On Bridging the Gap between Control Flow Integrity and Attestation Schemes
Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu,
Huawei Research, Germany
https://www.usenix.org/conference/usenixsecurity24/presentation/ammar

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA
978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
On Bridging the Gap between Control Flow Integrity and Attestation Schemes

Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu
Huawei Research, Germany
firstname.lastname@huawei.com

Abstract
Control-flow hijacking attacks remain a significant challenge in software security. Several means of protection and detection have been proposed but gaps still exist. To address such gaps, leading processor manufacturers have introduced new extensions in their latest-generation architectures, such as Pointer Authentication (PA) and Branch Target Identification (BTI) technologies in the ARMv8.5-A processor architecture. However, simply enabling these technologies would offer only limited security guarantees without trustworthy evidence of runtime integrity.

To bridge this gap, we propose CFA+, a practical hardware-assisted control flow attestation mechanism with prevention capabilities. CFA+ leverages ARMv8.5-A’s BTI security extension in combination with selective software instrumentation to enable lightweight always-on monitoring of the execution state without the need for maintaining in-memory control flow logs. The hybrid policy of CFA+ enables immediate prevention or quick detection of control-flow violations while providing trustworthy evidence of runtime integrity. CFA+ offers strong security guarantees for complex software stacks while maintaining high efficiency and scalability. Evaluation results demonstrate that CFA+ incurs an average runtime overhead of less than 3% when applied to various benchmark applications, including the SPEC CPU2006 suite and nginx.

1 Introduction
Despite advancements in system security, the foundation of most software stacks still depends on unsafe programming languages like C and C++ for their flexibility and performance benefits. However, these languages necessitate manual memory management, which often results in programming errors and memory corruption vulnerabilities, leaving software vulnerable to exploitation by adversaries. While traditional code injection attacks have been largely mitigated through the deployment of defenses like Data Execution Prevention (DEP) [72] and static Remote Attestation (RA) approaches [34, 86], control flow hijacking attacks, such as Return-Oriented Programming (ROP) [53] and Jump-Oriented Programming (JOP) [57], have gained popularity as effective alternatives for determined adversaries. These attacks exploit memory corruption vulnerabilities, such as buffer overflows and use-after-free bugs, to achieve malicious objectives, exemplified by privilege escalation. Adversaries accomplish this by manipulating a program’s control flow through the execution of carefully selected code snippets, known as gadgets. ROP attacks are characterized by chaining a sequence of gadgets, each ending with a return (ret) instruction. On the other hand, JOP attacks construct chains of gadgets on forward edges by targeting code fragments that end with indirect jump or call instructions.

Problem Statement. Over the past two decades, researchers have extensively studied Control Flow Integrity (CFI) schemes, which aim to mitigate control flow hijacking attacks by protecting both forward [13, 14, 17, 18, 35, 36] and backward [14, 18, 24, 36, 88] edges through enforcing various policies that must be respected during runtime. However, despite these efforts, such attacks continue to persist [19, 28, 33, 47, 58], highlighting the inherent limitations of enforcement-only solutions. These limitations arise from various factors, including the complex nature of target software [62], compatibility issues [60], and unbalanced trade-offs between security and performance [45]. Additionally, implementation mistakes represent a significant avenue for adversaries to exploit, bypassing detection by CFI defenses, which primarily operate at a local level [58, 62]. The concept of local detection within these mechanisms does not provide external evidence that could offer valuable insights into the execution flow of the target application, creating uncertainties regarding the current state of runtime integrity.

This, in turn, has motivated the design and implementation of Control Flow Attestation (CFA) mechanisms, which have emerged as a promising approach for providing external evidence of runtime integrity. As such, they offer crucial insights and enable remote detection of control flow violations [21, 22, 27, 56, 63]. The key idea behind CFA is the remote verification of the runtime integrity status of a target device/application, referred to as the prover (Prv), by an external trusted entity known as the verifier (Vrf). To ensure
trustworthy guarantees, \(Prv\) is typically equipped with a trust anchor to preserve the authenticity of collected execution traces that comprise the attestation report. This report is regularly or upon request transmitted to \(Vrf\) for verification.

However, despite the growing demand for verifying the runtime integrity of complex applications [21], the majority of existing CFA mechanisms are primarily designed for and limited to simple embedded systems [12,22,27,56,65]. Porting such solutions to tackle complex software stacks is hindered by several limiting factors. These include significant runtime overhead, insufficient scalability, and compatibility issues arising from intrusive hardware modifications.

While a few CFA proposals have been developed to tackle complex software [21,63], they still encounter performance and scalability challenges, as discussed in Section 3.2. Furthermore, these schemes primarily focus on remote attack detection, which typically occurs late. Considering the broader attack surface in complex software, early detection, similar to local CFI mechanisms, is preferable whenever feasible but without compromising the advantages of external evidence. Unfortunately, activating separate CFI and CFA mechanisms would impose significant performance overhead, rendering it impractical for various real-world applications.

In particular, existing CFA schemes have largely overlooked the potential benefits of incorporating elements from CFI mechanisms. Such integration has the potential to enhance performance, accelerate attack detection, and achieve greater scalability. This oversight has resulted in impractical solutions that struggle with performance and scalability challenges. Remarkably, the potential synergies between CFI and CFA approaches have not been extensively investigated, highlighting the significance of exploring this overlooked avenue to mitigate control flow hijacking attacks.

**Contribution.** We take a first step to address the aforementioned limitations by proposing CFA+, a hybrid runtime defense approach that combines the strengths of CFI and CFA approaches by offering both control flow attestation and prevention capabilities for complex user-space applications. Unlike traditional CFA mechanisms, CFA+ takes a different approach to attest runtime integrity without explicit monitoring and logging of execution traces or relying on costly and non-scalable verification procedures. The key idea of CFA+ is to offer implicit monitoring of the execution state through incorporating CFI building blocks in its design, prioritizing prevention (i.e. local detection) over remote detection, while always providing trustworthy evidence of runtime integrity. For this purpose, CFA+ reserves two general purpose registers, referred to as State Register (\(SR\)) and Reference Register (\(RR\)), which are leveraged to implicitly maintain awareness of execution integrity. To achieve so, CFA+ generates unique identifiers (IDs) for selected forward-edge control transfer instruction, depending on a pre-generated function-level Control Flow Graph (CFG). It then selectively instruments these instructions, to encode (before executing a call instruction) and decode (after returning from a callee function), via XOR operations, \(SR\) with the corresponding ID. This creates a two-way execution-dependent function-call chain in \(SR\), which can be used to assess runtime integrity without the need for in-memory history logs. Additionally, \(RR\) is employed to protect native code from vulnerabilities in shared libraries. To mitigate ROP attacks when backwarding (executing \(ret\), CFA+ applies a lightweight masking mechanism by XORing return addresses being pushed to or popped from the stack with \(SR\), enabling quick detection of corrupted return addresses.

To prevent bypassing the inlined instrumentation instructions and enhance prevention capabilities, CFA+ leverages the Branch Target Identification (BTI) feature, a security extension introduced as a CFI technology in the ARMv8.5-A architecture [6]. BTI restricts the possible targets for indirect branch instructions based on designated landing pad instructions. This ensures that indirect branches, including \(ret\), can only target landing pads, triggering a hardware-based exception otherwise. The integration of BTI forms the core of CFA+’s hybrid policy, enabling immediate prevention of non-BTI-compliant control flow violations or quick remote detection of sophisticated ones. Furthermore, CFA+ incorporates an efficient \(Vrf\) functionality based on SAT solvers. By considering the generated CFG and a minimal attestation report (including \(SR\), \(RR\), and Program Counter (\(PC\)) register values) as inputs, the SAT solver-based \(Vrf\) could detect control flow violations by identifying inconsistencies in \(SR\) and \(RR\) with respect to the execution context.

We evaluate CFA+ using various benchmarks, including the C benchmarks of the SPEC CPU2006 suite and the nginx web server. Evaluation results demonstrate that CFA+ introduces an average runtime overhead of no more than 3\% on \(Prv\), with an average increase in binary size by 22\%. Additionally, the verification process of attestation reports takes only a few seconds when executed on a laptop-class machine.

In summary, this paper presents several contributions:

- It introduces CFA+, an elegant hybrid control flow verification mechanism that efficiently combines runtime attestation and enforcement capabilities in a single design. CFA+ targets complex user-space software stacks at a large scale while incorporating a lightweight verifier functionality based on SAT solvers.
- It sheds light on the strengths and weaknesses of ARMv8.5-A’s Branch Target Identification (BTI) technology, integrating it as a CFI cornerstone within the CFA+ design to enhance its precision and granularity.
- It provides extensive performance evaluation, demonstrating superior advantages of CFA+ over existing state-of-the-art CFA and CFI mechanisms.

**Scope.** This work primarily focuses on C applications for simplicity, with the intention of discussing its applicability to embedded applications based on the ARMv8.1-M architecture. Extension to C++ applications, where particularly vtables should be handled, is left as future work.
2 Background

2.1 Fundamentals of ARMv8-A Architecture

ARMv8-A (also known as AArch64 or ARM64) is the first ARM architecture that supports a 64-bit address space, 64-bit wide registers and pointers, and 64-bit arithmetic operations, utilizing a 32-bit fixed-length instruction set, called A64 [5]. The architecture includes 31 64-bit general-purpose registers (GPRs), denoted as $x0 - x30$. Additionally, there are special-purpose registers (SPRs) such as the stack pointer (SP) and the program counter (PC). While the PC is not directly accessible as a GPR, it can be implicitly read by PC-relative address compute instructions.

ARMv8 ABI. The Application Binary Interface (ABI) defines the guidelines for executing code, including calling conventions and register usage. One aspect of the ABI focuses on GPRs, which also might serve special purposes. For instance, $x30$ functions as the link register (LR), responsible for returning the return address following a function call. As described in Section 4, CFA+ reserves two GPRs to ensure proper functionality without introducing compatibility issues.

2.2 Branch Target Identification (BTI)

BTI is a hardware feature introduced in the ARMv8.5-A and ARMv8.1-M processor architectures [4, 6]. It provides enhanced protection against control flow violations by offering landing pad instructions, which prevent the execution of unintended target instructions when a vulnerable indirect branch is executed. When BTI is enabled, the processor restricts indirect branch instructions to target only memory addresses that contain landing pads, which act as entry points to guarded memory blocks. Otherwise, a hardware-level exception is triggered, as depicted in the left half of Figure 1.

The encoding of landing pads looks like bti <target>, where <target> represents the assembler encoding of the type of the indirect branch instruction that is allowed to target such a landing pad. The ARMv8.5-A-based processor can distinguish between indirect jump and call instructions. Therefore, <target> can be one of three valid values, which are: c for indirect function calls, j for indirect jumps, and jc for all indirect branches. This distinction would reduce the number of possible JOP gadgets as shown in the right half of Figure 1. Notable is that return (ret) instructions represent a form of indirect jumps, which transform the control to the caller function by jumping to the loaded address in the link register (LR). Therefore, their possible valid targets can be restricted by inserting bti j after each call instruction. Yet, BTI alone offers only coarse-grained security guarantees.

3 Motivation & Related Work

3.1 Control Flow Integrity (CFI)

CFI is a principled approach that restricts all indirect branch instructions to adhere to a statically determined control flow graph (CFG) [36]. In general, CFI solutions insert inline reference monitors (checks) before indirect branch instructions, whose transfer targets could be compromised, to enforce that such instructions only jump to legitimate targets at runtime [14, 17, 18, 35, 59, 61, 74, 75]. The effectiveness of CFI mechanisms is strongly tied to the precision of the generated CFG and the enforced policy [45]. Consequently, CFI techniques are broadly categorized as coarse-grained or fine-grained. The former is simpler and less computationally expensive, favoring performance over security. The latter is more effective in detecting control flow violations as it inserts checks at a more granular level, imposing a higher performance overhead. CCFIR [18] is one of the early coarse-grained techniques that categorizes valid target addresses into three sets, each can be approached by only one type of indirect control transfer (ICT) instructions, i.e., return, indirect calls, or indirect jumps. While this approach reduces the attack surface (by acting as a software-based BTI), the number of potential gadgets that an adversary could exploit in complex applications is still quite large.

Several other coarse-grained CFI techniques have followed, which aimed at improving the accuracy of the enforced policy, without increasing the performance overhead [38, 59, 71]. However, bypassing such techniques has been demonstrated in various attacks [15, 19, 33, 46]. Therefore, a variety of fine-grained CFI approaches have been proposed [14, 17, 35, 74, 75]. Yet, the vast majority of them have not been widely adopted by industry due to their impact on performance. The few others that have been supported in mainstream compilers, e.g., LLVM CFI [17], are not comprehensive enough to completely prevent control flow violations [47, 52] or suffer from compatibility issues with some applications [60].

The accommodation of context-insensitive policies significantly contributes to the gaps present in CFI solutions [26, 47, 62]. Despite recent advancements, context-sensitive CFI schemes [11, 42, 43, 51] often suffer from limitations such as high performance overhead, compatibility issues, or false positives, which raise doubts about their ability to provide strong security guarantees [45, 60]. Additionally, the presence of implementation mistakes introduces additional hidden vulnerabilities that undermine the security of these approaches [62]. Consequently, the lack of external trustworthy evidence regarding the execution flow of the target application poses challenges in accurately assessing its runtime integrity status.

Figure 1: Possible valid targets by a vulnerable indirect call (BLR) or jump (BR) instruction in the presence or lack of BTI.
3.2 Control Flow Attestation (CFA)

CFA is a security service that enables a trusted party, i.e., verifier ($V_{rf}$), to verify the runtime integrity status of a prover ($P_{rv}$) application running on a remote device [56]. Unlike CFI, CFA offers a distinct approach to identifying control flow violations by leveraging insights derived from external evidence obtained remotely. To this end, $P_{rv}$ is instructed to build an authenticated log, referred to as $CF_{report}$, which contains pointers for all ICT instructions that have been executed since the last attestation operation. $V_{rf}$ will eventually receive $CF_{report}$ to verify the integrity of the execution state and detect potential control flow violations. $CF_{report}$ is commonly constructed by relying on software instrumentation [56], customized hardware [22], or a combination thereof [27]. C-FLAT [56] is the first to introduce CFA. It proposes a software-based instrumentation technique to collect and log execution traces, relying on ARM TrustZone to securely store and authenticate the collected measurements, before reporting to $V_{rf}$.

The main disadvantage of C-FLAT is the high-performance overhead due to frequent context switching to the TrustZone when executing any ICT instruction. As a follow-up, various CFA schemes with different assumptions and guarantees have been proposed [12, 16, 22, 27, 65]. Unfortunately, these schemes are mainly designed for simple bare-metal embedded software. Attempting to port such designs to complex user-space applications would either be infeasible or result in excessive performance overhead.

ScaRR [21] and ReCFA [63] are the only two CFA schemes that are aimed at handling complex software. In their essence, they still follow the traditional CFA approach of explicitly tracking and logging the execution order of ICT instructions. To handle complex software, they leverage various optimization techniques, such as skipping safe ICT instructions, compressing logs, and fragmenting paths into smaller parts. While both schemes address some of the shortcomings in other CFA designs, they remain limited due to some fundamental issues:

- **High runtime overhead** on $P_{rv}$: ReCFA incurs an average runtime overhead of 42.3% when applied to some benchmarks. ScaRR incurs a higher overhead than ReCFA due to its design, which depends on frequent context switching between the kernel and user modes. This overhead is not acceptable in real-world scenarios.

- **Limited scalability**: This implies different limitations on both $P_{rv}$ and $V_{rf}$, which are summarized as follows:
  - **The potential of Denial of Service**: Having several CFA-enabled $P_{rv}$ applications running on the same device and continuously sending $CF_{reports}$ to $V_{rf}$, as in ScaRR [21], might impact the availability of the device as this would consume a significant portion of the network bandwidth.
  - **Costly verification**: The design of both ReCFA and ScaRR requires a continuous and open-ended verification process of all executed control flow events by any $P_{rv}$ application. This approach is neither scalable nor efficient since the occurrence of attacks is typically an exceptional case.

- **Feasibility Concerns**: The design of both schemes overlooks the compatibility with static RA mechanisms in the sense that the reported results are not anchored in the same Root of Trust (RoT). This oversight not only complicates the attestation process but also raises concerns about the provided security guarantees. This is particularly crucial when aiming to ensure integrity through relying on some form of confidentiality, as is the case with ReCFA, which employs the Intel Memory Protection Keys (MPK) technology as a RoT [54], despite the reported limitations and security flaws [50].

3.3 BTI & Friends

Several hardware extensions have been recently supported in processor architectures to mitigate control flow hijacking attacks. For instance, the ARMv8.5-A architecture supports Pointer Authentication (PA), Memory Tagging Extensions (MTE), and BTI [7]. Notably, Apple has integrated the PA feature in several products [3], while Google has enabled MTE in their latest Pixel phones [2]. Furthermore, Academia has proposed various defenses, leveraging PA [23, 24, 55] as well as MTE [48]. Nevertheless, certain measures have been bypassed due to serious design flaws [9, 30, 62, 64].

On the other hand, BTI can be bypassed by a simple buffer overflow vulnerability, as visualized in Figure 2, where: (i) the else branch of the main function is entered and the `processInput` function is invoked (ln. 8 and 21), (ii) a buffer overflow vulnerability is triggered and overwrites the correct return address inside the `processInput` function stack frame (ln. 3), and (iii) returning inside the if statement body, where only privileged users are allowed to enter, without violating the BTI-enforced protection (ln. 4 and 13). Although Figure 2 demonstrates a ROP-like attack by manipulating backward edges, the same also applies to forward ones. Furthermore, the naive activation of BTI does not block control flow bending attacks [47]. Nevertheless, contrary to PA and MTE, BTI has the advantage of being simple and secret-independent, which are important features when leveraged in security designs.

```
1 main(void){
2     /* function that executes some syscalls, e.g., execv() */
3     void processInput(...);
4     void executePrivilege(void){
5         /* function that executes some syscalls, e.g., execv() */
6         void @processInput(void);
7         void processInput(char *arg[]){
8             /* buffer overflow vulnerability */
9             // @processInput
10             /* executePrivilege */
11             processInput: // @processInput
12             processInput: // @processInput
13             processInput: // @processInput
14             processInput: // @processInput
15             processInput: // @processInput
16             processInput: // @processInput
17             processInput: // @processInput
18             processInput: // @processInput
19             processInput: // @processInput
20             processInput: // @processInput
21             processInput: // @processInput
22             processInput: // @processInput
23             processInput: // @processInput
24             processInput: // @processInput
25             processInput: // @processInput
26             processInput: // @processInput
(a) Sample C code with a buffer overflow vulnerability.
(b) BTI-enabled assembly for ARMv8.5-A translated from (a).

Figure 2: A BTI-bypassable control flow violation Example.
```
4 CFA+

Goal. The primary objective of CFA+ is to bridge the existing gap between control flow integrity and attestation schemes. While these schemes diverge in various aspects, CFA+ specifically aims to provide both local and remote detection capabilities within a single, efficient, and scalable design.

Overview. CFA+ employs selective software instrumentation to implicitly monitor the execution flow and provide basic detection capabilities, while it depends on BTI as a principled CFI foundation to extend the detection capabilities and ensure proper execution of inlined instrumentation instructions. In contrast to traditional CFA approaches that often suffer from extensive measurement collection and reporting mechanisms, CFA+ utilizes XOR operations to encode and decode data in two dedicated registers, enabling implicit monitoring and the preservation of relevant state data during runtime. If a violation is detected, the application will optionally be terminated and the values of these two registers in addition to the PC register are logged in an append-only log file stored in non-volatile memory, e.g., a hard disk. The updated hash value of this file is always anchored in a RoT to verify its integrity when necessary. When required, \( \forall \) interacts with an attestation agent (\( \text{ATTTEST}_{\text{agent}} \)), which is a user-space process, to obtain a signed CF\(_{\text{report}}\) for all active CFA+-enabled \( \mathcal{P}_v \)-s. Subsequently, the runtime integrity status of each \( \mathcal{P}_v \) is verified based on the received CF\(_{\text{report}}\).

4.1 Threat Model & Assumptions

We consider a powerful adversary (\( \text{Adv} \)) attempting to achieve arbitrary code execution on a potentially flawed but benign application by exploiting memory corruption vulnerabilities to hijack the control flow. We assume a BTI-enabled hardware architecture, specifically ARMv8.5-A or higher versions. Moreover, we assume that the underlying OS kernel and hardware are trusted, providing the user space with essential protection features such as W\( \oplus \)X [78]. It is worth noting that the kernel should not store any user-level registers in user-accessible memory during context switches, which is the case in major OS kernels. Finally, we consider a \( \mathcal{P}_v \)-device equipped with a root of trust (RoT) like a Trusted Platform Module (TPM) [68] or ARM TrustZone [80]. Non-control data attacks, side channels, and physical attacks are beyond the scope of our threat model, which aligns with state-of-the-art CFI [14,17,25,39,42] and CFA schemes [21,63].

4.2 CFA+ Terminology

The following terms are used to simplify describing CFA+:

- **State Register (\( \text{SR} \)) and Reference Register (\( \text{RR} \))**: Two GPRs exclusively reserved for CFA+, which are utilized for maintaining relevant runtime state information.
- **Call Identifier (\( \text{ID} \))**: A distinctive hard-coded value assigned to each call instruction. It undergoes two XOR operations with \( \text{SR} \): (i) before executing the call to encode \( \text{SR} \), and (ii) after the call to decode \( \text{SR} \) and restore its original value.

![Figure 3: An overview of the entire attestation architecture of CFA+ (Blue/Red components are added to support CFA+).](image)

4.3 System Overview

An overview of the CFA+ architecture is depicted in Figure 3. During compilation, CFA+ employs a series of analysis and transformation passes (visualized as CFA+ plugin), to harden the target software with its instrumentation. The details of this phase are discussed in Section 4.4. On the BTI-enabled \( \mathcal{P}_v \) device, the kernel loads a custom kernel module, known as CFA+ Exception Handler (\( \text{EH}_{\text{CFA+}} \)), which performs certain actions once triggered by a BTI violation. These actions include logging essential attestation data and communicating with the onboard RoT to sign the updated log. The \( \text{ATTTEST}_{\text{agent}} \) process, responsible for sending and receiving attestation-related messages, can be implemented as an extension of any attestation agent used for static \( \mathcal{R}_A \). For instance, in the Linux kernel, the Integrity Measurement Architecture (IMA) [86] enables static \( \mathcal{R}_A \) through an attestation agent that can be adapted to support CFA+ with minor modifications. Whenever needed, \( \forall \) initiates the attestation procedure to verify the runtime integrity of all CFA+-enabled user-space applications running on a target \( \mathcal{P}_v \). Further details on the attestation process are provided in Section 4.5.

4.4 Compiler Instrumentation

4.4.1 Target Instructions

To detect control flow violations, CFA+ needs to instrument all ICT instructions, which can be divided into two categories: forward-edge instructions, i.e., indirect calls and jumps, and backward-edge instructions, i.e., returns. Abstractly speaking, ret instructions can also be considered as a form of indirect jumps, given that, in the AArch64 architecture, returning from any non-leaf function means: (i) the return address is loaded from the stack into \( \text{LR} \), and (ii) ret is then executed through jumping to the address in \( \text{LR} \). Therefore, one of the key principles of CFA+ is to bind the genuineness of stack-saved return addresses to the correct value of \( \text{SR} \). This can be achieved by instrumenting direct call instructions to non-leaf functions.

On the other hand, indirect jumps are mainly generated from switch statements that are translated to jump tables by compilers. These tables are usually read-only, bound-checked,
and have entries targeting addresses within the same function body. Therefore, the vast majority of CFI schemes pay attention to indirect calls, considering that indirect jumps are protected by default [28]. CFA+ follows the same principle but also adds some flags, i.e., -fno-jump-tables, in the compilation pipeline to disable the generation of these tables. While such a configuration might slightly increase the runtime overhead, it has the advantage of not only eliminating some JOP gadgets but also protecting against some microarchitectural attacks [49]. This is considered one of the main countermeasures to mitigate such attacks in the Linux kernel [20] and in some CFI mechanisms, e.g., RAP [87].

As such, CFA+ targets instrumenting indirect call and ret instructions, in addition to some direct calls, as a way to map forward- and backward-edges and stay context-aware.

4.4.2 Basic Form of Instrumentation

CFA+ relies on two main pillars: (i) software instrumentation, and (ii) BTI. Figure 4 illustrates the positioning of these pillars, considering a basic form of instrumentation, where CFA+ targets instrumenting three types of code elements:

- **Call Instructions**: This includes all indirect calls in addition to the direct ones that invoke non-leaf functions. A unique ID is generated for each selected instruction, which is used to encode and decode SR. A landing pad for ret instructions (bti j) is inserted after each call.

- **Function Prologue**: Each candidate function for indirect calls is guarded with bti c, followed by a block of instructions to verify the legitimacy of the caller instruction. A new label (an entry point), is added for direct calls targeting the corresponding function to skip non-relevant instrumentation instructions. The return address of each non-leaf callee function is masked with SR (i.e., LR = LR ⊕ SR) before being saved in the stack.

- **Function Epilogue**: This entails unmasking the return address by XORing with SR again before executing ret.

4.4.3 Detailed Working Mechanism

The following steps are performed during compilation:

1. CFA+ generates a function-level CFG for any target application. Consider the example application in Figure 5 (A), the resulting CFG is shown in Figure 5 (B).

2. CFA+ extracts the strongly connected components in the constructed CFG [76] and sorts them topologically [67]. It then traverses the CFG to analyze each function and extract relevant information, such as the function type (e.g., leaf, external, etc.), direct callers, number and type of indirect calls, the set of callees and their types, etc.

3. Based on type propagation analysis, CFA+ tracks the origin of each indirect call and constructs equivalence classes (ECs) accordingly. Each candidate function for an indirect call joins the corresponding EC. Partially overlapping ECs remain disjoint to maintain a high level of precision. Each EC is then carefully assigned a 48-bit unique identifier (IDEC) as multiples of 2 to distinguish individual bits as much as possible.

4. Considering the above analysis, CFA+ visits each function according to its topological order and generates a 64-bit unique ID for each call instruction (e.g., using a hash function). The 1D generation process for indirect calls considers generating only a unique 16-bit value, which when combined with the corresponding IDEC, the resulted combination represents the real unique 1D for any indirect call. A key property to maintain is that XORing any ID with one or more IDs in the same path should result in a unique value at the program level. Maintaining this requirement is feasible even in many complex applications, given the entropy of 64 bits. Considering that the return addresses of leaf functions are not stored in memory, their corresponding direct call instructions are skipped from ID generation or additional instrumentation, except for adding landing pads.

5. To instrument target calls, CFA+ reserves two GPRs (SR and RR) for its exclusive use, which are initialized to zero in the main function. Listing 4.1 showcases the base control-transfer instructions of three functions (func1, func2, and func3) from the ones shown in Figure 5.

Listing 4.2 depicts the instrumentation of func1’s call instructions, where each call is preceded by an encod-
Listing 4.1: Sample machine code (AArch64) for a subset of functions from the sample application shown in Figure 5.  

Listing 4.2: CFA+ instrumentation of func1 (Figure 4).

Listing 4.3 and Listing 4.4. Following by a block of instructions to validate the legitimacy of the indirect call instruction. The lower part starts with a symbol, introduced by CFA+, that has the same visibility as the corresponding function to act as an entry point for direct calls (which are rewritten accordingly in caller functions), skipping the execution of irrelevant instrumentation instructions (cf. ln. 8 in Listing 4.3 and ln. 7 in Listing 4.4). If the target function is a member in one EC and is targeted by one indirect call, the verification procedure is simple, where only the ID of the indirect call instruction is checked (cf. ln. 3-6 in Listing 4.4). If the expected ID is verified, the execution proceeds normally, bypassing the trap instruction. Otherwise, a BTI violation is triggered by indirectly jumping to an invalid random address (i.e., performing an indirect call to the middle of an arbitrary function using the value in LR (x30). Assuming that func2 can be targeted by more than one indirect call instruction, CFA+ generates, during the analysis phase, a mask value, which will be used to verify the legitimacy of the indirect call instruction. This per-function value is generated by ORing the IDs of all legitimate indirect call instructions to the underlying function. Performing a bitwise AND operation between the resulted mask and ID of the indirect call instruction should yield the same ID as a correct result (cf. ln. 3-7 in Listing 4.3). Given the way of generating IDs for indirect calls, bypassing this check by non-legitimate calls is difficult and would eventually be detected if happened.

8. To detect malicious tampering, the return addresses of indirectly called functions are also masked and unmasked in certain places as described in point 6 (cf. ln. 9 and 15 in Listing 4.3 and ln. 8 and 14 in Listing 4.4).

Corner Cases. The previous discussion did not delve into scenarios such as set jmp/long jmp, where non-traditional control flow transfers may occur, leading to a return to an unexpected location. While cases like set jmp/long jmp are considered unsafe and have become rare in production software, our approach has the potential to support them by modifying the relevant stack unwinding library in the underlying compiler (e.g., LLVM-Libunwind [69] in the LLVM/Clang compiler). The concept involves saving a copy of SR on the stack whenever the return address in LR is stored there. Upon reloading LR, the copy of SR would be discarded. During stack unwinding for exceptional circumstances, each intermediate stack frame between the current stack pointer and the target stack pointer needs to be traversed to restore each stored copy of SR and perform an XOR operation with the current value. This serves as an alternative approach to the typical SR decoding method. In this particular case, even if
malicious tampering occurs with the copies of $SR$, it would not compromise the security of CFA+, as it would result in an inconsistent $SR$ value in the target location, which would eventually be detected.

### 4.4.4 Handling Shared Libraries

The previous discussion aligns well with statically compiled software. However, complex applications usually rely on shared libraries, which can be used by any process running on the same device. In general, handling shared libraries in runtime defenses is a challenge by itself, which is not considered by many defenses, including the recent ones that depend on Pointer Authentication (PA) [23]. Therefore, CFA+ proposes a lightweight protection approach for shared libraries that brings two main advantages. First, it maintains compatibility with legacy software stacks. Second, it shields native CFA+-enabled software from exploitable vulnerabilities in shared code. To achieve so, CFA+ requires compiling shared code with extra flags that would do the following:

- Reserve $SR$ and $RR$ to ensure they remain uncorrupted by any instruction in shared code. For instance, to exclude the $x27$ and $x28$ registers from the register allocation phase in the LLVM/Clang compiler, the following flags should be added to the compilation pipeline: 
  ```plaintext
  -ffixed-x27 -ffixed-x28.
  ```
- Emit `bti c` instructions at the start of any address-taken function. Additionally, after each call instruction, emit `bti j` instructions. This lightweight instrumentation provides coarse-grained protection to shared libraries without introducing compatibility issues with legacy code, as the added landing pads will be interpreted as NOPs for any non-BTI-enabled code.

On the $Prv$ side, the instrumentation of calls to external functions in shared libraries is illustrated in Listing 4.5. The main difference, compared to the instrumentation of calls to native functions, is that $SR$ is backed up in $RR$ before executing the encoding $XOR$ and call instructions (cf. ln. 3–4), and it is immediately verified after executing the decoding

```
1We note that CFA+ considers the Procedure Linkage Table/Global Offset Table (PLT/GOT) entries protected using the available security features in compilers or kernels, such as Relocation Read-Only (RELO) [82].

### 4.5 Runtime Attestation & Verification

CFA+ not only prioritizes prevention over remote detection but also reduces the frequency of $Vr$-$Prv$ interaction, as signs of violations can be inferred at any time from $SR$ and $RR$.

#### Attestation Procedure

The instrumented $Prv$ application is expected to run normally. When a BTI violation is triggered, control will be transferred to $EH_{CFA+}$, which maintains a central append-only log file for all $Prv$-s. This log file is stored

Listing 4.3: CFA+ instrumentation of $func2$ (cf. Figure 4).

```assembly
Listing 4.4: CFA+ instrumentation of $func3$ (cf. Figure 4).

Listing 4.5: CFA+ instrumentation of a call to external func.

XOR instruction (cf. ln. 6-10). Given that each call instruction has a unique ID, returning to a different location than the expected one would be immediately prevented due to the mismatch between $SR$ and $RR$ values.

As can be seen in Listing 4.5, the instrumentation of calls to external functions is further optimized, where $XOR$ instructions operate directly on immediate values. Thus, extra load instructions to load the hard-coded IDs are no longer needed. A crucial aspect in the ARMv8 architecture is that logical instructions, e.g., $XOR$, accept specific 13-bit bitmask immediate values as a third operand, which each would map to a 64-bit unique value when the instruction is executed [5]. This mechanism can generate 5,334 unique 64-bit values. For instance, the shown bitmask $0x17EB$ would be translated to a 64-bit ID equal to $0x0FFFFFFFE00003FF$. Leveraging these pattern-based generated values as IDs would yield many collisions in terms of getting the same $SR$ value in different locations. This will not be the case for call instructions to external functions, given that $SR$ will not be updated by these external callees and it will be immediately checked upon return.

```

```

```

```
in non-volatile writable memory, such as a hard disk. EH_{CTA+} will then add a new record in the log file, trigger the RoT to sign the hash value of the updated version of it, and optionally terminate or restart the victim P_{rv}. The record will contain (i) the P_{rv} process name and ID, (ii) IMA measurement (if exists), which is the hash value of the P_{rv} binary image when loaded, (iii) the values of S_R, R_R, and P_C, and (iv) a binary flag indicating whether this record is added due to a violation (1 means a violation, and 0 otherwise). When necessary, V_rf asks ATTEST_{agent} to send CFA\_report for all CFA+ enabled P_{rv} applications running on the same device. Taking into account the potential compromise of certain P_{rv}-s, which may result in a violation of the expected control flow without local detection (i.e., no BTI violation), ATTEST_{agent} instructs EH_{CFA+} to record the most recent execution status of all active P_{rv}-s. EH_{CTA+} will then create an information record for each P_{rv}, indicating the latest values of S_R, R_R, and P_C, while setting the binary flag to 0 to indicate its inclusion for attestation purposes only. Eventually, V_rf will receive a signed CFA\_report from ATTEST_{agent}, which mainly consists of the log-file records in addition to the RoT signature.

**Verification Procedure.** After successfully verifying the integrity of CFA\_report, V_rf can verify the runtime integrity status of any P_{rv} in the log file. P_{rv}-s with records indicating violations are considered compromised without verification. For other P_{rv}-s, V_rf does the verification by checking the consistency of the reported register values w.r.t. the legitimate control flow of the corresponding CFG, as depicted in Figure 6. Given that the idea of CFA+ is to dynamically label the CFG edges depending on the execution context, where S_R always holds the label of the last visited edge by P_C, one way to verify the control flow integrity is to extract the related path and check the consistency of the reported values, e.g., the value of S_R should equal to the XOR operation of all \textsc{id}s starting from the root function until reaching the last ICT instruction executed by P_C. To achieve so, V_rf turns the path(s) extraction process into a Boolean satisfiability problem, to be solved by a SAT solver. The input to the SAT solver would be (i) the CFG of the corresponding P_{rv}, represented as a complex data structure that contains all the needed information about caller and callee functions along with the associated \textsc{id}s of related call instructions, and (ii) the reported S_R value. Despite the low probability of having collisions (as we discuss later), the S_R value is expected to be seen in more than one part of the program. Thus, the SAT solver can be configured to output all possible root-anchored paths that would lead to such a value.\footnote{The details of the constructed Boolean formulas and variables are described in Appendix A.} If eventually no path is found, the P_{rv} is regarded as compromised due to the illegitimate S_R value. If a path or more are found, the following checks are performed to further verify the execution integrity:

- For each path, V_rf checks the consistency of R_R, which should have limited values compared to S_R, as follows:
  - If the last assignment to R_R was made before a call to a function in a shared library, then the correct value of R_R should equal to the reported value of S_R (potentially XORed with intermediate \textsc{id}s if S_R is updated in a further different location).
  - If the last assignment to R_R was made due to an indirect call, then the R_R value would be the \textsc{id} of that call instruction or the result of XORing this \textsc{id} with the corresponding mask in the callee.

If the above checks fail for all paths, a control flow violation is likely the cause and thus the corresponding P_{rv} is considered compromised.

- After passing the aforementioned checks, V_rf proceeds to verify the consistency of S_R with the value of P_C for each validated path. To accomplish this, V_rf needs to access the Executable and Linking Format (ELF) file associated with the corresponding P_{rv}. This access allows V_rf to examine whether P_C points to an instruction within one of the functions that constitute the underlying path. This verification step can still be easily conducted despite the potential activation of Address Space Layout Randomization (ASLR) at runtime, which randomizes the base address but leaves the lower part of P_C unchanged. If at least one valid path is found, V_rf concludes that P_{rv} is benign.

5 Implementation

CFA+ prototype consists of three parts: (i) the compiler toolchain, (ii) the helper software modules on P_{rv}, and (iii) the V_rf framework. On the compiler side, CFA+ is implemented as a set of in-tree analysis and transformation passes that extend the LLVM 15.0 compiler framework. The analysis pass is implemented in the LLVM Intermediate representation (IR) layer at the Module level. The transformation pass is realized at the Machine Function level and implemented in the...
AARM64 backend. The LLVM Linker (LLD) is used to enable Link Time Optimization (LTO) \cite{70}, which is a requirement for CFA+. On the \texttt{Prv} side, \texttt{EH$_{CFA}$} is implemented as a kernel module on top of the ARM kernel v5.11, extending the default handler of BTI instructions. Furthermore, ATTEST$_{agent}$ is implemented as a user-space process that provides not only support for CFA+ but also for any RA scheme that depends on IMA \cite{86}, which is the de-facto standard for static attestation in the Linux kernel. To maintain further compatibility with IMA, we leverage the TPM as a RoT, where a differ-
cations like 401.bzip2 and 429.mcf, where indirect call instructions are minimal or non-existent. We note that OS-CFI focuses solely on forward-edge protection.

Overall, CFA+ demonstrates superior performance compared to many coarse- and fine-grained CFI schemes, while providing stronger security guarantees [14, 17, 18, 25, 59, 61]. However, it is essential to acknowledge that there is no one-size-fits-all solution, as specific application-dependent factors may favor other schemes for individual benchmarks.

Compile-time overhead. To assess the additional compilation time introduced by CFA+ passes, we employed the LLVM command-line option -time-passes. Table 2 presents the normalized compilation time (in seconds) for selected benchmarks, comparing runs with and without CFA+, and highlighting the analysis pass duration. On average, the compile-time overhead of CFA+ remains below 1.5%. Notably, the nbench-byte benchmark is excluded as it exhibits negligible overhead.

Binary Size Overhead. While many CFI and CFA schemes struggle to balance the trade-off between security and performance, the design of CFA+ prioritizes both features, albeit with increased binary size overhead. Table 3 demonstrates the size of the .text section, which contains all instructions, including CFA+ instrumentation instructions and hard-coded IDs. ELF binaries’ precise sizes (in bytes) were measured using the size command in the Linux kernel. The 456.hmmer benchmark exhibits the highest binary size overhead, amounting to 48.44%. However, in more complex applications like 403.gcc and 400.perlbench, this overhead is reduced by at least 10%, resulting in 38.69% and 34.28% respectively. The nbench-byte benchmark demonstrates the lowest overhead, with only 2.26% increase in binary size. The average binary size overhead across all benchmarks is approximately 22%.

The binary size overhead is influenced by various factors, with the ratio of call instructions to the total number of instructions being particularly significant, along with their distribution across functions. Table 4 provides detailed statistics on the selected benchmarks, showing that the 456.hmmer benchmark exhibits the highest ratio of call instructions at 9% of the total instruction count. This indicates the addition of at least 5 extra 4-byte instructions as pre- and post-call instrumentation, along with 8 bytes as a hard-coded unique ID for each call instruction. While this explains the significant binary size overhead observed in certain applications, it is worth noting that, by excluding certain outlier cases, e.g., the 456.hmmer benchmark, the average binary size overhead of CFA+ remains comparable to other forward-edge CFI schemes, such as LLVM-CFI (up to 23.21% code increase) and FineIBT (up to 19.05% code increase) [13].

Verification speed. The speed of the verification process is primarily determined by the performance of the SAT solver module, which has a dominant impact. To evaluate it, Vrf was configured to attest each benchmark 50 times during execution. The reported register values were then offloaded to the SAT solver for path(s) extraction, which ran on a laptop equipped with an Intel Core i7-1185G7 3GHz CPU, and 16GB of DDR4 RAM, running Ubuntu 20.04.5. Figure 8 depicts the minimum, average, and maximum time (in seconds)
While most applications took only fractions of a second, complex applications like 403.gcc took up to 80 seconds to find the matching path. On average, the verification process was required to find the first path confirming the reported values. While most applications took only fractions of a second, complex applications like 403.gcc took up to 80 seconds to find the matching path. On average, the verification process was completed in less than half a minute in most cases.

6.2 Security Evaluation

The security of CFA+ relies on five pillars:

- **Transparency.** CFA+ is inherently transparent as it does not rely on any confidential data, rendering it immune to a wide range of attacks, including certain side-channels.

- **Register-level atomicity.** CFA+ is not susceptible to race conditions in multi-threaded applications, where the intermediate values of the inline reference monitors can be manipulated while residing in memory. The values of the inline reference monitors of CFA+ are protected by design as they never leave the SR and RR registers, which are not shared among threads.

- **Strong collision resistance.** The entropy source to maintain collision-free SR values based on the generated IDs is 64 bits, which is big enough to achieve this property. For small and mid-sized applications, all potential values of SR can be checked during compile-time, where custom IDs can be regenerated in case of detecting a collision. This check is hard to perform for big and complex applications. Nevertheless, we note that the biggest application, i.e., 403.gcc, we encountered has around 2^50 unique paths. This means that the probability of having collisions is (2^50)^(-1) ≈ 3%, which is low and unlikely to yield sufficient gadgets to launch an attack that complies with BTI instructions and eventually bypasses CFA+.

- **Resilience.** The hybrid design of CFA+ renders it highly resilient against sophisticated attacks. For instance, consider the scenario where SR is XORed (encoded) with a call’s ID before its execution to invoke an intermediate function. Within this function, the return address is masked and saved on the stack. Tampering with this address, such as overwriting it, would result in one of two cases upon unmasking: (i) targeting a non-landing pad location and thus immediately triggering EH_CFA+ or (ii) leading to an arbitrary landing pad with a different ID, resulting in an inconsistent SR value. This inconsistency can lead to either (i) quick detection and prevention of the attack if the SR value is used to unmask previously masked return addresses along the execution path, or (ii) eventual detection by Vrf during attestation and verification. Vrf can determine that either (i) no legitimate path could have had the reported SR value, or (ii) there are inconsistencies between SR and PC, indicating that SR appears in the wrong path. A similar approach is employed during the verification process to detect control flow bending attacks [47].

- **Shielded execution.** As discussed in Section 4.4.4, CFA+ does not allow vulnerable shared code to return to a different location as this would be immediately detected and prevented due to mismatch values between SR and RR. This shields native code from exploitable vulnerabilities in shared code.

### 6.2.1 Experimental Security Evaluation

**Limited short gadgets.** The incorporation of BTI in CFA+ design significantly reduces the number of short gadgets, which are the most common ones in real-world attacks [31]. This further complicates any adversarial attempts to exploit potential collisions in complex applications. For instance, we leveraged capstone 5.0 [10] and Ropper 1.13.8 [84] to filter
out all potentially useful ROP and JOP gadgets that are up to 6-instruction long in the SPEC CPU2006 suite. We did not record any gadget in CFA+ enabled applications compared to their unprotected versions. Table 5 shows the impact of CFA+ on the number of available gadgets.

### Real World Exploits

Although it is hard to reproduce CVEs, especially for the AArch64 architecture, we managed to reproduce CVE-2013-2028. It is a stack-based buffer overflow vulnerability (triggered via an integer underflow), which affects nginx 1.3.9, allowing for a ROP attack that causes a denial of service and arbitrary code execution. We were able to reproduce this CVE on the AArch64 architecture and illegally execute the `execve` system call in the absence of CFA+. Although the entire nginx binary did not have the encoding of the syscall SVC #0 instruction, i.e., 0x010000d4, we directed the `ngx_execute_proc` function to perform `execve` on our behalf as part of the exploit. When running the recompiled version of nginx with CFA+ enabled, this attack is immediately prevented, due to attempting to jump to a corrupted address that when unmasked, resulted in an invalid target. We further performed a deliberate follow-up attestation that concluded the compromise of the corresponding `Prv` after observing the related record in `CF_report`.

### 6.3 Compatibility

**Functional correctness** CFA+ successfully maintains the functionality of all the benchmarks under consideration, as it does not cause any execution failures or unexpected results that would prevent performance measurements from being reported by the evaluation frameworks. Additionally, the borrowed type propagation analysis mechanism from Typro [35] ensures compatibility with potential valid targets for indirect calls. This stands in contrast to other approaches, such as LLVM-CFI [17] and ReCFA [63], which are incompatible with at least one application in the SPEC CPU2006 suite.

**Impact of instrumented shared libraries on legacy code.** Considering that the evaluation frameworks we employed utilize static linking for objective performance measurement reporting, we aimed to investigate the impact of instrumented shared libraries on legacy code in terms of compatibility and runtime overhead. To achieve this, we made modifications to the evaluation scripts of the SPEC CPU2006 benchmark suite, enabling dynamic linking against a custom GLIBC that we pre-compiled with instrumentation, as described in Section 4.4.4. By comparing the runtime of each application using this modified dynamic linking configuration to the default static linking configuration, we observed a negligible overhead, averaging at 0.03%. These results not only confirm the absence of compatibility issues but also demonstrate the functional correctness of our approach.

### 7 Discussion

**Precision of CFA+**. CFA+ exhibits a precision that is on par with other CFA schemes [21, 63], despite generating a CFG at the function-level and primarily instrumenting call instructions. The need for generating a CFG at a lower level, e.g., the basic-block level, only arises when indirect jump instructions are a concern, which is not the case for CFA+. As discussed in Section 4.4.1, CFA+ adopts a recommended practice of disabling the generation of jump tables, where most indirect jump instructions typically reside [87]. This proactive measure reduces the number of JOP gadgets and provides protection against specific microarchitectural attacks [20].

Nevertheless, even in rare scenarios where the compiler emits indirect jump instructions for optimization purposes, the number of such instructions and their legitimate targets remain limited. The CFA+ compiler toolchain will then ensure the generation of relevant landing pads, particularly `bt i j`, to handle these cases appropriately. Based on the results presented in Table 5, these instructions are unlikely to form useful gadgets. Furthermore, they cannot target other valid landing pads designed for `ret` instructions (i.e., the `call`-preceded ones), as triggering such landing pads means executing subsequent instrumentation instructions that introduce arbitrary updates to `sr`. This inconsistency would have a cascading effect on the entire invocation chain, violating its integrity. This violation would eventually be detected, as elaborated in Section 6.2.

**CFA+ vs other defenses.** As discussed in Section 3, while the current CFA and CFI schemes vary in their designs and security guarantees, CFA+ stands out as the first solution to combine the advantages offered by both approaches. Table 6 and Table 7 compare CFA+ with the most relevant CFA and CFI techniques respectively. In addition to its distinguishing prevention capabilities, Table 6 shows that CFA+ is the only attestation scheme that handles complex software stacks with minimal runtime and network overhead. Furthermore, it is the only scheme that performs attestation at scale, where the status of many `Prv` applications is reported in one compact report and verified smoothly. Please note that the CFA schemes reported in the upper part of Table 6 are included for illustrative purposes and are not directly comparable to CFA+ as they primarily focus on embedded software. Table 7 highlights CFA+’s superiority over relevant CFI schemes, as it is the only one that covers forward and backward edges, while providing trustworthy evidence of runtime integrity. We note that FineIBT [13] introduces an optimized

### Table 5: Number of potential gadgets w.r.t. CFA+

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of ROP gadgets</th>
<th>Number of JOP gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Up to 6 instructions long)</td>
<td>(Up to 6 instructions long)</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>932</td>
<td>461</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>403.gcc</td>
<td>1791</td>
<td>4546</td>
</tr>
<tr>
<td>429.mcf</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>454.gobmk</td>
<td>47</td>
<td>548</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>86</td>
<td>116</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>37</td>
<td>6</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>193</td>
<td>113</td>
</tr>
</tbody>
</table>

---

**USENIX Association**

33rd USENIX Security Symposium 6645
CFA+ proposes a lighter-weight alternative for signing and verifying return addresses using \texttt{XOR} instructions. Nevertheless, the current design and implementation of CFA+ are fully compatible with PA. When the \texttt{-mbranch-protection=pac-ret} flag is added to the compiler pipeline, the transformation pass of CFA+ relies on it for return address integrity. As a result, CFA+ emits fewer instrumentation instructions accordingly.

Based on experimental evaluations, incorporating PA as a primary component in the CFA+ design offers an average reduction of approximately 2% in binary size overhead. However, we did not consider so for two key reasons. First, PA instructions introduce additional runtime overhead. Reports indicate that executing 7 \texttt{XOR} instructions can be 0.15% faster than executing 1 PA instruction [40]. Considering that signing and verifying a ret instruction requires at least two PA instructions, the worst-case scenario of CFA+ would be faster with 7 additional non-cryptographic instructions (2 pre-call, 1 bitc c, 4 post-call) executed within a call-ret edge. Furthermore, recent research concludes that signing ret instructions with PA could result in an average runtime overhead of 3% [24], with extreme outlier cases reaching a runtime overhead of 17% [77]. As previously discussed, CFA+ could effectively protect both backward and forward edges with a comparable average runtime overhead. Second, PA relies on confidential data, such as secrets, which exposes it to a wider range of attack surfaces including various side channels [9, 30, 64]. In contrast, the current design of CFA+ is transparent, i.e., secret-independent, resulting in a smaller attack surface.

### Table 6: CFA+ vs relevant CFA mechanisms.

<table>
<thead>
<tr>
<th>CFA Scheme</th>
<th>Target</th>
<th>Objective</th>
<th>RoT</th>
<th>Scalability</th>
<th>Runtime overhead</th>
<th>memory overhead</th>
<th>Network overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-FLAT [56]</td>
<td>✗</td>
<td>RD</td>
<td>TZ</td>
<td>✗</td>
<td>☀</td>
<td>●</td>
<td>⪫</td>
</tr>
<tr>
<td>OAT [65]</td>
<td>✗</td>
<td>RD</td>
<td>TZ</td>
<td>✗</td>
<td>●</td>
<td>●</td>
<td>★</td>
</tr>
<tr>
<td>ARH [29]</td>
<td>✗</td>
<td>RD</td>
<td>TZ</td>
<td>✗</td>
<td>●</td>
<td>●</td>
<td>&gt;&gt;&gt;</td>
</tr>
<tr>
<td>ISC-FLAT [16]</td>
<td>✗</td>
<td>RD</td>
<td>TZ</td>
<td>✗</td>
<td>⬤</td>
<td>●</td>
<td>★</td>
</tr>
<tr>
<td>BLAST [62]</td>
<td>✗</td>
<td>RD</td>
<td>TZ</td>
<td>✗</td>
<td>●</td>
<td>●</td>
<td>★</td>
</tr>
</tbody>
</table>

#### Legend:
- ✗ Embedded SW
- ● Complex user-space SW
- RD: Remote Detection
- LD: Local Detection
- TZ: ARM TrustZone
- MPK: Intel Memory Protection Keys
- TPM: Trusted Platform Module
- X: Not applicable
- ☀ Has this feature
- ● ● Low overhead (runtime: ≤ 5% memory: ≤ 10%)
- ● Moderate overhead (runtime: between 5% and 10%, memory: between 10% and 30%)
- ● High overhead (runtime: > 10%, memory: > 30%)
- X Not reported
- ☀ (Potentially) High network overhead
- ♦ Moderate network overhead
- ☀ Low network overhead

### Table 7: CFA+ vs context-sensitive CFI mechanisms.

<table>
<thead>
<tr>
<th>CFI Scheme</th>
<th>Target</th>
<th>Coverage</th>
<th>Objective</th>
<th>Trust. Evidence</th>
<th>HW assist</th>
<th>Runtime overhead</th>
<th>memory overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrioPat [51]</td>
<td>⬤</td>
<td>LD</td>
<td>✓</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>eCFI [10]</td>
<td>⬤</td>
<td>LD</td>
<td>✓</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OS-CFI [41]</td>
<td>⬤</td>
<td>LD</td>
<td>✓</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>CF-CFI [16]</td>
<td>⬤</td>
<td>LD</td>
<td>✓</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>VFI-CFI [59]</td>
<td>⬤</td>
<td>LD</td>
<td>✓</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>TCFI [18]</td>
<td>⬤</td>
<td>LD</td>
<td>✓</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>CFA+</td>
<td>⬤</td>
<td>LD+RD</td>
<td>✓</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

#### Legend:
- ⬤ Embedded SW
- LD: Local Detection
- RD: Remote Detection
- LD+RD: Local+Remote Detection
- LX: Local Protection
- X: Not applicable
- ☀ Has this feature
- ● ● Low overhead (runtime: ≤ 5% memory: ≤ 10%)
- ● Moderate overhead (runtime: between 5% and 10%, memory: between 10% and 30%)
- ● High overhead (runtime: > 10%, memory: > 30%)
- X Not reported
- ☀ (Potentially) High network overhead
- ♦ Moderate network overhead
- ☀ Low network overhead

and hardware-assisted version of LLVM-CFI [17], leveraging Indirect Branch Tracking (IBT), which is Intel counterpart of ARM BTI [88]. IBT is part of Intel’s control flow Enforcement Technology (CET) where hardware-assisted shadow stack is available as well [88]. Therefore, the contribution of FineIBT is limited to protecting indirect forward edges, without providing any evidence of runtime integrity.

**Compatibility with RA.** CFA+ prioritizes seamless integration with static RA approaches. It not only utilizes and shares the same RoT withIMA [86] but also follows a similar design for maintaining attestation reports. In particular, likewise IMA, CFA+ maintains a unified log file for all events, with its hash value stored in a designated PCR register within the TPM. When required, ∀rf can obtain the two signed log files in one attestation request.

**Integration with Pointer Authentication (PA).** PA is a hardware security feature introduced in the Armv8.3 architecture, which aims at ensuring pointers integrity with cryptographic primitives [7]. To achieve this, new instructions are added to sign and verify pointers. The computed cryptographic hash, known as Pointer Authentication Code (PAC), is stored in the unused upper bits of 64-bit pointers. Mainstream compilers like LLVM/Clang and GCC now include support for signing return addresses using PA by adding the \texttt{-mbranch-protection=pac-ret} flag.
underlying assumptions of the proposed defenses may only be applicable to simple specialized embedded software, which C-FLAT and OAT, among others, target. For instance, C-FLAT relies on recording all executed inputs and including them in the attestation report for verification, whereas OAT requires manual efforts to annotate critical data variables. These assumptions may not hold for general-purpose complex software, which CFA+ aims to address. Nonetheless, CFA+’s threat model encompasses control flow bending attacks, which is the relevant generalization of non-control-data attacks that manipulate control data [47]. As previously discussed, the detection of these attacks relies only on identifying inconsistencies between SR and PC values. We note that CFA+ has the potential to leverage other architectural features to cover non-control-data attacks. For instance, it could utilize PA to enforce data-flow integrity, as proposed by RSTI [41]. Further exploration of these possibilities is left for future research.

Limitations of CFA+. One limitation of CFA+ is its reliance on preserving two registers, which can pose compatibility issues for software that utilizes inline assembly and relies on these registers. For these cases, CFA+ would terminate the compilation process, indicating the potential compatibility issue. Additionally, the binary size overhead associated with CFA+ may not be acceptable in certain scenarios or applications. To address this limitation, CFA+ could benefit from several compiler optimization techniques that have not been considered so far, including static pruning of instrumentation points, redundant checks eliminations, and instrumentation hoisting (especially for provably safe call instructions).

8 Conclusion

This paper presented CFA+, a novel hybrid runtime defense approach that combines the strengths of Control Flow Integrity (CFI) and Control Flow Attestation (CFA) schemes to provide both local and remote detection capabilities in an efficient design. By leveraging the Branch Target Identification (BTI) feature of ARMv8.5 and selective software instrumentation, CFA+ not only enforces a local policy to mitigate control flow hijacking attacks but also enables lightweight always-on monitoring of the execution state without the need for maintaining in-memory control flow logs. To facilitate trustworthy verification, CFA+ encapsulates relevant runtime state information in dedicated registers that can be securely obtained by an external verifier. Furthermore, CFA+ features an efficient design for the verification process of attestation reports, allowing for easy detection of control flow violations. Evaluation results demonstrate that CFA+ achieves high efficiency and scalability, effectively balancing strong security guarantees with performance advantages.

Acknowledgments

The authors thank the anonymous shepherd and reviewers for their constructive comments and feedback. They also thank Gleb Ingman for implementing part of the verifier module during his internship at Huawei.

References


[82] Red Hat. Hardening ELF binaries using Relocation Read-Only (RELRO).

A SAT-based Verification Procedure

As described in Section 4.5, the Vrf turns the verification process into a Boolean satisfiability problem, which can be solved by any efficient SAT solver. To this end, the SAT solver would need to determine whether there exists an assignment of truth values to variables in a given logical formula so that such a formula evaluates to True. In the context of CFA+, the SAT problem involves determining whether a valid assignment of truth values to the variables exists in the corresponding CFG, such that the set of connected edges identified in the CFG would correspond to the reported $SR$ value. To solve this, we only need one Boolean variable to mark the taken edges (1 means taken, 0 means not taken).

More formally, a Boolean variable $e_{used}$ is created for each edge $e$, which is set to 1 if the corresponding edge $e$ is selected. The solver takes the function-level CFG and the reported $SR$ value as inputs and produces a labeled CFG with 0s and 1s. In this labeled CFG, the edges labeled with 1s lead to the reported $SR$ value. Two constraints must be satisfied when assigning these values:

• Connectivity: the selected edges should form a path, i.e., a spanning tree.
• Equity: the XOR value of the ID of these edges should equal to the input $SR$, which is formalized as

\[
\text{XOR}(\{ e_{\text{id}} \mid e \in E, e_{\text{used}} = 1 \}) = SR
\]

We note that the ID in $e_{\text{id}}$ is the ID of the related call instruction.

Assuming a collision-free CFG, where only one path would lead to the reported $SR$ value, the first constraint can be met by looking for the Eulerian path, where each edge should be visited only once, and every chosen edge should be reachable from the root vertex (root function). This means that for any vertex (function), except for the root and leaf vertices, the number of incoming edges that are selected should equal to the number of outgoing selected ones. This is formalized in the following formula:

\[
\forall v \in V \times \{ v_{\text{root}}, v_{\text{leaf}} \} : \left| \{ e \in E \mid e_{\text{used}}, e_{\text{caller}} = v \} \right| = \left| \{ e \in E \mid e_{\text{used}}, e_{\text{callee}} = v \} \right|
\]

The balance mentioned above is not mandatory for the root vertex, which should have an additional outgoing edge, as well as for leaf vertices, each of which should have an additional incoming edge. The following two formulas formalize these statements for the root and leaf vertices, respectively:

\[
\left| \{ e \in E \mid e_{\text{used}}, e_{\text{caller}} = v_{\text{root}} \} \right| = \left| \{ e \in E \mid e_{\text{used}}, e_{\text{callee}} = v_{\text{root}} \} \right| + 1
\]

\[
\left| \{ e \in E \mid e_{\text{used}}, e_{\text{callee}} = v_{\text{leaf}} \} \right| + 1 = \left| \{ e \in E \mid e_{\text{used}}, e_{\text{callee}} = v_{\text{leaf}} \} \right|
\]

The second constraint can be satisfied by formulating individual equations for each of the 64 bits in the ID. This approach allows for grouping the edges based on the XOR value of their respective bit, ensuring alignment with the corresponding counter bit in $SR$. The formalization of this approach is as follows:

\[
\forall \text{bit} \in \{ 1, 2, 3, \ldots, 64 \} : \text{XOR}(\{ e_{\text{used}} \mid e \in E, e_{\text{id}}^{\text{bit}} = 1 \}) = SR^{\text{bit}}
\]

where $x^i$ denotes $i$-th bit of $x$, and $x$ refers to either $e$ or $SR$.

If the SAT solver successfully finds a solution, the path can be simply recovered using a Depth-First Search (DFS) algorithm.

To address potential collisions and extract multiple paths, additional measures can be taken in the SAT formulation. One approach is to introduce duplicated edges in the CFG to extract more distinct paths. Another approach is to introduce several boolean variables. These variables serve the same goal but target different choices within the CFG, allowing for the exploration of alternative paths. It is important to ensure that the assignment of these variables is not exactly the same across all paths, as this would result in identical paths being extracted.