulting applications with the CFI hardware.

Instrumentation of most indirect control flow transfers requires explicit CFG information. Only the targets of function returns (which are, strictly speaking, indirect) and indirect jumps resulting from switch statements can be automatically determined. We use manually created CFGs and match node names with function names extracted from the `rtx_insn` in our approach. The following section describes how this information is generated and how the graph is represented.

5.3.1 CFG Information

Fine-grained CFI instrumentation requires CFG information in order to perform the instrumentation with necessary CFI instructions. A large part of the required information is provided implicitly by the source code, specifically the targets of direct jumps and calls but also the targets of function returns. However, information regarding indirect calls and some jumps needs to be provided and cannot be easily extracted.

Compiling the set of possible targets for indirect branches is a hard problem and involves some imprecision on the CFG in terms of over-approximation of allowed targets in most cases [dCV17, BCN+17]. For this thesis, CFG generation for indirect control flow transfers is considered out of scope. Manually created graphs for all of the applications used for evaluation (see Section 6.2 for a list) are provided. The resulting CFGs are probably more precise than any automatically generated ones can be. However, extraction of CFG information is a distinct and very extensive field of research on its own. We believe it is reasonable to put our focus on CFI itself, including the accompanying hardware- and software implementations.

Towards finding ways for encoding the CFG in some program, we first looked at ways for enriching source code directly with CFG information. Unfortunately, C provides no suitable annotation mechanism like, for example, Java does. Attributes such as `noline`

[1] `https://github.com/riscv/riscv-gnu-toolchain`

are remotely related to annotations and can carry parameters, but cannot be placed at arbitrary positions [Att]. Another idea was to use comments of a well-define format, but such are stripped by the pre-processor and can also not be used.

These reasons left us with using a configuration file for holding CFG information of indirect control flow transfers for each program. A convenient side effect of this approach is that the source code itself does not need to be changed in any way. The configuration file is completely external, does not harm source code semantics or syntax and could also be dynamically selected dependent on the actual CFI implementation.

An exemplary configuration file for the CoreMark benchmark[2], which uses an indirect function call for dynamically selecting a comparator function, is shown in Listing 5.8. The file first lists functions, that are potentially called in an indirect way and assigns labels to them. These labels are used for forward edge protection on the callee side in some CFI enforcement concepts.

```
1 # indirectly called functions
2 core_list_join.c cmp_idx label:13
3 core_list_join.c cmp_complex label:13
4
5 # indirect function calls
6 core_list_join.c core_list_mergesort line:551 label:13: core_list_join.c:
      cmp_idx core_list_join.c:cmp_complex
```

Listing 5.8: Example for a CFG configuration file

The second section in Listing 5.8 contains a list of actual indirect function calls, including their position in the source code (file name, function name and line number), the label provided for forward edge protection on caller side and a list of allowed call targets. The same principle is used for explicit indirect jumps too.

[2] https://github.com/eembc/coremark

CHAPTER 6

Evaluation

We implemented FIXER [DBGJ19], HAFIX [DHP+15], HCFI [CCAI16], HECFI [SAD+16], Intel CET [Gar20] and our novel CFI scheme called EXCEC presented in Section 4.5 on a common hardware platform and executed the same set of benchmark applications for all of these approaches in order to get comparable benchmark results. Here we present a wide range of evaluations and show that EXCEC outperforms existing academic schemes in terms of performance and code size.

Previous works in this field, such as the survey of hardware-based CFI by De Clercq and Verbauwhede in [dCV17], list various indicators like performance- and hardware overhead, if available at all, as stated by the authors of the respective proposals. Those are based on different platforms and are benchmarked with different sets of applications though, which makes a direct comparison not very meaningful. The interested reader is referred to the original papers as well as to [dCV17] for detailed descriptions of implemented CFI concepts and a qualitative security comparison of respective proposals.

This chapter first describes the methodology for all measurements in Section 6.1 and lists the benchmark applications used for evaluation in Section 6.2. Comparisons and benchmark results regarding security, performance, code size and hardware utilization are discussed in Section 6.3, Section 6.4, Section 6.5 and Section 6.6, respectively. Eventually a final comparison with a summary of these different aspects is presented in Section 6.7.

6.1 Methodology

The 32-bit PULPissimo[1] SoC with its CV32E40P[2] CPU core was used for all implementations. The platform is open-source and fully synthesizable, which enables convenient

[1] https://pulp-platform.org
[2] https://github.com/openhwgroup/cv32e40p

modifications of the core. This microcontroller architecture was described before in Section 4.1 in more detail. Platform-specific differences between our setup and the platforms used for evaluation by the authors of the original CFI proposals need to be considered when interpreting benchmark results. We therefore emphasize that the rankings presented here are only valid in our setup and do not necessarily represent criticism towards the original CFI implementations.

Benchmark Problems & Workarounds

The performance and code size evaluations in Section 6.4 and Section 6.5 are based on the set of benchmark applications listed in Section 6.2. The `slre` and `towers` applications are excluded from all aggregated performance indicators because of incompatibilities caused by recursive calls with two of the CFI implementations.

As described in Section 5.3, we were not able to instrument code in *libgcc*. Unfortunately, this limitation requires certain exclusions from CFI instrumentation. For CFI concepts, which need explicit instructions for backward edge protection, function calls are not instrumented with CFI instructions. This is because it is not possible to instrument the respective backward edges too, which would result in CFI violations because of shadow stack inconsistencies. However, `NOPs` are added in order to achieve the same number of injected instructions so that these exclusions neither influence performance overhead nor code size. The missing possibility to instrument functions in *libgcc* also poses a problem for indirect forward edges for Intel CET because with this approach the CFI hardware automatically enters a state where an `ENDBRANCH` instruction is expected upon every `JALR` and `JR`. It is not possible to inject these `ENDBRANCH` instructions in targets residing in *libgcc* though. As a workaround CFI enforcement is disabled in general for Intel CET when collecting benchmark results. This is allows to get meaningful run-time results at least.

Applied Optimizations

All applications are compiled using the PULP toolchain with its GCC 7.1.1, the PULP SDK and the compiler options `-O2 -g0` in order to enable common optimizations and disable output of any debug information. However, call-tail elimination is disabled with the GCC option `-fno-optimize-sibling-calls` because of its incompatibility with CFI. This problem was described in Section 3.2. Most applications are only less than 0.001% slower when not using call-tail elimination and some are not affected at all. There are certain cases though with clearly noticeable effects: For a simple `factorial` implementation the slowdown is almost 3000% because calls and returns are no longer optimized away. This is, however, independent of any CFI enforcement strategy.

Instruction Alignment

Instrumentation influences the time required for instruction fetching from memory. This is because any instrumentation, not necessarily for CFI reasons, changes the alignment

of symbols in the `.text` section of some program, which affects and sometimes even improves performance. When comparing run-time results of an application with and without CFI enforcement, a large share of the differences, which cannot be assigned to additional CFI instructions, stems from memory effects such as non-ideal fetches from instruction memory. These effects are not related to the nature of CFI instructions themselves but can also be caused by adding or removing arbitrary other instructions. RISC-V offers dedicated Control and Status Registers (CSRs) for getting both the current number of processed instructions and the cycles passed during execution. We believe that counting instructions instead of cycles results in more meaningful statistics since it clears these memory effects from our data.

Effect of ISA Variations

Compiling with the PULP ISA extension affects run-time performance of programs in most cases in a positive way. There are, however, also certain cases where execution of programs compiled *with* the extension is actually slower. These effects are not related to CFI. Therefore the average number of executed instructions for runs *with* and *without* PULP extensions are used throughout the evaluation, unless stated otherwise.

The unmodified PULPissimo SoC and program executions without any CFI enforcement acts as a baseline for all comparisons.

6.2 Benchmarks

A meaningful set of benchmark applications is required for testing, comparing and evaluating hardware-based CFI implementations. Throughout the evaluation part of this work CoreMark[3], most applications from the UC Berkeley RISC-V Benchmarks[4], all programs of the Embench-IoT Benchmark suite[5], some of the smaller applications in the MiBench Benchmark suite[6] as well as three small custom recursive applications are used. We believe this set of benchmarks constitutes a representative mix of applications for embedded systems.

Given our single-threaded environment, the multi-threaded part of the UC Berkeley RISC-V Benchmarks is not used. Furthermore, many of the applications of the MiBench suite contain code or input data too large for our setup (see Section 4.1) or require file IO, which is not possible on our platform without major modifications of the respective programs. Only a subset of this suite is used for this reason.

Table 6.1 lists all benchmark applications together with their source and indicators for whether the respective application uses indirect function calls, indirect jumps and/or recursion. Note that the applications might still contain indirect control flow transfers in

[3] `https://www.eembc.org/coremark`
[4] `https://github.com/ucb-bar/riscv-benchmarks`
[5] `https://github.com/embench/embench-iot`
[6] `http://vhosts.eecs.umich.edu/mibench`

shared library code. The table also shows which of the four academic hardware-based CFI proposals described in Section 3.3 used the benchmarks for evaluation in their original evaluation. Most of the CFI schemes were evaluated with other applications, which are not include in this work. It becomes apparent that `coremark` and `dhrystone` are frequently used by academia.

All of the applications listed in Table 6.1 work with our implementations of the four academic proposals, with two exceptions. The `slre` benchmark results in false-positive CFI errors for HAFIX because it contains nested recursive calls, which are not supported as also stated by the authors of the original proposal in [DHP+15]. Similarly, `towers` results in wrong CFI errors with HCFI, also because of limitations regarding recursion. However, we are not entirely sure whether this is a shortcoming of the original HCFI concept [CCAI16] or an implementation error of ours. In order to enable a fair comparison, `slre` and `towers` are not included in aggregated run-time performance results.

Our implementation of Intel CET [Int19], which is the only commercially available concept comparable to the academic proposals described in Section 3.3, can only be executed with disabled CFI enforcement for applications containing indirect control flow transfers in *libgcc* functions. This is because CFI enforcement on forward edges is implicit, but the callees cannot be instrumented for reasons described in Section 5.3. This does, however, not influence the run-time or code size of applications executed with our implementation of Intel CET.

6.2.1 Observations

We analyzed our set of benchmarks in order to get a feeling for suitable values for the common parameters listed in Section 4.4. This section gives insights to our observations.

Figure 6.1 shows the number of functions in each of the benchmarks listed in Table 6.1 when being compiled with the PULP ISA extension and GCC's optimization option `-O2`. Only `basicmath`, `cubic`, `dhrystone`, `fft`, `susan` and `wikisort` contain more than 100 functions. None falls below 80 or exceeds 125 different functions. Concepts like the active-set require this information in order to find a working and efficient size for the set.

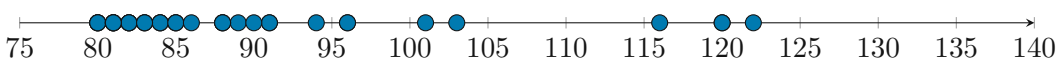


Figure 6.1: Numbers of functions in our benchmark applications

Figure 6.2 depicts the share of different types of control flow transfers in our applications. The numbers are rounded to the first decimal place. The vast majority of all control flow transfers are performed in a direct way, i.e., with statically known targets.

From our set of benchmarks only five contain indirect function calls in their *user code* at all, namely `coremark`, `bitcount`, `picojpeg`, `sglib` and `wikisort`. *User code* is the portion of code, which is specific to some application and does not belong to a

		Indirect transfers			Used in original evaluation			
	Benchmark	Calls	Jumps	Recursion	FIXER	HAFIX	HCFI	HECFI
	coremark	●	○	○	○	●	●	●
Embench IoT	aha-mont64	○	○	○	○	○	○	○
	crc32	○	○	○	○	○	○	○
	cubic	○	○	○	○	○	○	○
	edn	○	○	○	○	○	○	○
	huffbench	○	○	○	○	○	○	○
	matmult-int	○	○	○	○	●	●	○
	minver	○	○	○	○	○	○	○
	nbody	○	○	○	○	○	○	○
	nettle-aes	○	○	○	○	○	○	○
	nettle-sha256	○	○	○	○	○	○	○
	nsichneu	○	○	○	○	○	○	○
	picojpeg	●	●	○	○	○	○	○
	qrduino	○	●	○	○	○	○	○
	sglib	●	○	●	○	○	○	○
	slre*	○	○	●	○	○	○	○
	st	○	○	○	○	○	○	○
	statemate	○	○	○	○	○	○	○
	ud	○	○	○	○	○	○	○
	wikisort	●	○	○	○	○	○	○
UCB RISC-V	dhrystone	○	○	○	●	●	●	○
	median	○	○	○	●	○	○	○
	multiply	○	○	○	●	○	○	○
	qsort	○	○	○	●	○	○	○
	rsort	○	○	○	●	○	○	○
	sort	○	○	○	○	○	○	○
	spmv	○	○	○	○	○	○	○
	towers*	○	○	●	○	○	○	○
	vvadd	○	○	○	●	○	○	○
MiBench	basicmatch	○	○	○	○	○	○	○
	bitcount	●	○	●	○	○	○	○
	dijkstra	○	○	○	○	○	○	○
	fft	○	○	○	○	○	○	○
	qsort†	○	○	○	○	○	○	○
	stringsearch	○	○	○	○	○	○	○
	susan†	○	○	○	○	○	○	○
custom	factorial	○	○	●	○	○	○	○
	nqueens	○	○	●	○	○	○	○
	tak	○	○	●	○	○	○	○

* excluded from aggregated results due to missing support
† reduced problem set

Table 6.1: List of used benchmark applications

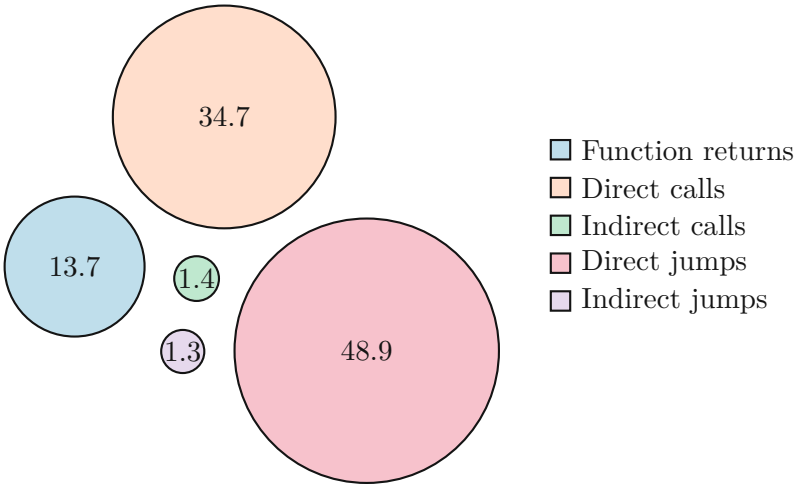


Figure 6.2: Relative shares of control flow transfers types in benchmark applications

shared library. In the `coremark` benchmark 15% of all function calls are indirect ones, respectively 0.1% in `picojpeg`, 26% in `wikisort` and even about 61% in `bitcount`. The `sglib` benchmark, however, does not execute its indirect calls because of conditional branch decisions at runtime. None of the application contains more than 10 functions, which are (also) called indirectly.

The `picojpeg` and `qrduino` benchmarks contain switch statements in the timed section of their code, which are translated to indirect jumps by the compiler. These hardly reflect on run-time performance at all though.

In total, less than 3% of all control flow transfers in the program (user code and libraries combined) are indirect function calls or indirect jumps. This information helps, for example, when deciding for a meaningful label width for forward edge protection. Sometimes it is also argued that protecting indirect function calls and jumps is less relevant in hardware because the share of these operations is so small [DHP+15].

The depth of nested function calls does not exceed 15 for most applications. Only dedicated recursive benchmarks like `tak` and `factorial` stack function calls higher. Those are custom examples of ours though and not part of any benchmark suite. This information about call- and recursion depth is relevant for dimensioning shadow stacks and optional elements for recursion handling. In general, it can be said that recursion occurs on multiple benchmark applications, but not in an extensive way.

6.2.2 Changes to the Benchmarks

The benchmark applications listed in Table 6.1 require certain minor modifications in order to be executable on our development platform. To the best of our knowledge, none of the modifications presented here changes application behavior in a relevant way. We

first state modifications, which were implemented for all applications, and then list those for specific benchmarks.

The timing of *all* benchmark programs is unified for ideal comparability. RISC-V CSRs are used for counting the number of instructions between start and end of each benchmark body execution. The relevant CSR is reset at the beginning of each run. Setup and verification code is excluded from timing and outputs of debug information or benchmark results are only printed outside of the benchmark body.

Certain includes and constants of the PULP SDK are required for *all* applications. The included headers link platform specific code like a custom version of `printf`, which outputs to the Universal Asynchronous Receiver Transmitter (UART) interface of the FPGA board.

Some applications require more memory on the stack than the 2 KiB defined in the original PULP SDK. In order to enable execution of more benchmarks, the stack size was increased to 18 KiB. Two programs of the MiBench suite, namely `qsort` and `susan`, were still not executable with this stack size. The problem set of these two applications was decreased so that they are runable on our platform.

Calls to unavailable functions of the C standard library were replaced with custom versions thereof because the PULP platform compiles with `-nostdlib` per default. The `rsort` benchmark application, which is part of UC Berkeley RISC-V Benchmarks, requires the `__sync_fetch_and_add` built-in function for atomic memory access. This function is not available on our platform. All calls were replaced with a custom software implementation, thereby losing atomicity. This does not affect the intended semantics since our platform is single-threaded anyways.

6.3 Security

Basic security tests were performed in order to check the effectiveness of our respective CFI implementations. A detailed qualitative security evaluation is no focus of this thesis. We instead refer to other works like [dCV17], which extensively analyze security concepts.

Exemplary applications containing simple memory corruption attacks that manipulate the return address or the target address of some indirect call or jump are used in order to confirm whether the resulting deviations from the expected control flow are detected by our CFI modules. All of these manipulations would enable CRAs. In addition to simple custom Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) examples, some tests of the Runtime Intrusion Prevention Evaluator (RIPE)[7] suite were used. RIPE is a RISC-V port of the considerable testbed for memory corruption vulnerabilities presented in [WNY+11].

A possible metric for a security evaluation and comparison of multiple CFI concepts is the Average Indirect target Reduction (AIR) proposed in [ZS13], which expresses the

[7] `https://github.com/draperlaboratory/hope-RIPE`

reduction of possible indirect branch targets. However, there is criticism about this approach. It is sometimes considered not particularly meaningful because most CFI approaches reach similar numbers when being evaluated with the AIR metric (usually $> 99\%$ reduction of targets) and the remaining unprotected branches might still offer enough targets to launch an attack [BCN+17].

While a CPU core without any enforcement mechanisms in place accepts a manipulated return address and simply continues execution there, all of the CFI concepts implemented for this thesis are able to detect the attack and throw a corresponding exception. Forward edge protection is implemented in all approaches but HAFIX, where the authors argue that software-side forward edge checks are sufficiently efficient [DHP+15]. Only HECFI, Intel CET and our EXCEC scheme support CFI enforcement on indirect jumps. Only CET and EXCEC support interrupts and also enforce CFI during ISRs.

Table 6.2 briefly summarizes the scope of security features of the CFI implementations. This functional scope must be considered when interpreting the following evaluation results for performance, code size and hardware utilization because these numbers on their own might give an distorted view.

	FIXER	HAFIX	HCFI	HECFI	Intel CET	**EXCEC**
Function returns	●	●	●	●	●	●
Indirect calls	●	○	●	●	●	●
Indirect jumps	○	○	○	●	●	●
Fine-grained CFG	●	○	◐	●	○	●
Protected interrupts	○	○	○	○	●	●

Table 6.2: Overview of the supported CFI protection scope

6.4 Performance

One of the most important parameters when comparing CPUs is runtime, i.e. the time required for executing some application. This section examines the effects of CFI enforcement on the runtime of our benchmark applications. The basis for all evaluations presented here is the number of instructions executed between start and end of the benchmark code. All run-time numbers represent the average values of executions *with* and *without* the PULP extension. Detailed and not aggregated run-time overheads of all CFI variants and all applications are shown in the appendix in Table A.1.

Overview

Figure 6.3 presents an overview of the introduced performance overhead. The chart shows the average relative increase for the benchmark applications listed in Section 6.2 in terms of additional instructions executed as compare to a baseline without any CFI enforcement in place.

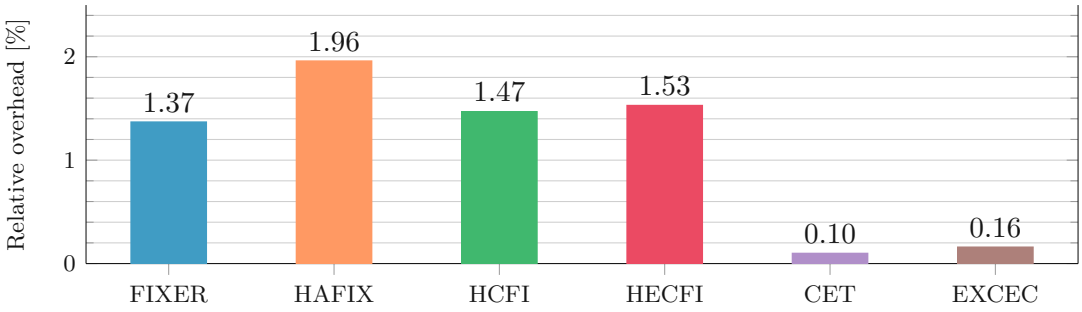


Figure 6.3: Comparison of performance overhead introduced by CFI schemes

Two observations stand out in Figure 6.3. The comparatively large run-time overhead of HAFIX is due to its higher number of instructions required for backward edge protection. As shown in Section 3.3.2, HAFIX requires CFI instructions at the very beginning and very end of *every* function. Furthermore, the concept needs one additional instruction for each function call. This sums up to at least three instructions for one direct call, which is more than any other CFI concept requires. The particularly low overheads of Intel CET and EXCEC on the other hand can be explained by the fact that these two approaches don't require additional instructions for backward edge CFI. 100 % of their introduced overhead results from forward edge protection on indirect control flow transfers. FIXER, HCFI and HECFI all use shadow stacks for backward edge protection and need the same number of instructions to interface with the stack. The relatively small differences between these three implementations stem from forward edge CFI, where FIXER only requires one additional instruction per call, HCFI two and HECFI three (or even more, in case trampolines are used).

Comparison for Selected Applications

The performance impact of our implementations depends on the number of additional instructions required for CFI enforcement. This value is a product of the number of relevant forward and backward edges present in the execution flow and the number of CFI instructions needed to instrument each of them. For example, the `edn` application does not contain any relevant control flow transfers in the timed section of its code, so naturally none of the CFI variants introduces any run-time overhead. Figure 6.4 shows the run-time increase for some other specific benchmarks, namely `basicmath`, `coremark`, `drhystone`, `factorial` and `bitcount`, where deviations between the various CFI implementations, but also between different applications, are significant.

The `basicmath` benchmark contains no indirect function calls and only small number of direct ones, leading to a minimal run-time overhead of 0.28% for most CFI implementations. An above-average number of direct calls is executed in `dhrystone` on the other hand. HAFIX is significantly slower here because its instrumentation concept adds instructions not only directly at function calls, but also to every function entry and exit. This leads to significantly more executed instructions in some cases, depending on

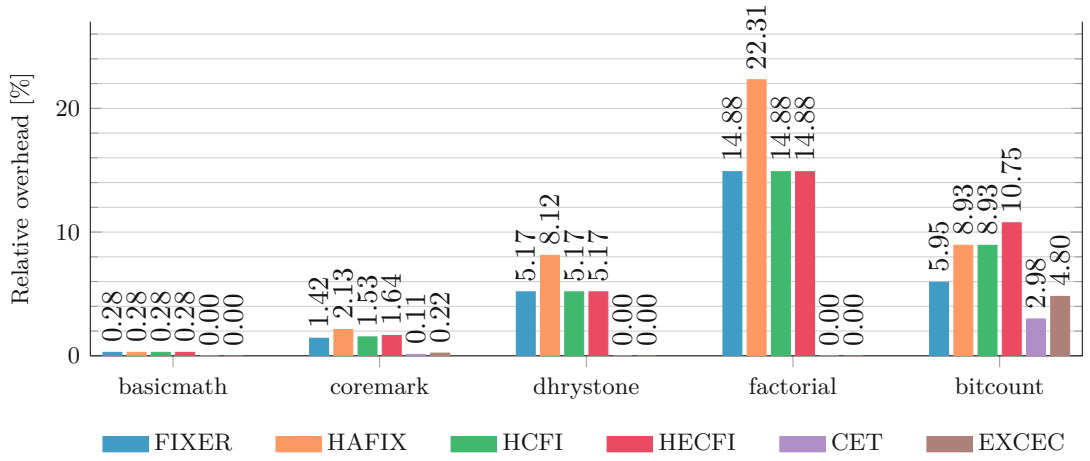


Figure 6.4: Comparison of performance overhead for selected applications

the structure of some program. Intel CET and EXCEC don't need instrumentation for direct calls, so there is no overhead at all for such applications. However, `coremark` and `bitcount` contain indirect function calls, which make up for as much as 15.4% of all calls for `coremark` and even 61.2% for `bitcount`. HECFI's instrumentation concept, which requires at least three instructions per indirect call, is particularly vulnerable here. The numbers for the recursive `factorial` application are particularly bad for the academic approaches. FIXER, HCFI and HECFI all use the same number of instructions for protecting direct function calls and returns. HAFIX requires more instructions for the same task, while Intel CET and EXCEC get along without any additional ones.

Comparison of Forward and Backward Edge CFI

CFI proposals differ in the way they protect forward edges in front of all, while backward edge CFI is implemented in a similar way with shadow stacks in most concepts. We therefore compare the average run-time overhead introduced for applications, where only backward edge protection is required, with the overhead for applications also containing indirect function calls and/or jumps in Figure 6.5. HAFIX is not considered in this evaluation because it does not offer protection of forward edges. The values for backward edge protection represent the average overhead for all of the applications described in Section 6.2, which do *not* contain indirect control flow transfers. Intel CET and EXCEC introduce no overhead at all when securing function returns because they don't require custom instructions but only couple their protection mechanisms to already existing ones.

The second series in Figure 6.5 shows the combined overhead of forward- and backward edge protection for the remaining applications, namely `coremark`, `picojpeg`, `qrduino`, `sglib`, `wikisort` and `bitcount`, which all contain indirect function calls and/or jumps. Note that indirect calls are not executed for `sglib` at runtime because of dynamic branch decisions. Functions, which could potentially be called in an indirect way, have to be

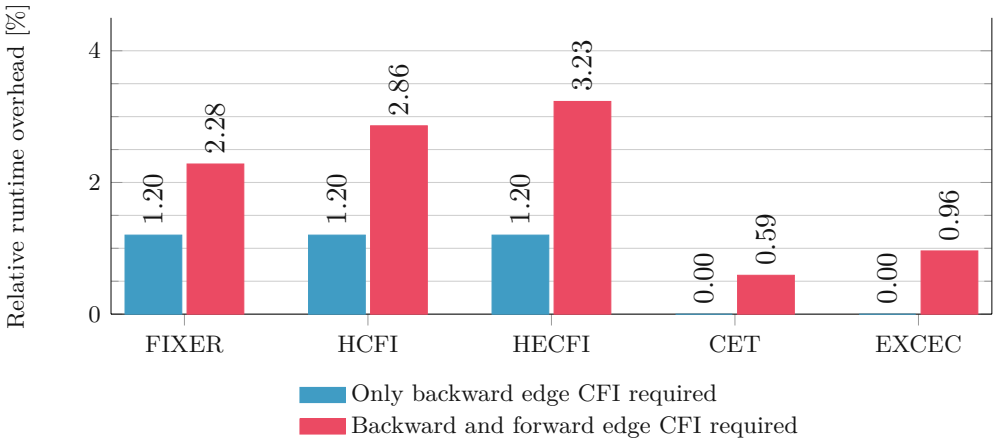


Figure 6.5: Comparison of performance overhead for programs only containing CFI-protected backward edges with applications also containing indirect calls and/or jumps

instrumented with CFI checks anyway. These instructions contribute to the performance overhead, even if they are not actually interpreted but are in fact treated similarly to `NOP`s during execution. Significant differences can be seen for these applications. HECFI requires the most additional instructions for indirect function calls and stands out here. Indirect jumps, which are used for switch-statements in `picojpeg` and `qrduino`, are rarely executed and only CFI-enforced in HECFI, CET and EXCEC.

Comparison with Original Evaluation Results

Ultimately we also compare the run-time overhead as measured for our implementations with the numbers stated in the original papers for FIXER, HAFIX, HCFI and HECFI. Concrete statements regarding overhead for Intel's CET were not available at the time of writing this thesis, so the concept is omitted here. Figure 6.6 shows a direct comparison of the respective numbers.

The main reason why our implementation of HCFI appears to be slower than the original one is presumably the fact that the authors of the original implementation could place their CFI instructions in branch delay slots, which eliminates some overhead. This is not possible on our RISC-V platform and is generally not always applicable in other architectures. However, the numbers for FIXER in Figure 6.6 need further explanation even though they are very close. The performance overhead measured by us represents an optimized version of FIXER with fewer additionally required instructions, as previously explained in Section 4.3. The set of benchmark applications used for the original evaluation in [DBGJ19] might be more beneficial for this approach because most programs therein contain only few function calls. In general, a portion of the differences for all of the schemes shown in Figure 6.6 can be explained by the different sets of applications used for evaluation.

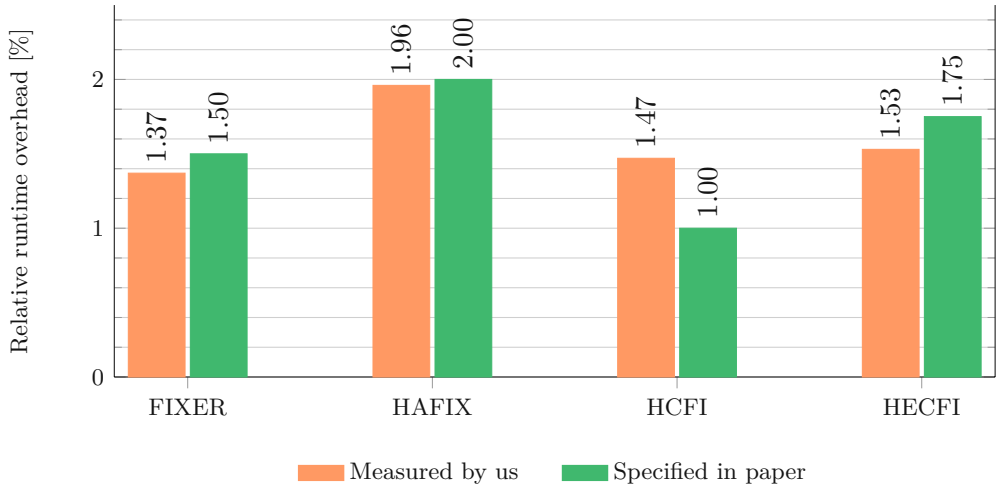


Figure 6.6: Comparison of measured and specified performance overhead

6.5 Code Size

Especially in the embedded world, where memory is limited compared to full-scale computers, the size of applications is of particular relevance. Therefore also the effects of CFI instrumentation on code size, i.e., size of the `.text` section in some binary, are considered for the benchmark applications. Not aggregated numbers for the overheads on code size for all CFI implementations and all benchmark applications are listed in Appendix A.2.

Overview

A basic analysis and in terms of the relative size overheads introduced by the CFI implementations is shown in Figure 6.7. The reason for HECFI being the worst here can be explained by its protection concept. HECFI injects CFI instructions just before and right after every function call while most other concepts add one instruction before the call and one before each of the callee's return statements. This difference means that HECFI adds two instructions for *every call* instead of just one. Similarly, HAFIX adds two instructions to every function, regardless whether it is called or not, and an additional one for every function call.

FIXER and HCFI handle direct function calls in the same way. Given the very small share of indirect calls in the benchmark applications, these two approaches bring a very similar number of additionally injected instructions with them. Intel CET and EXCEC don't require instrumentation for direct function calls at all, so their overhead in size is naturally minimal. The difference between the two of them can be explained by the fact that EXCEC instruments caller and callee with an additional instruction at indirect control flow transfers while Intel CET only injects one at the callee. Also, EXCEC adds trampolines, if needed.

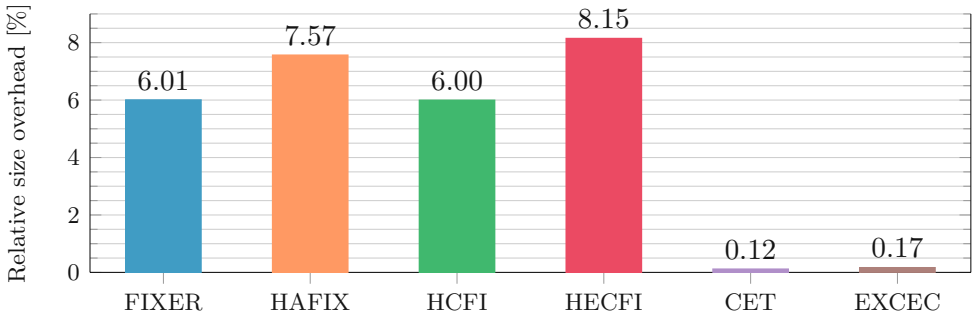


Figure 6.7: Comparison of code size overhead for various CFI schemes

It becomes apparent from Figure 6.7 that instrumentation of indirect function calls and jumps only accounts for a minor share of all injected instructions. However, all CFI variants show a wide range of `.text` section size overheads for different applications spanning from only slightly above 0 % to an increase of about 10 %. Figure 6.8 shows a more detailed perspective and utilizes a logarithmic scale for a better overview. The outliers for CET and EXCEC are of course applications with indirect control flow transfers.

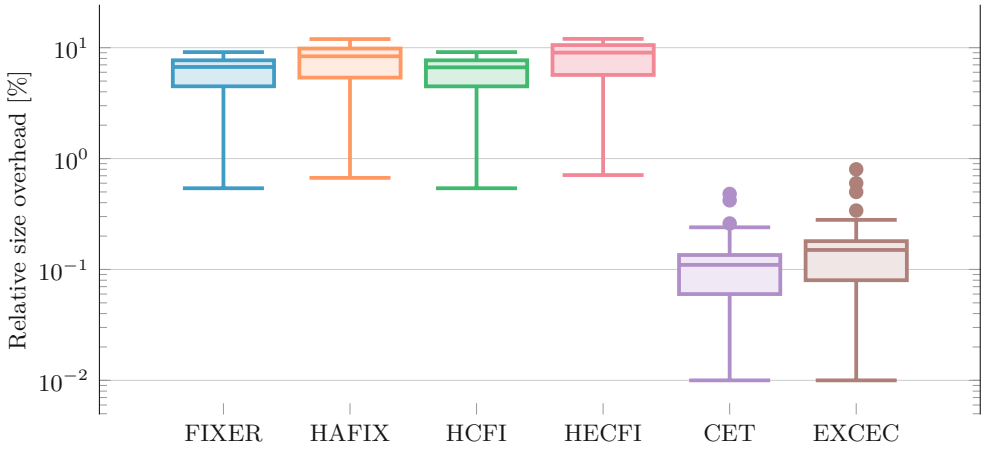


Figure 6.8: Comparison of code size overhead (log scale)

Correlation of Performance and Code Size Overhead

The overview numbers for performance- and code size overhead shown in Figure 6.3 and Figure 6.7 suggest that there is a correlation between the two of them. Figure 6.9 shows a direct comparison of these indicators. However, this correlation is not always true in individual cases because the overheads on performance and code size strongly depend on the nature of a program's CFG. For example, programs with many repeated function calls require minimal CFI instrumentation but show significant loss of performance.

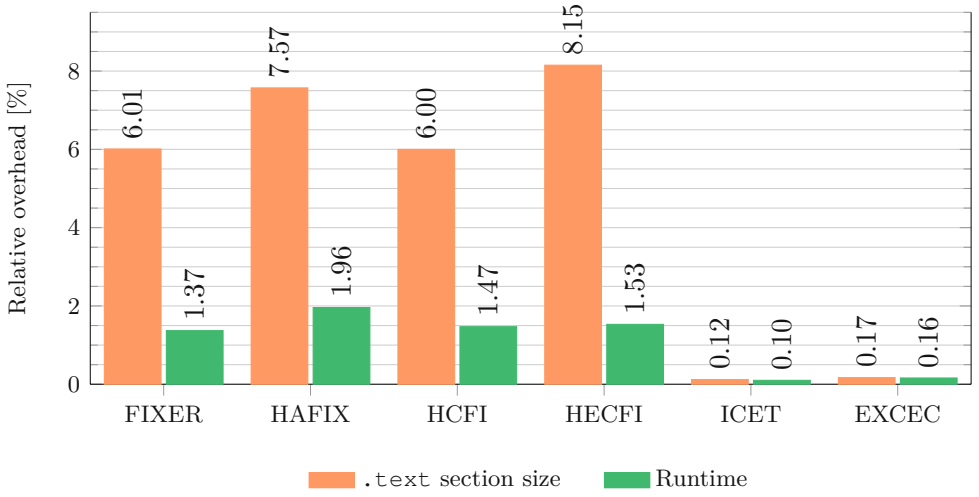
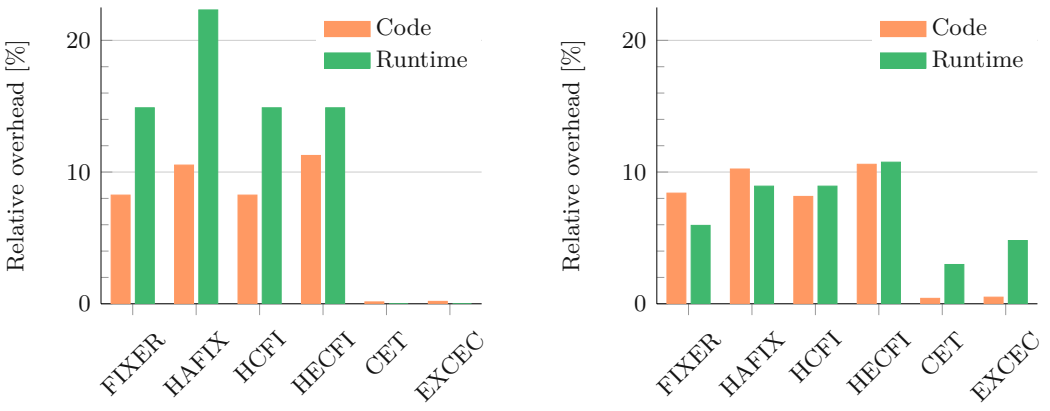


Figure 6.9: Comparison of code size and run-time overhead

An example thereof is shown in Figure 6.10a for the `factorial` application. Here the run-time overhead is significantly larger than the increase of code size because the very same instructions are executed over and over again in recursively called functions.

Another case is a comparison for the `bitcount` benchmark as shown in Figure 6.10b, which contains forward edge protection for indirect function calls. Here the relative performance- and code size overheads are almost aligned for the academic concepts. Note the significant differences to the average case shown in Figure 6.9 for both examples.



(a) `factorial` (b) `bitcount`

Figure 6.10: Comparison of code size and run-time overheads for specific programs

6.6 Utilization

Another very important aspect to consider when working with FPGAs or Application-specific integrated circuits (ASICs) is utilization of available hardware resources or area, respectively. We compare the effects of CFI-related changes to the CPU and use the utilization report created by Vivado shown in Table 6.3 as a baseline. These are numbers for synthesis and implementation of the complete PULPissimo microcontroller architecture on a Xilinx ZedBoard without any CFI enforcement in place.

Resource	Utilization	Available	Utilization [%]
LUT	40484	53200	76.10
FF	18624	106400	17.50
BRAM	128	140	91.43

Table 6.3: PULPissimo's base utilization without CFI enforcement

PULPissimo uses 512 KiB of data- and instruction memory, occupying the address space `0x1C000000 - 0x1C080000` [tea20]. This memory is implemented with BRAM elements on the FPGA, using about 91 % (128 of 140 blocks) thereof. BRAM blocks offer considerable amounts of memory, but introduce an access delay of at least one cycle [zyn19]. FFs on the other hand offer fast access, but are only available in limited numbers and additionally require accompanying LUTs. The available number of FFs from Table 6.3 directly corresponds to the available number of registers within the whole project. These factors need to be considered when implementing memory elements.

Each of the CFI approaches implemented for this thesis entails certain additional hardware requirements for logic and memory elements. While overhead in terms of LUTs depends on implementation details, the required additional memory is mostly determined by the dimensions of respective memory elements. The parameters defined in Section 4.4 were used for these dimensions in all CFI schemes to enable better comparability. All memory elements were implemented with FFs.

Overview

Figure 6.11 shows an overview of the introduced hardware overhead in terms of the absolute numbers of additionally required LUTs and FFs for each of the CFI variants. We consider a comparison of the absolute LUTs and FFs overheads more meaningful when comparing the effects of CFI implementations on hardware. Relative numbers are way more influenced by the setup. For example, enabling or disabling the Floating Point Unit (FPU), which is an optional component in the CV32E40P core, moves relative overhead numbers noticeably.

It becomes apparent from Figure 6.11 that FIXER has the highest hardware demands, which does not come by surprise because the concept includes both a shadow stack and a very expensive policy matrix.

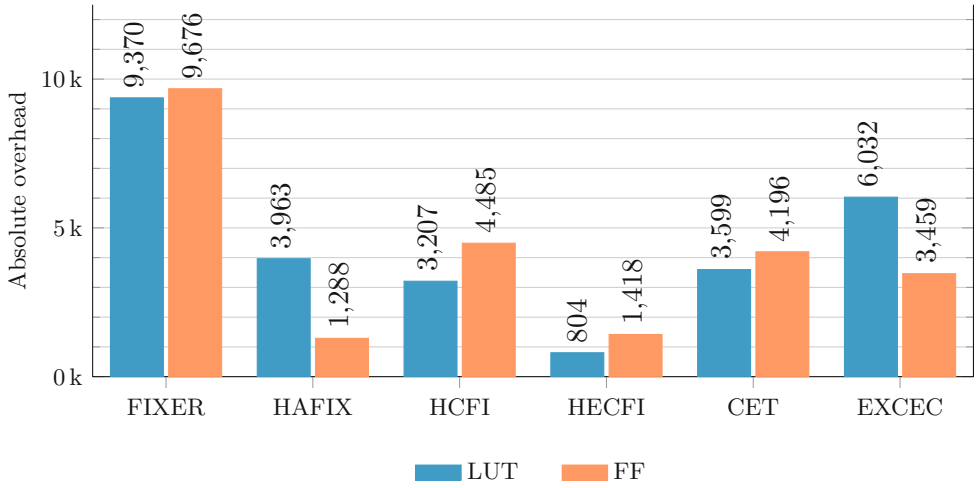


Figure 6.11: Comparison of additionally required hardware resources

HAFIX on the other hand does not include forward edge protection and only uses an active set for CFI enforcement on function returns, which is cheaper in terms of required memory but also less secure.

Partition of FF Usage

These observations suggest to take a closer look on how the overhead on FFs is composed in detail. Figure 6.12 shows this partition. The shadow stacks used in FIXER, HCFI and Intel CET for backward edge protection all require the same number of FFs because they store full 32-bit addresses. They all require 4096 FFs with a shadow stack size of 128, which is used for all of the stack implementations (as defined in Section 4.4). HECFI does not store full addresses but 10-bit labels on its stack, which results in a significantly smaller footprint.

Similarly, only 18 out of 32 address bits are stored in EXCEC, thereby cutting its demand of FFs almost in half for the shadow stack. EXCEC includes a particularly thorough handling of recursion though with counters associated to *every* shadow stack entry, which of course requires additional hardware resources. From the concepts analyzed and implemented for this thesis, only HAFIX does not use a shadow stack but an active-set. The authors of HAFIX suggested to use an active-set with 16384 entries in [DHP+15], which we consider disproportionate for embedded systems. Given our limited setup, we decided for with a smaller set with 1024 entries.

However, the differences between forward edge protection concepts are significantly larger. Label-based approaches, which are used in HCFI, HECFI, EXCEC and roughly also in Intel CET, hardly introduce any overhead in terms of FFs. Given the minimal size of the registers required for labels, the respective elements are placed in the *others* category in Figure 6.12. FIXER on the other hand implements forward edge CFI with a policy

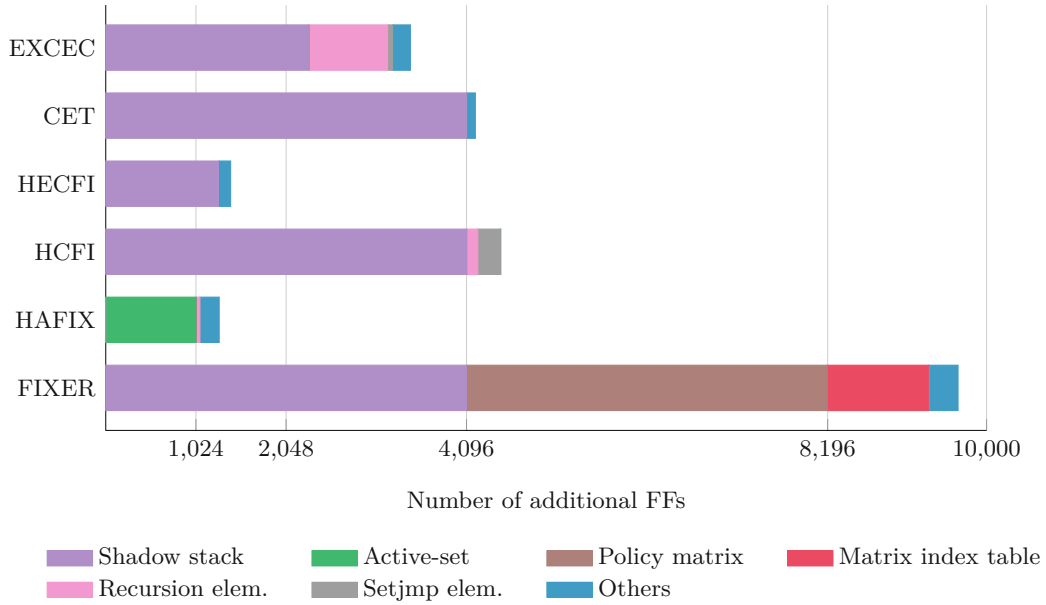


Figure 6.12: Partition of FFs usage for various CFI schemes

matrix, which is a very fine-grained and expensive approach. The matrix is implemented as an array with 64×64 entries, based on the parameters defined in Section 4.4. This component alone requires 4096 FFs. In addition, FIXER needs some mechanisms for translating 32-bit address to matrix indices. In our implementation this is a *matrix index table* with 64×18 bits, which stores the addresses used in the matrix and their associated indices. HAFIX does not come with forward edge protection at all.

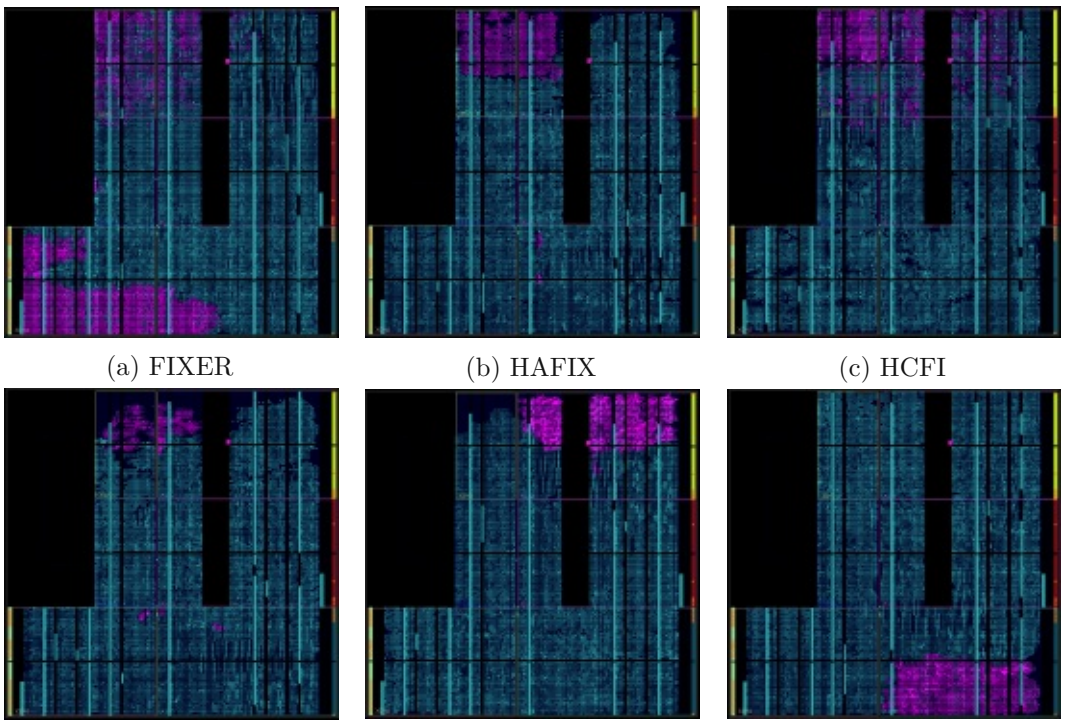
Label-based approaches are superior compared to a policy matrix in terms of required FFs. It can be argued that labels provide a less fine-grained protection, but improvements like *trampolines* close this gap. Furthermore, a 64×64 matrix, as used in our implementation and the original FIXER proposal in [DBGJ19], is rather optimistic.

Timing

The additional hardware modules required for CFI enforcement also impact the maximum frequency with which the SoC can be operated. Synthesis of the original PULPissimo SoC for the ZedBoard is possible with 16 MHz without timing violations with still some margin available. All of our CFI implementations influence the timing to some extent, but FIXER decreases this margin noticeably. However, it can still be synthesized without timing violations. In addition, our FIXER implementation has the highest number of logic levels because its memory elements require a large amount of associated logic for the lookup operations in the policy matrix. Higher frequencies can of course lead to problems in all CFI schemes.

Floorplans

The numbers presented in this section are also reflected in the FPGA floorplans shown in Figure 6.13. The turquoise areas represent cells used for PULPissimo itself, purple areas show the cells required for the respective CFI implementation and dark areas indicate free resources. FIXER shows a particularly high utilization. Note that Figure 6.13 only represents exemplary floorplans because placement is not deterministic and depends very much on the Electronic Design Automation (EDA) tool used.



(a) FIXER (b) HAFIX (c) HCFI

(d) HECFI (e) Intel CET (f) EXCEC

Figure 6.13: Visual device utilization comparison for the ZedBoard

6.7 Final Comparison

Eventually, Figure 6.14 shows the relative overheads on runtime, code size and hardware utilization in a combined way for a better overview. Note that the figure now shows the *relative* increase of hardware utilization (average relative overhead of LUTs and FFs) as compared to a vanilla SoC without any CFI hardware here. This is not meaningful when comparing different platforms, but can be used as an indicator for the impacts of various modifications to the same system.

The figure hints that FIXER's particularly large hardware utilization does not pay off because its run-time- and code size overheads are still at average levels, when being

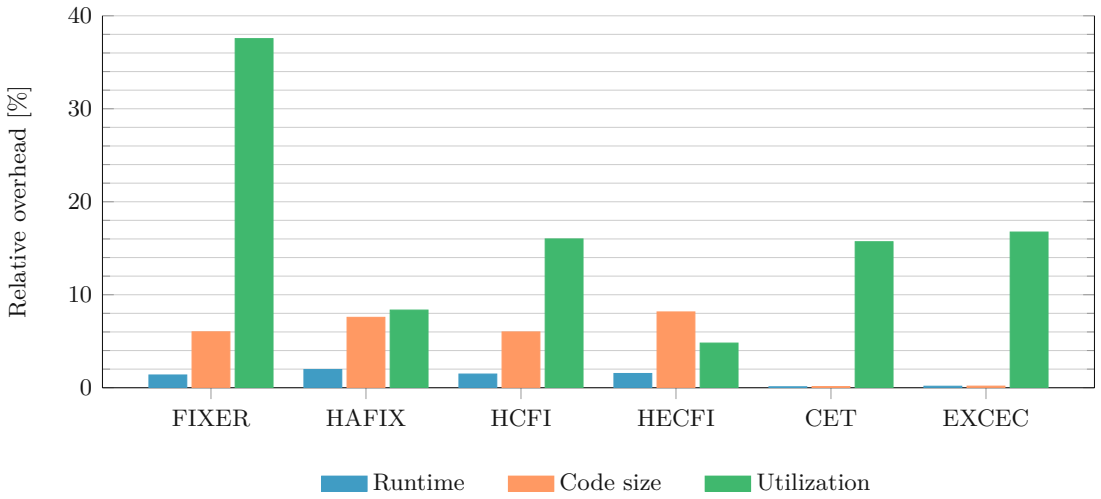


Figure 6.14: Comparison of runtime, code size and hardware utilization overheads

compared to the other academic approaches. The policy matrix, which is mostly responsible for FIXER's increased hardware demands, offers very fine-grained CFI, but similar results can be achieved with labels and trampolines too, if needed.

In contrast, HECFI entails the lowest hardware overhead because it uses labels for CFI enforcement on forward edges, which are practically free in terms of additional hardware, and a shadow stack with only a reduced width since HECFI stores 10-bit labels instead of full 32-bit addresses. The necessary additional instructions affect runtime and code size though.

Intel's CET and EXCEC appear almost identical in the comparison in Figure 6.14. Both introduce no performance- and code size overhead at all for backward edge protection and only a minimal overhead for CFI on forward edges. However, CET only offers coarse-grained CFI for indirect jumps and function calls. Compared to more fine-grained schemes like EXCEC, this imprecision of the CFG leaves more opportunities for attacks.

CHAPTER 7

Conclusion

The main goal of this thesis is to provide a meaningful quantitative comparison of various hardware-assisted CFG-based CFI schemes. This was enabled by implementing five existing approaches on a common RISC-V platform. Previous comparative works only analyzed security aspects and listed overheads on performance, code size and hardware utilization as stated by the authors of the respective proposals for different hardware platforms, if at all.

In addition to implementations of existing approaches, we designed and implemented a novel CFI scheme called EXCEC based on concepts and ideas from existing proposals. EXCEC offers a sound protection for all sorts of control flow transfers, considers elsewhere seldom supported concepts like `setjmp` calls and interrupts and comes with dedicated recursion handling. We also developed a GCC plugin for all instrumentation tasks, which injects required CFI instructions into applications during the compilation process.

Using these implementations, the impacts of CFI enforcement on runtime, code size and hardware utilization were compared. To that end almost 40 benchmark applications for embedded systems were ported to our platform, mostly from respected benchmark suites. While differences in performance and code size are rather small, some concepts stand out when it comes to hardware requirements - both in positive and negative ways.

Approaches for CFI protection on forward edges with labels and a policy matrix were evaluated. Hardware demands of the latter are extreme and the concept does not offer an otherwise unreachable level of security. Policy matrices, while being very intuitive and fine-grained, are not favorable in comparison to the very cheap but effective label-based approaches.

For backward edge protection shadow stacks and an active-set were compared. It is well established nowadays that only stateful protection, like it is offered by shadow stacks, provides perfect CFI enforcement while weaker approaches can be bypassed. However,

shadow stacks are usually more expensive compared to active-sets in terms of hardware utilization.

The platform on which a CFI concept is to be implemented has significant consequences. CFI modules implemented as coprocessors to some CPU can entail a comparably large performance- and code size overhead. In contrast, a deep integration into some CPU's pipeline allows for a very efficient implementation. CFI enforcement on backward edges is even possible without any additional performance- and code size costs when the respective CFI module has access to all required pipeline signals.

Platform-specific knowledge also allowed us to design a very efficient shadow stack, which only stores a part of instruction addresses. This is possible because only a small portion of the whole address space is used for instructions on some systems, especially in the embedded world where memories are rather small.

The probably biggest challenge remains the generation of CFG information, which is required for enforcement of indirect forward-edge CFI. In cases where no precise CFG is available, it is worth considering to use concepts like *Branch Regulation*, where only certain properties of control flows are enforced. An example for such a constraint is that every function call may only target the very beginning of any function. Intel's CET basically follows this coarse-grained approach.

In conclusion it can be said that the actual requirements and available resources decide which CFI approach is favorable: Coarse-grained CFI, e.g. with a combination of shadow stack and branch regulation, does not require a CFG, already prevents ROP attacks and significantly reduces the gadget space for other CRAs. More fine-grained approaches do need a CFG and further improve security.

In the course of working on this thesis we also wrote a paper on *Comparative Analysis and Enhancement of CFG-based Hardware-Assisted CFI Schemes* [TT21] and submitted it to the 48th International Symposium on Computer Architecture. It was rated with "Strong Accept" by two out of five reviewers but was unfortunately rejected after the rebuttal nevertheless. We are currently looking for another venue to publish it.

7.1 Future Work

There remain some possibilities for future work despite the extent of this thesis. On the software side, instrumentation is currently only possible for applications written in C because the PULP SDK does not provide support for C++ as of this time. Instrumenting C++ applications would be a valuable addition to our work, especially since concepts like the *vtables* used C++ should also be CFI protected. An evaluation of the effects on runtime would be obvious for such programs. It can also be interesting to implement a different concept for instrumentation like *binary rewriting* because our compiler-based requires the availability of source code.

The implementations presented in this thesis are based on a bare-metal system. An extension with OS support could give interesting insights, for example to evaluate how

an OS kernel can interact with the CFI hardware. This could also enable the evaluation of larger and more complex applications like web browsers. An increased performance overhead for such programs is expected because of dynamic linking and the use of object-oriented techniques.

Manually created CFG information for indirect function calls was used for instrumentation because CFG generation is a hard problem on its own and no focus of this thesis. A combination of our GCC plugin for instrumentation with some framework for extracting the control flow of a given program seems tempting and could solve this shortcoming.

The hardware modules for CFI enforcement were implemented as extensions to a single-threaded RISC-V SoC. Porting the CFI schemes to other ISAs with different characteristics would offer interesting insights for evaluating the effects of platform specifics like branch delay slots. Implementations for multi-threaded environments could also be considered but require more advanced concepts because CFI states for each thread context must be maintained.

APPENDIX A

Raw Data

A.1 Performance Overhead

Table A.1: Relative run-time overhead in terms of additionally executed instructions

Benchmark	FIXER	HAFIX	HCFI	HECFI	CET	EXCEC
coremark	1.423	2.135	1.533	1.643	0.110	0.220
aha-mont64	0.056	0.084	0.056	0.056	0.000	0.000
crc32	0.000	0.000	0.000	0.000	0.000	0.000
cubic	0.308	0.310	0.308	0.308	0.000	0.000
edn	0.000	0.000	0.000	0.000	0.000	0.000
huffbench	0.084	0.126	0.084	0.084	0.000	0.000
matmult-int	0.000	0.000	0.000	0.000	0.000	0.000
minver	0.655	0.982	0.655	0.655	0.000	0.000
nbody	1.069	1.069	1.069	1.069	0.000	0.000
nettle-aes	0.008	0.011	0.008	0.008	0.000	0.000
nettle-sha256	0.070	0.105	0.070	0.070	0.000	0.000
nsichneu	0.000	0.000	0.000	0.000	0.000	0.000
picojpeg	1.139	1.708	1.139	1.168	0.028	0.057
qrduino	0.174	0.261	0.174	0.176	0.002	0.004
sglib	2.987	4.481	3.105	3.105	0.118	0.118
slre	3.790	n/a	3.790	3.790	0.000	0.000
st	1.563	1.564	1.563	1.563	0.000	0.000
statemate	0.000	0.000	0.000	0.000	0.000	0.000
ud	1.837	1.886	1.837	1.837	0.000	0.000
wikisort	1.984	2.444	2.261	2.538	0.277	0.554
dhrystone	5.165	8.117	5.165	5.165	0.000	0.000

Continued on next page

Table A.1 – continued from last page

Benchmark	FIXER	HAFIX	HCFI	HECFI	CET	EXCEC
median	0.000	0.000	0.000	0.000	0.000	0.000
multiply	0.000	0.000	0.000	0.000	0.000	0.000
qsort	0.000	0.000	0.000	0.000	0.000	0.000
rsort	0.001	0.002	0.001	0.001	0.000	0.000
sort	0.000	0.000	0.000	0.000	0.000	0.000
spmv	1.435	1.435	1.435	1.435	0.000	0.000
towers	5.520	8.280	n/a	5.520	0.000	0.000
vvadd	0.000	0.000	0.000	0.000	0.000	0.000
basicmath	0.280	0.284	0.280	0.280	0.000	0.000
bitcount	5.952	8.929	8.929	10.750	2.976	4.797
dijkstra	0.153	0.230	0.153	0.153	0.000	0.000
fft	0.655	0.663	0.655	0.655	0.000	0.000
qsort	0.764	1.145	0.764	0.764	0.000	0.000
stringsearch	0.000	0.000	0.000	0.000	0.000	0.000
susan	0.083	0.083	0.083	0.096	0.000	0.000
factorial	14.876	22.314	14.876	14.876	0.000	0.000
nqueens	0.769	1.153	0.769	0.769	0.000	0.000
tak	7.260	10.890	7.260	7.260	0.000	0.000

A.2 Code Size Overhead

Table A.2: Relative code size overhead

Benchmark	FIXER	HAFIX	HCFI	HECFI	CET	EXCEC
coremark	7.24	8.67	7.19	10.59	0.26	0.34
aha-mont64	7.04	9.02	7.04	9.69	0.11	0.15
crc32	7.30	9.36	7.30	10.04	0.12	0.16
cubic	2.97	3.38	2.97	3.54	0.02	0.03
edn	6.15	7.90	6.15	8.37	0.10	0.13
huffbench	6.85	8.81	6.85	9.30	0.11	0.14
matmult-int	6.85	8.80	6.85	9.31	0.11	0.15
minver	6.55	8.18	6.55	8.75	0.09	0.12
nbody	5.64	6.73	5.64	7.12	0.06	0.09
nettle-aes	3.48	4.45	3.48	4.77	0.06	0.07
nettle-sha256	4.84	6.21	4.84	6.67	0.08	0.10
nsichneu	2.82	3.61	2.82	3.90	0.05	0.06
picojpeg	5.15	6.29	5.13	8.02	0.48	0.60
qrduino	4.77	6.08	4.77	6.64	0.24	0.28

Continued on next page

Table A.2 – continued from last page

Benchmark	FIXER	HAFIX	HCFI	HECFI	CET	EXCEC
sglib	6.85	8.55	6.75	9.84	0.13	0.80
slre	7.35	9.32	7.35	9.85	0.12	0.15
st	5.86	7.00	5.86	7.38	0.07	0.09
statemate	6.47	8.08	6.47	8.63	0.09	0.13
ud	6.41	8.17	6.41	8.67	0.10	0.13
wikisort	4.89	5.77	4.76	6.04	0.21	0.24
dhrystone	9.12	11.96	9.12	12.00	0.12	0.16
median	8.03	10.28	8.03	11.03	0.14	0.18
multiply	7.32	9.37	7.32	10.06	0.12	0.17
qsort	7.77	9.92	7.77	10.66	0.13	0.18
rsort	7.68	9.86	7.68	10.52	0.13	0.17
sort	7.71	9.80	7.71	10.52	0.13	0.17
spmv	0.54	0.67	0.54	0.71	0.01	0.01
towers	7.87	10.04	7.87	10.82	0.13	0.17
vvadd	8.01	10.26	8.01	11.01	0.14	0.18
basicmatch	2.98	3.39	2.98	3.54	0.02	0.03
bitcount	8.40	10.23	8.15	10.59	0.42	0.50
dijkstra	1.49	1.89	1.49	2.06	0.02	0.03
fft	4.20	4.97	4.20	5.32	0.04	0.05
qsort	6.90	8.79	6.90	9.55	0.11	0.15
stringsearch	2.92	3.72	2.92	4.00	0.05	0.07
susan	3.59	4.41	3.59	4.78	0.05	0.06
factorial	8.25	10.53	8.25	11.26	0.14	0.18
nqueens	8.07	10.32	8.07	11.13	0.14	0.18
tak	8.23	10.49	8.23	11.33	0.14	0.18

List of Figures

List of Tables

List of Listings

Acronyms

AIR Average Indirect target Reduction. 73, 74

API Application Programming Interface. 62

ASIC Application-specific integrated circuit. 81

ASLR Address Space Layout Randomization. 4, 10, 11, 18

BR Branch Regulation. 20, 28, 29, 31, 38

BRAM Block RAM. 31, 34, 45, 50, 81

CET Control-Flow Enforcement Technology. 20–22, 38, 39, 41, 44, 47, 56, 68, 70, 74–78, 82, 84, 85, 88

CFB Control-Flow Bending. 16, 17

CFG Control Flow Guard. 21, 38

CFG Control Flow Graph. xi, xiii, xv, 11–20, 23–28, 30, 31, 33, 36, 37, 50, 58, 59, 61, 65, 66, 79, 85, 87–89, 95, 99

CFI Control Flow Integrity. xi, xiii, xv, 1, 2, 11–39, 41–57, 59–63, 65–70, 73–79, 81–85, 87–89, 95–97

COP Call-Oriented Programming. 5, 10

COTS commercial-off-the-shelf. 62

CPU Central Processing Unit. 1, 9–11, 22, 24, 30, 31, 38, 42, 47, 55, 56, 67, 74, 81, 88

CRA Code-Reuse Attack. xi, xiii, xv, 4, 5, 7, 9, 10, 13, 18, 28, 56, 73, 88

CSR Control and Status Register. 69, 73

CU Compilation Unit. 30, 39, 46, 61–64

CWE Common Weakness Enumeration. 1

DEP Data Execution Prevention. 4, 10, 18

DIFT Dynamic Information Flow Tracking. 11

EDA Electronic Design Automation. 84

EXCEC EXtensive CFI Enforcement Concept. xiii, xv, 12, 41, 49–56, 67, 74, 77, 82, 85, 87, 95, 97

FF Flip Flop. 31, 42, 45, 50, 54, 81–84, 96

FPGA Field Programmable Gate Array. 32, 34, 42, 73, 81, 84

FPU Floating Point Unit. 81

FSM Finite State Machine. 29, 35, 38, 50, 53

GCC GNU Compiler Collection. 12, 21, 29, 42, 44, 48, 57, 62–65, 68, 70, 87, 89, 99

HDL hardware description language. 42

IBT Indirect Branch Tracking. 38

IoT Internet of Things. 1

IP Intellectual Property. 31

IPA Inter-Procedural Optimization. 62

IRM Inlined Reference Monitor. 18

ISA Instruction Set Architecture. xv, 5, 6, 10, 17, 19, 22, 31, 34, 36, 38, 41–45, 47, 50, 52, 56–60, 69, 70, 89

ISR interrupt service routine. 12, 49, 56, 74

JOP Jump-Oriented Programming. 5, 8–10, 73, 95

LIFO last in - first out. 26

LSB least significant bit. 50

LSS Label State Stack. 37

LTO Link Time Optimization. 21, 39, 48, 61, 64, 65

LUT Lookup Table. 42, 81, 84

MMU Memory Management Unit. 18

MPU Memory Protection Unit. 10

MSB most significant bit. 50

OS operating system. 1, 6, 10, 21, 22, 88, 89

PAC Pointer Authentification Code. 21, 22

PC Program Counter. 9, 31, 32, 35, 37, 45, 55, 56

PL programmable logic. 42

PUF physical unclonable function. 28

RILC return-into-libc. 5

RIPE Runtime Intrusion Prevention Evaluator. 73

RoCC Rocket Custom Coprocessor. 32, 33

ROP Return-Oriented Programming. 5–8, 10, 73, 88, 95, 99

RTL Register Transfer Level. 62, 64

SDK software development kit. 42, 50, 68

SFI Software Fault Isolation. 18

SMAC Software Memory Access Control. 18

SoC System-on-a-Chip. 32, 34, 41, 42, 45, 67, 69, 83, 84, 89

SSA Static Single Assignment. 62

UART Universal Asynchronous Receiver Transmitter. 73

Bibliography

[AAB+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. Rocket. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, CCS '05, page 340–353. Association for Computing Machinery, 2005.

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[and20] Control flow integrity | android. `https://source.android.com/devices/tech/debug/cfi`, September 2020. (Accessed on 10/14/2020).

[ARM15] ARM. D4.4.1 memory access control ARM architecture reference manual for ARMv8-A. `https://developer.arm.com/documentation/ddi0487/ag/`, 7 2015. (Accessed on 12/13/2020).

[Att] Attribute Syntax - Using the GNU Compiler Collection (GCC). `https://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Attribute-Syntax.html#Attribute-Syntax`. (Accessed on 11/09/2020).

[AW17] Krste Asanovic Andrew Waterman. The RISC-V Instruction Set Manual Volume I: User-Level ISA. `https://riscv.org//wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`, May 2017. (Accessed on 09/06/2020).

[BCN+17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[Ben20] Marco Benatto. Fighting exploits with control-flow integrity (CFI) in clang. https://www.redhat.com/en/blog/fighting-exploits-control-flow-integrity-cfi-clang, May 2020. (Accessed on 10/14/2020).

[BJF11] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pages 353–362, 2011.

[BJFL11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIA-CCS)*, pages 30–40, 2011.

[Bra16] David Brash. Armv8-a: 2016 additions. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions, October 2016. (Accessed on 10/14/2020).

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, pages 27–38, 2008.

[BST+00] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual Technical Conference, General Track (USENIX ATC)*, pages 251–262, 2000.

[CAS+17] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (S&P)*, pages 289–303. IEEE, 2017.

[CBP+15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[CCAI16] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 38–49, 2016.

[CH01] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 409–417. IEEE, 2001.

[claa] Control flow integrity — Clang 12 documentation. https://clang.llvm.org/docs/ControlFlowIntegrity.html. (Accessed on 10/14/2020).

[clab] Control flow integrity design documentation — Clang 12 documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. (Accessed on 10/14/2020).

[Cor19] Jonathan Corbet. Comparing GCC and clang security features. https://lwn.net/Articles/798913/, September 2019. (Accessed on 10/14/2020).

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Security 1998)*, volume 98, pages 63–78. San Antonio, TX, 1998.

[cv3] OpenHW Group CORE-V CV32E40P RISC-V IP. https://github.com/openhwgroup/cv32e40p. (Accessed on 11/07/2020).

[CW14] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

[CXS+09] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security (ICISS)*, pages 163–177. Springer, 2009.

[CZM+14] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against rop attack. *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.

[DBGJ19] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. FIXER: Flow integrity extensions for embedded RISC-V. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 348–353. IEEE, 2019.

[DCGÜ+16] Ruan De Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. SOFIA: Software and control flow integrity architecture. *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 68:16–35, 2016.

[dCV17] Ruan de Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (CFI). *arXiv preprint arXiv:1706.07257*, 2017.

[DDE+12] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, volume 26, pages 27–40, 2012.

[DHP+15] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[DZL16] Sanjeev Das, Wei Zhang, and Yang Liu. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(11):3193–3207, 2016.

[ES00] Ulfar Erlingsson and Fred B Schneider. IRM enforcement of java stack inspection. In *Proceeding 2000 IEEE Symposium on Security and Privacy (S&P)*, pages 246–255. IEEE, 2000.

[FC08] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, pages 15–26, 2008.

[GABP14] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy (S&P)*, pages 575–589. IEEE, 2014.

[Gar20] Tom Garrison. Intel CET answers call to protect against common malware threats. `https://newsroom.intel.com/editorials/intel-cet-answers-call-protect-common-malware-threats`, June 2020. (Accessed on 10/13/2020).

[grs16] On the Effectiveness of Intel's CET Against Code Reuse Attacks. `https://grsecurity.net/effectiveness_of_intel_cet_against_code_reuse_attacks`, June 2016. (Accessed on 10/15/2020).

[Int19] Intel. Control-flow enforcement technology specification. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, May 2019. (Accessed on 10/15/2020).

[JTC18] JTC 1/SC 22/WG 14. Programming languages — C. Std 9899:2018, ISO/IEC, July 2018.

[KOGP12] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu Ghazaleh, and Dmitry Ponomarev. Efficiently securing systems from code reuse attacks. *IEEE Transactions on Computers (TC)*, 63(5):1144–1156, 2012.

[LH19] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1867–1881, 2019.

[MIT20] MITRE. CWE - 2020 CWE Top 25 Most Dangerous Software Weaknesses. `http://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html`, 2020. (Accessed on 10/16/2020).

[msc18] Control flow guard. `https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard`, May 2018. (Accessed on 10/14/2020).

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, volume 5, pages 3–4, 2005.

[NT14] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 577–587, 2014.

[PH] David A Patterson and John L Hennessy. Computer organization and design RISC-V edition: The hardware software interface (the morgan kaufmann series in computer architecture and design).

[Pro02] Mark Probst. Dynamic Binary Translation. In *UKUUG Linux Developer's Conference*, volume 2002, 2002.

[pul] PULPissimo Github page. `https://github.com/pulp-platform/pulpissimo`. (Accessed on 11/07/2020).

[PW17] David Patterson and Andrew Waterman. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[QT17] Inc. Qualcomm Technologies. Pointer authentication on armv8.3: Design and analysis of the new software security instructions. `https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf`, January 2017. (Accessed on 10/14/2020).

[RKHB12] Mehryar Rahmatian, Hessam Kooti, Ian G Harris, and Elaheh Bozorgzadeh. Hardware-assisted detection of malicious software in embedded systems. *IEEE Embedded Systems Letters*, 4(4):94–97, 2012.

[SAD+16] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[SAS15] AliAkbar Sadeghi, Farzane Aminmansour, and Hamid Reza Shahriari. Tiny jump-oriented programming attack (a class of code reuse attacks). In *2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*, pages 52–57. IEEE, 2015.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, pages 298–307, 2004.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy (S&P)*, pages 48–62. IEEE, 2013.

[StGDC20] Richard M. Stallman and the GCC Developer Community. GNU compiler collection internals. *Free Software Foundation*, 2020.

[SWK+16] Takamichi Saito, Ryohei Watanabe, Shuta Kondo, Shota Sugawara, and Masahiro Yokoyama. A survey of prevention/mitigation against memory corruption attacks. In *2016 19th International Conference on Network-Based Information Systems (NBiS)*, pages 500–505. IEEE, 2016.

[tea20] The PULP team. PULPissimo: Datasheet. `https://github.com/pulp-platform/pulpissimo/blob/master/doc/datasheet/datasheet.pdf`, May 2020. (Accessed on 09/08/2020).

[TEB+11] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In

International Workshop on Recent Advances in Intrusion Detection (RAID), pages 121–141. Springer, 2011.

[TRC+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in *gcc & llvm*. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[TT21] Mario Telesklav and Stefan Tauner. Comparative analysis and enhancement of CFG-based hardware-assisted CFI schemes. *arXiv preprint arXiv:2103.04456 [cs.AR]*, 2021.

[TW17] Michael Theodorides and David Wagner. Breaking active-set backward-edge CFI. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 85–89. IEEE, 2017.

[Van20] Jens Vankeirsbilck. *Advancing Control Flow Error Detection Techniques for Embedded Software using Automated Implementation and Fault Injection*. PhD thesis, 2020.

[VTVW+17] Jens Vankeirsbilck, Venu Babu Thati, Jonas Van Waes, Hans Hallez, and Jeroen Boydens. Control flow aware software-implemented fault injection for embedded CPUs. In *2017 XXVI International Scientific Conference Electronics (ET)*, pages 1–4. IEEE, 2017.

[WJ10] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy (S&P)*, pages 380–395. IEEE, 2010.

[WMUW19] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37, 2019.

[WNY+11] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pages 41–50, 2011.

[ZDN20] ZDNet. Chrome: 70% of all security bugs are memory safety issues. `https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues`, May 2020. (Accessed on 10/16/2020).

[ZQQQ18] Jiliang Zhang, Binhang Qi, Zheng Qin, and Gang Qu. HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal (IoT-J)*, 6(1):458–471, 2018.

[ZS13] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, 2013.

[ZWC+13] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy (S&P)*, pages 559–573. IEEE, 2013.

[zyn19] 7 series FPGAs memory resources. `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`, July 2019. (Accessed on 09/08/2020).