# SoK: Integrity, Attestation, and Auditing of Program Execution

Mahmoud Ammar
*Independent Researcher*
*mail@mahmoud-ammar.org*

Adam Caulfield
*Rochester Institute of Technology*
*ac7717@rit.edu*

Ivan De Oliveira Nunes
*Rochester Institute of Technology*
*ivanoliv@mail.rit.edu*

***Abstract*—This paper provides a systematic exploration of Control Flow Integrity (CFI) and Control Flow Attestation (CFA) mechanisms, examining their differences and relationships. It addresses crucial questions about the goals, assumptions, features, and design spaces of CFI and CFA, including their potential coexistence on the same platform. Through a comprehensive review of existing defenses, this paper positions CFI and CFA within the broader landscape of runtime defenses, critically evaluating their strengths, limitations, and trade-offs. The findings emphasize the importance of further research to bridge the gaps in CFI and CFA and thus advance the field of runtime defenses.**

***Index Terms*—Control Flow Integrity, Control Flow Attestation, Software Security, System Security.**

## 1. Introduction

Unsafe programming languages like C and C++ are still prevalent, especially for lower-level system development [1]. Memory safety bugs, such as buffer overflows, are prominent enablers of attacks on programs written in such languages. Attacks that modify/inject code can be (to some extent) mitigated by existing defenses. Among them, Data Execution Prevention (DEP) and Write-Xor-eXecute (W⊕X) [2] policies can prevent user-space code injection attempts at runtime. Secure boot can locally enforce boot-time code integrity (including the integrity of privileged software, e.g., stage 1 and 2 boot-loaders and kernel) [3], [4]. Static (i.e., boot-time or load-time) Remote Attestation (RA) can further convince a remote party of the integrity of the booted code chain [5], [6].

On the other hand, code-reuse attacks [7] (exemplified by Return Oriented Programming – ROP [8] – and Jump Oriented Programming – JOP [9]) can pose significant threats without modifying the installed code, even in the presence of existing defenses. They instead exploit memory corruption vulnerabilities to trigger out-of-order execution of sub-sequences of instructions (known as gadgets) within a program. This can result in unintended behavior even when code modifications are prevented. As illustrated in Figure 1, code-reuse attacks can be broadly classified into two categories: control flow hijacking and data-only attacks. The former directly corrupts memory storing code pointers, e.g., return addresses [8] and function pointers [9] during execution. The latter changes control flow related data, e.g., loop/conditional variables or counters, without causing control flow transfers that do not exist in the Control Flow Graph (CFG) of the target program [10], [11].

Much attention has been devoted to code-reuse attack mitigations due to their popularity and effectiveness [12] with several protection and detection mechanisms proposed in the past few decades [13]. Most notably, Control Flow Integrity (CFI) mechanisms for both forward and backward edge protection have been widely recognized as key mitigations [14], [15]. Ergo, recent years have seen efforts to adopt both academic and industry proposals, each with their own sets of trade-offs [15]–[17]. Nevertheless, only a few of these proposals, e.g., LLVM CFI [18], have become available in production compilers, despite known limitations in terms of granularity and compatibility [17], [19].

On the hardware side, both ARM and Intel have equipped their latest-generation architectures with new extensions to assist control flow attack mitigations. Examples include Pointer Authentication (PA), Memory Tagging Extension (MTE), and Branch Target Identification (BTI) features from ARM [20], and the Control Flow Enforcement Technology (CET) from Intel [21]. While various contemporary CFI approaches leverage these extensions in their designs [22]–[29], gaps still persist [30]–[32].

In a parallel line of efforts, Control Flow Attestation (CFA) [33]–[43] has been proposed to enable remote Verifier(s) ($\mathcal{V}$rf) to ascertain the execution integrity (including the absence of control flow attacks/violations) of an operation of interest performed by a remote device (called a prover or $\mathcal{P}$rv). In its ideal form, CFA generates an authenticated log containing all dynamically defined control transfers occurring during the execution of an attested operation of interest[1]. Nonetheless, similar to CFI, coarser-grained CFA approaches are also possible [40], establishing trade-offs between the completeness of CFA evidence and performance, especially when overheads related to storage and transmission of said evidence to $\mathcal{V}$rf are a concern (e.g., when $\mathcal{P}$rv is a resource-constrained embedded platform).

Notably, not all CFA techniques guarantee that CFA evidence is received by $\mathcal{V}$rf. While this is sufficient for

1. Note that some CFA variations aim at enabling continuous verification of all control flow transfers on $\mathcal{P}$rv, rather than focusing on individual operations of interest. For more details, see Section 2.3.

attestation, wherein a $\mathcal{V}$rf would not trust responses/values received from $\mathcal{P}$rv unless accompanied by CFA evidence, it does not support secure runtime auditing [41], [42]. The latter aims to ensure that CFA evidence always reaches $\mathcal{V}$rf, even if $\mathcal{P}$rv is compromised, allowing for attack root cause analysis and appropriate remediation.

CFI and CFA goals can be viewed as runtime analogs of boot-time code integrity guarantees offered by secure boot vs. static RA. While CFI enables *in loco* detection of control flow violations (typically triggering exceptions when detected), CFA provides remotely verifiable (unforgeable) evidence of the control flow path followed by an operation of interest executed by a $\mathcal{P}$rv device, thus enabling control flow path analysis by a remote $\mathcal{V}$rf.

### 1.1. Motivation & Intended Contributions

Although CFI and CFA approaches exist due to the common threat of control flow attacks, their different goals, designs, and capabilities are not yet systematically discussed in the literature. Naturally, the current lack of systematization prompts questions such as:

**[Q1]** How do CFA and CFI goals differ?

**[Q2]** What are the assumptions, features, and design spaces of CFI vs. CFA, as well as their similarities and differences?

**[Q3]** What makes CFA different from remotely attesting adherence to a CFI policy? Could CFA uncover attacks that CFI would not (and vice-versa)?

**[Q4]** Could CFI and CFA coexist on the same platform?

Additionally, there is often confusion surrounding the terminology in the context of control flow-related mechanisms (e.g., prevention vs. local detection vs. remote detection; runtime attestation vs. runtime auditing; fine-grained vs. coarse-grained approaches; etc.) and their relationship to memory safety and compartmentalization defenses. This ambiguity makes it challenging to precisely understand the guarantees provided by each approach. Therefore, it becomes crucial to delve into such nuances to clearly grasp the benefits of each approach and their roles within the broader landscape of runtime software defenses.

In this paper, we explore the relationships and differences between CFI and CFA by systematically examining the fundamental goals and trade-offs associated with both approaches. Towards this goal, we present a systematic review of existing runtime defenses to provide context and position CFI and CFA within the broader landscape of execution integrity defenses. Subsequently, we classify recent work in CFI and CFA according to design choices, weighing their advantages and disadvantages and aiming to grasp a better understanding of existing limitations. Finally, we discuss missing links between CFI and CFA and future research avenues.

### 1.2. Literature Selection Criteria

The selection criteria for the inclusion of academic or industrial proposals in our systematization are as follows:

- We aim to include all available literature on CFA due to the manageable number of existing proposals (except unintended oversights).
- Given the extensive volume of CFI proposals, we use the following criteria for selection within the past 10 years:
  - Papers published in prestigious security-focused conferences such as USENIX Security, IEEE S&P, CCS, and NDSS.
  - Papers with more than 100 citations, indicating their broad influence in subsequent work.
  - Papers or proposals adopted in mainstream compilers or hardware architectures.

### 1.3. Scope & Related Systematizations

Memory safety [44]–[47] approaches aim to eliminate or reduce vulnerabilities that could lead to control/data flow attacks and data corruption during software development, i.e., before deployment. These typically work in two ways. First, memory safety can be a built-in security feature of programming languages such as Go and Rust. Rust [48], for instance, utilizes static compile-time analysis to optimize safety checks and memory management decisions, such as bounds check elimination, while incorporating mechanisms (e.g., value ownership and borrowing) to ensure temporal safety. Second, memory safety can be obtained as memory-safe dialects of memory-safe programming languages. An example of this is Checked C [49], which augments C with spatial memory safety checks introduced at compilation time and/or runtime. This involves refining the C type system with safe pointer and array types with stricter usage models. Even so, this approach provides partial protection and presents compatibility challenges with legacy software.

A third category of techniques focuses on fortifying and isolating code through runtime checks [50]. These techniques include compartmentalization [51], software fault isolation [52], memory layout randomization [53], as well as CFI. While the primary goal of this class is to detect and isolate runtime violations, some literature still categorizes these methods as memory safety techniques. There is, however, no clear consensus on whether the term "memory safety" should be limited to the first two categories or expanded to include this third one (and possibly others). This lack of agreement has led to confusion over the scope of the term [44].

Terminology aside, our work focuses on systematizing and discussing the relationship between runtime integrity enforcement [14] and runtime attestation [33] methods used after software deployment (hence "runtime"). This is complemented by existing systematizations focused, for instance, on memory safety or compartmentalization. Related to our work, Szekeres et al. [13] provide a general model of memory corruption attacks, which serves as a foundation for identifying the different policies that can prevent such attacks. Song et al. [54] offer a systematic overview of sanitizers, emphasizing their role in uncovering security vulnerabilities. Larsen et al. [55] present a comprehensive and unified overview of software diversification approaches,

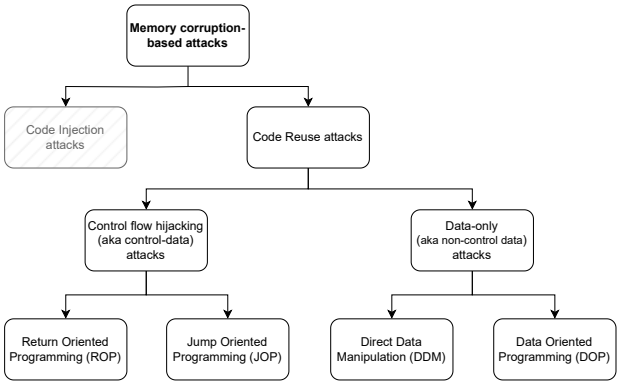


Figure 1. Classes of memory corruption-based attacks to software integrity.

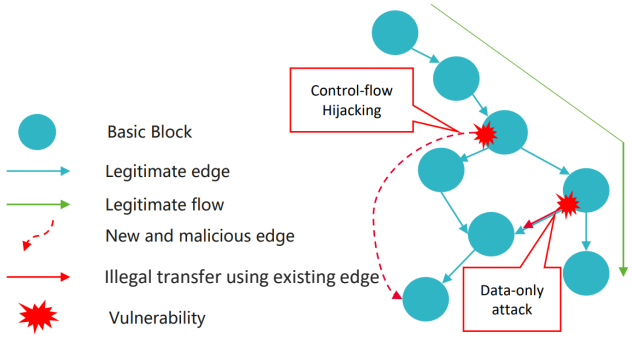


Figure 2. Control flow hijacking vs. Data-only attacks on a CFG.

highlighting their inherent trade-offs. Burow et al. [56] conduct a thorough evaluation of the design space of shadow stacks, considering performance, compatibility, and security aspects. Contrary to the aforementioned efforts, this SoK focuses on shedding light and evaluating recent CFI and CFA methods as well as their relationship and differences.

## 2. A Lightning Tour

This section reviews code-reuse attacks and existing defenses, highlighting the role of CFI and CFA in this landscape.

### 2.1. Code Reuse Attack Background

Figure 1 illustrates the general classes of memory corruption-based attacks. Code-reuse attacks are further classified into control flow hijacking and data-only attacks. At a high level, their difference lies in the former performing control flow transfers that do not exist in the legitimate CFG of the target program and the latter causing unintended transfers via edges that exist in the CFG. The two cases are depicted in Figure 2. Return Oriented Programming (ROP) [8] and Jump Oriented Programming (JOP) [9] are the two main categories of control flow hijacking attacks. Both ROP and JOP stitch out-of-order sub-sequences of instructions, so-called *gadgets*, to modify the control flow path of the target program to perform a malicious action. As their names indicate, ROP corrupts backward edges, targeting gadgets that end with `return` instructions. JOP corrupts forward edges, targeting gadgets that end with indirect `jump` or `call` instructions.

Data-only attacks can be classified into Direct Data Manipulation (DDM) and Data Oriented Programming (DOP) based on the type of non-control data being manipulated [57]. DDM attacks can be as simple as illegally modifying the value of a variable [10]. DOP attacks [11], on the other hand, aim to perform expressive (often Turing-complete) computations by chaining carefully selected DOP gadgets, ensuring that the gadget chain forms a valid path within the CFG. This is typically achieved by corrupting non-control data, such as variables that define paths in conditional statements and loop counters.

### 2.2. Runtime Defenses

Figure 3 illustrates the relationship between memory vulnerabilities, runtime exploits, and associated defenses. The primary categories of runtime defenses against memory corruption-based attacks are illustrated in (❷ - ❺), which have been adapted from [13] and [50] respectively. Software testing tools, such as sanitizers [54] and fuzzers [58], act as a front-line defense in the pre-deployment phase, where the main goal is to find as many vulnerabilities as possible and fix them. Boot- and Load-time software verification mechanisms, such as Secure Boot [59], Measured Boot [60], and Load-time attestation (e.g., the Linux Integrity Measurement Architecture (IMA) for user-space software [61]), are deployed as a primary shield in the post-deployment phase to prevent booting/loading of non-authentic software. However, the presence of memory corruption vulnerabilities at runtime remains a concern after this stage. Therefore, several runtime defenses have been proposed, each targeting specific steps of the attack process. Considering the five distinct attack steps (❶ - ❺) outlined in the general model of memory corruption attacks from [13], Figure 3 illustrates which class of defenses can counter each type of exploit and at which step. As information leaks are not integrity violations, they are not considered in Figure 1. In the following, we summarize the individual attack steps and relevant defenses:

❶ **Memory Vulnerability:** Finding and exploiting a memory corruption vulnerability is an essential requirement for any of the runtime attacks considered in Figure 3. Illegal access to a memory address, whether to read, write, or both, depends on the particular vulnerability. We note that vulnerabilities that enable read-only access are (by themselves) not sufficient to corrupt the execution integrity of the target program.

❷ **Integrity Violation:** Exploiting vulnerabilities that grant illegal write access enables adversaries to tamper with the various aspects of a program, including the (i) program's code (instructions in memory), (ii) control data (e.g., `return` addresses and function pointers), and (iii) non-control data (e.g., data variables and pointers). Isola-

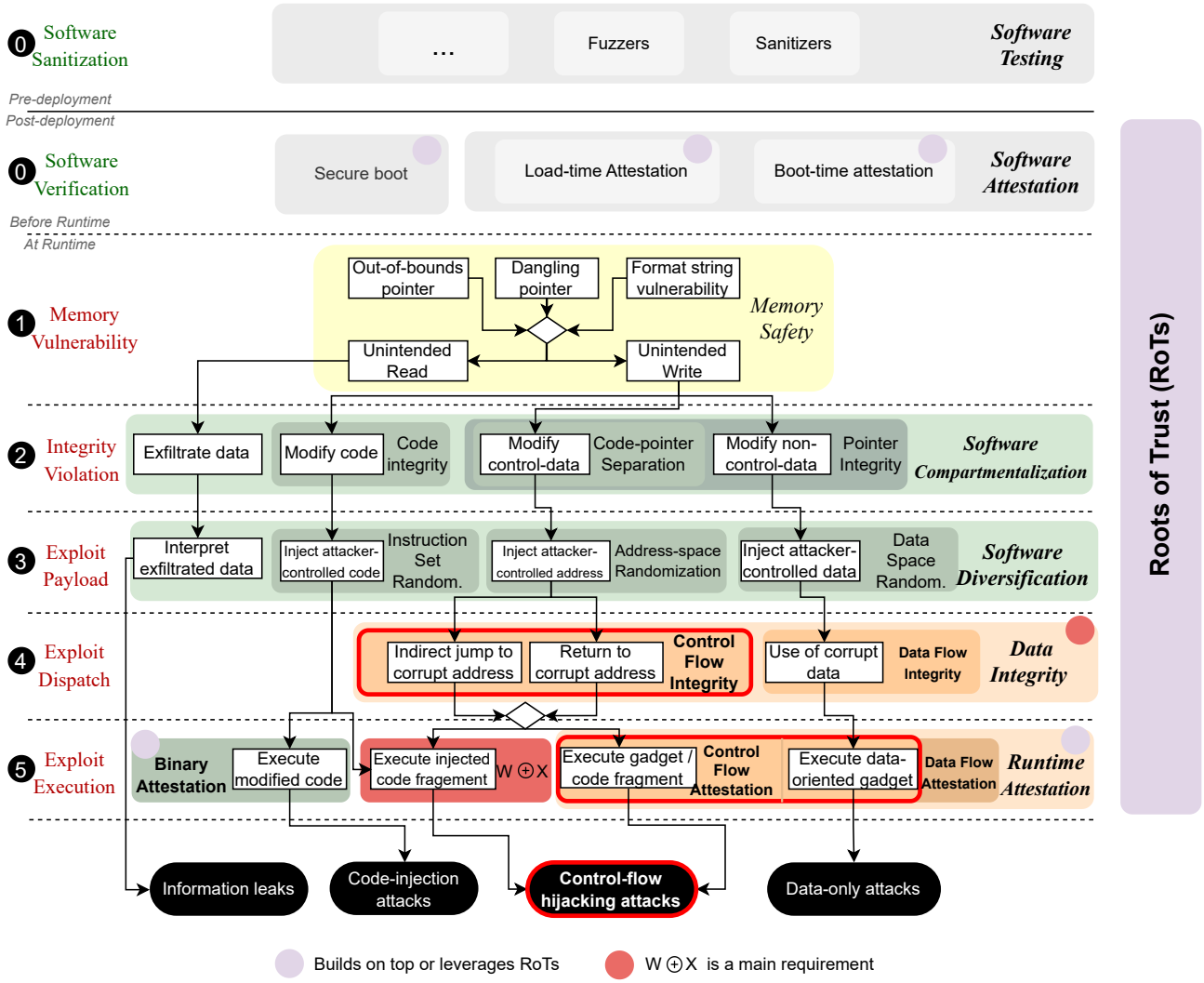


Figure 3. A high-level overview of defenses against memory corruption-based attacks with a focus on runtime defenses (expanded based on [13] and [50]).

tion and compartmentalization mechanisms play a crucial role in enforcing access control permissions to mitigate integrity violations. These mechanisms restrict the targets that adversaries can access, often thwarting attacks at an early stage or preventing their spread to the rest of the system. For instance, access control mechanisms like the AArch64 (Un)Privileged Execution Never feature [62] make it significantly harder to directly corrupt program code [63]. Code Pointer Integrity (CPI) [64] is a security mechanism that safeguards all code pointers and data pointers pointing to code by storing them in an isolated memory area. Code Pointer Separation (CPS) [64], a variant of CPI, isolates only code pointers while leaving the protection of data pointers to other measures for performance reasons. Software Fault Isolation (SFI) [52], memory tagging [65], and capability-based architectures (exemplified by CHERI [51]) operate at various granularity levels to isolate larger software components into distinct protection domains. These mechanisms limit the consequences of attacks that exploit memory vulnerabilities by confining them within specific compartments.

❸ **Exploit Payload:** If previous defenses are bypassed, the adversary can inject payloads to manipulate the data and control flows of the target program. In general, the payload injection process requires knowledge of the program's memory layout. In response, software diversification aims to impede the crafting of reusable exploits by introducing uncertainty through randomization. For instance, Address Space Layout Randomization (ASLR) [53] and Instruction Set Randomization (ISR) [66] are lightweight defenses that randomize memory layouts, making payload injection more challenging for control flow hijacking and code-injection attacks. Additionally, Data Space Randomization (DSR) [67] can complicate data-only attacks. While these techniques offer probabilistic guarantees, they significantly raise the difficulty of runtime attacks.

❹ **Exploit Dispatch:** To successfully launch sophisticated attacks, the adversary needs to divert the target program to operate on the injected payload. This step is crucial for expressive code-reuse attacks such as ROP, where the attack is initiated by manipulating the stack pointer to execute a sequence of selected gadgets in a predetermined

order, with each gadget returning to a specific memory address in the following gadget to implement the desired attack behavior. Control Flow Integrity (CFI) [14] and Data Flow Integrity (DFI) [68] are two commonly used defenses to ideally detect and block control flow hijacking and data-only attacks at this stage. These techniques involve implementing and enforcing policies that must be followed during program execution. However, contemporary literature shows that maintaining gap-free policies is inherently challenging, leaving potential exploit opportunities [19], [69], [70].

❺ **Exploit Execution:** As discussed above, ensuring the complete integrity of a victim program can be challenging. As a result, runtime attestation mechanisms have been proposed as a last line of defense to enable remote verification of code and execution integrity in a trustworthy manner. These mechanisms aim to detect tampering with code or violations of control/data flow. In addition, they also provide means to convince a remote party of the execution integrity of the target program during an operation of interest and enable, in some cases, auditing root cause vulnerabilities in case of exploits. In addition to measures such as $\mathbf{W} \oplus \mathbf{X}$ [71] and DEP [2] policies, which are deployed to prevent code-injection attacks, remote attestation approaches of code binary [6], [72]–[74] are widely regarded as essential for providing remotely verifiable evidence of binary integrity at runtime. At any time during execution, they can be used to attest that the code (including CFI/CFA instrumentation instructions) remains untampered. RA becomes paramount for most-privileged code (and single privilege systems, e.g. bare-metal micro-controllers) where full disablement of runtime code modifications implies the inability to perform remote software updates [75]. Control Flow Attestation (CFA) [33], [35] and Data Flow Attestation (DFA) [37], [38], [76] approaches have emerged to specifically detect and audit code-reuse attacks, enabling trustworthy remote verification of control and data flow integrity respectively.

As shown in Figure 3, attestation mechanisms build atop Roots of Trust (RoTs) as a foundation to provide trustworthy evidence of system/software state that can be remotely verified. For instance, RoTs are utilized in several key aspects of attestation mechanisms. They serve as a foundation for securely measuring system state and/or installed software (RoT for Measurement), securely storing attestation secret keys (RoT for Storage), and/or signing attestation reports (RoT for Reporting). Examples include Trusted Platform Modules (TPMs) [77], DICE [78], hardware extensions in Intel SGX-capable processors [79], ARM TrustZone-based RoTs [80], [81], and various academic proposals such as Keystone [82] and BYOTee [83], among others.

## 2.3. CFI & CFA: Definitions & Threat models

Considering their prominent status as actively researched defenses, the rest of the paper systematically explores CFI and CFA techniques, shedding light on their underlying principles, relationships, trade-offs, and other crucial aspects to provide insights into unclear considerations for adoption in real-world scenarios.

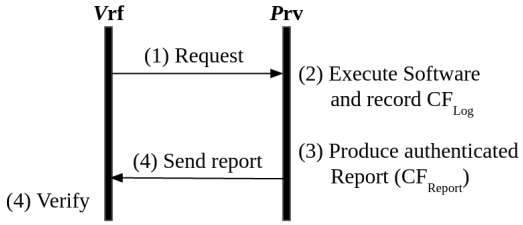


Figure 4. Typical CFA Interaction

**2.3.1. Control Flow Integrity (CFI).** Originally proposed by Abadi et al. [14], CFI is a policy-based mitigation against control flow hijacking attacks, restricting the execution path of a program at runtime based on a pre-computed CFG. In principle, enforcing CFI on a target program involves:

- Generation of an over-approximated CFG, denoted ≈CFG.
- Enforcement of control flow to comply with ≈CFG through Reference Monitors (RMs), which are software- or hardware-based runtime checks that verify the target of any indirect branch instruction at runtime.

≈CFG can be generated statically (as proposed originally [14]) or dynamically, as seen in following proposals [84], [85]. When ≈CFG ≡ CFG, it is generally difficult for an adversary to manipulate control flow and alter a program's intended behavior without detection by the activated or inserted RMs. However, statically determining strict CFGs for complex programs remains an open challenge [86], leading many practical approaches to over-approximate ≈CFG.

**2.3.2. Control Flow Attestation (CFA).** CFA focuses on producing unforgeable evidence of the control flow path followed by an executable on a prover device ($\mathcal{P}$rv). This evidence allows a remote verifier ($\mathcal{V}$rf) to assess the trustworthiness of execution and its outcomes. CFA is an (on-demand) challenge-response protocol, as shown in Figure 4.

A CFA instance starts with $\mathcal{V}$rf sending a request containing a cryptographic challenge to $\mathcal{P}$rv. Upon receiving the request, $\mathcal{P}$rv must execute the operation requested by $\mathcal{V}$rf (either specified implicitly or explicitly within the request). During the execution of the requested task, an RoT in $\mathcal{P}$rv must ensure that an authenticated log ($\text{CF}_{Log}$) containing a representation of the control flow path executed during the operation is built. After execution completes, the RoT computes an authenticated integrity measurement (e.g., using a Message Authentication Code (MAC) or signature) over the received challenge, $\text{CF}_{Log}$, and the executed binary to produce a response token ($\text{CF}_{Report}$). Finally, $\mathcal{P}$rv transmits $\text{CF}_{Report}$ to $\mathcal{V}$rf along with $\text{CF}_{Log}$. Given the need to securely store the secret used to authenticate $\text{CF}_{Report}$ even when $\mathcal{P}$rv is potentially compromised, RoT implementations typically involve some form of secure hardware support.

Upon receiving $\text{CF}_{Report}$, $\mathcal{V}$rf can use this evidence to determine if $\mathcal{P}$rv executed the expected software correctly through a valid control flow path. Further, when $\text{CF}_{Report}$ shows an invalid path, $\mathcal{V}$rf can analyze the anomalous evidence to determine its cause and potentially remediate it.

Existing CFA techniques (see Section 3) use either (1) binary instrumentation along with Trusted Execution Environment (TEE) support; or (2) custom hardware modifications to generate $CF_{Log}$ by detecting and saving each branch destination to a dedicated and protected memory region. For techniques that use binary instrumentation, a pre-processing phase modifies the binary so that branch instructions are prepended with additional calls to a TEE-protected trusted code. Once called, the trusted code appends to $CF_{Log}$ the current branch destination. In hardware-based techniques, custom hardware interfaces with the CPU to detect branches and record their destinations in protected memory.

**2.3.3. CFI/CFA Coverage.** We define the coverage of CFI/CFA proposals in terms of *granularity* and *sensitivity*.

**Granularity:** The granularity of CFI/CFA mechanisms refers to the detail in which a particular control flow transfer is monitored/checked. In this work, we categorize the granularity of a specific technique as either *coarse-grained* or *fine-grained*. Since CFI and CFA have different security goals (local detection/prevention vs. providing runtime evidence to a remote party), their granularity pertains to different aspects.

A *coarse-grained* approach refers to broadly applied checks that are independent of specific control flow transfers within the code. In the case of CFI, this involves techniques applied based on instruction type and agnostic to individual transfers. For instance, the following CFI policies can be classified as *coarse-grained*: enforcing landing pads for calls/returns, checking function type/parameter for indirect calls, and restricting indirect control flow targets within the bounds of a specific sandbox/address space. Since these policies are generally applied to all control flow transfers within a specific scope and do not account for the specific details of each transfer, they are considered *coarse-grained*. In CFA, a scheme is deemed *coarse-grained* if it does not record all control flow transfers within the attested application into $CF_{Log}$.

A *fine-grained* technique refers to mechanisms that apply a specific check or action for each control flow instruction. In the case of CFI, this entails schemes that verify each indirect target against a unique set of valid locations rather than applying a broader rule based on the instruction type. For instance, enforcement through techniques like shadow stacks, jump-tables, or definition sets determined by data-flow analysis are considered *fine-grained* solutions. A CFA scheme is classified as *fine-grained* if it records all control flow transfers within the attested application.

**Sensitivity:** Although closely related to granularity, the *sensitivity* of a certain technique describes a different characteristic. It refers to the extent to which execution context is considered for determining the set of valid targets. In this work, we categorize schemes as *insensitive*, *context-sensitive*, or *path-sensitive*.

Techniques have *insensitive* enforcement if they do not consider the calling context or current execution path when defining the set of valid targets for a particular control flow transfer. As such, the majority of coarse-grained CFI mechanisms are regarded *insensitive* because they employ generic rules, e.g., based on the instruction type, ignoring the current execution path or the calling context.

*Context-sensitive* approaches consider the calling context to determine the set of valid targets. Examples include target bounds being within a particular function/sandbox. Additionally, when a function is called at multiple locations within a second function, a *context-sensitive* approach might determine returns to any call site within the second function as valid. For forward edges, a *context-sensitive* approach allows any valid definition that can reach the function containing the forward edge.

*Path-sensitive* approaches determine the set of valid targets by considering both the calling context and the current executing path. For instance, shadow stacks are regarded as *path-sensitive* enforcement mechanisms for return addresses because they limit a return to a single call site. Furthermore, schemes that employ data flow analysis, such as reaching definitions or points-to analysis, to determine the valid destinations of indirect branches are considered *path-sensitive*.

In CFI, sensitivity affects the local decision on whether a transfer constitutes a violation, whereas in CFA, sensitivity reflects the type of analysis/detection that can be performed by $\mathcal{V}$rf based on the received evidence.

**2.3.4. CFI/CFA Threat Models & Assumptions.** The security of most CFI techniques depends on the presence of added instrumentation used to enforce CFI checks. In many cases, this is attained via W⊕X permissions for memory accesses, as shown in Figure 3. While sensible for user-space code, privileged code can typically disable W⊕X enforcement. Therefore, most CFI approaches that target privileged code (e.g., Kernel) rule out code injection/modification from their threat model.

CFA mechanisms require an RoT to implement their attestation functionality, including the acquisition and signing of relevant evidence. The RoT function can also attest the executed binary (and any instrumentation therein) as performed by regular RA. This removes the need for W⊕X enforcement, as long as code is attested in a temporally consistent manner, i.e., code remains the same in the interim between its measurement and execution. This also makes CFA useful to verify privileged code and code that runs on single-privilege Micro Controller Units (MCUs).

Similar to other TEE-based security services, TEE-based CFA (e.g., [33], [38], [43]) assumes that any application outside the (hardware-protected) trusted realm of the TEE (e.g., outside the Secure World in TrustZone) can be modified/compromised whereas the RoT implementation within the Secure World is trusted. This is typically supported by a secure boot of the trusted code and implicitly assumes a minimal and vulnerability-free RoT implementation, as vulnerabilities in the RoT can lead to full system compromise [87]. Some CFA methods eliminate the need to trust a software TCB within the TEE by implementing the CFA RoT entirely in hardware [35]–[37].

Generally, both CFI and CFA consider the underlying hardware to be trusted, focusing on software-based exploits.

## 3. Design Space

Figure 5 illustrates the distinguishing factors in CFI and CFA, highlighting the consequences of design choices on their effectiveness and susceptibility to attack vectors. Accordingly, Table 1 presents a classification of recent work in CFI and CFA, capturing design principles of each mechanism and assessing their trade-offs. Aside from aspects related to security goals (defined in Section 2.3), the rest of this section elaborates on design factors. Afterward, Section 4 discusses the consequences of these design choices.

### 3.1. Different Objectives

CFI mechanisms primarily focus on locally detecting control flow violations during the dispatching stage to prevent execution of exploited code from continuing, as depicted in ❹ in Figure 3. In this context, "prevent" is not to be confused with the goal of memory safety defenses as mechanisms that aim to remove vulnerabilities (see Section 1.3). In other words, CFI does not remove root-cause vulnerabilities. Instead, it impedes certain attack stages, increasing adversaries' difficulty in achieving arbitrary code execution.

Conversely, CFA is concerned with providing authentic execution evidence that can be verified and inspected remotely. In this case, attack detection occurs at a relatively late stage but provides essential insights into attack behavior that can be used to respond to attacks that have evaded prevention measures. This also includes logical bugs (i.e., those not caused by a memory vulnerability) in a program's control flow that CFI would not treat as an exception. In contrast, CFI does not aim to inform or convince a remote party of execution integrity, handling exceptions and faults locally.

When incorporated atop CFA, runtime auditing [41], [42] aims to reliably deliver evidence to $\mathcal{V}$rf, even when a compromised $\mathcal{P}$rv attempts not to follow the CFA protocol (see Section 5), refusing to send reports to $\mathcal{V}$rf in an attempt to hide the exploit behavior.

### 3.2. Action Mechanisms

Action mechanisms fall into (i) enforcement, (ii) monitoring techniques, or (iii) hybrid (i.e. a combination thereof) and can be hardware-assisted or implemented in software. Early CFI designs relied heavily on enforcement through software-based instrumentation (SWI) using generic instructions, so-called Inline Reference Monitors (IRMs) [18], [85], [88]–[97]. More recent proposals leverage hardware extensions for specialized CFI instructions as IRMs [20]–[22], [25]–[27], [98]–[100].

A significant limitation in the above-mentioned approaches is the lack of context sensitivity, with transfers checked individually, making these CFI techniques bypassable, as demonstrated in several attacks [30], [101]–[105]. This has fueled the development of context-sensitive CFI [84], [106]–[112]. Some proposals in this area use advanced points-to-analysis to incorporate path/flow sensitivity to enforce policies effectively. They also leverage commodity hardware features to safeguard the integrity of critical variables that represent the main reference of execution history in such policies [109], [110].

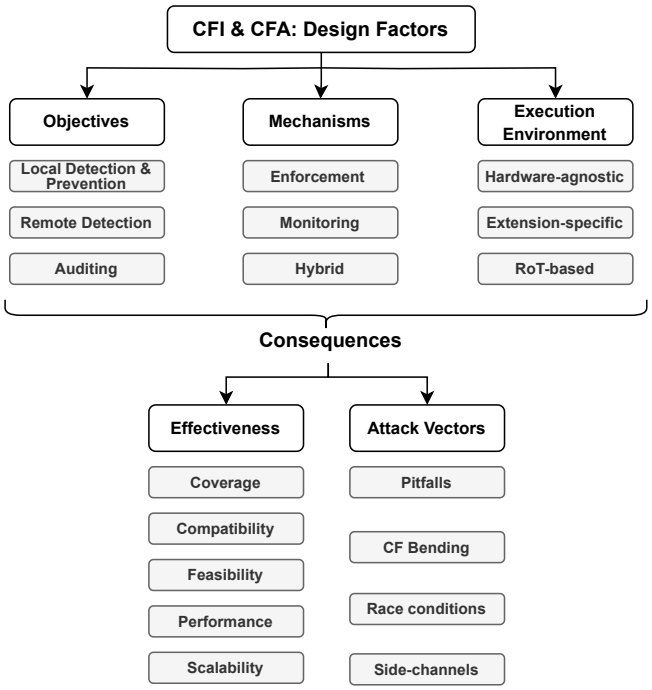


Figure 5. Design Factors of CFI/CFA and Related Consequences

Hybrid CFI approaches implement Hardware Reference Monitors (HRMs) using hardware features to locally save sequences of control flow transfers for asynchronous verification by a separate trusted software module. For instance, PathArmor [106] leverages the Intel Branch Record (LBR) registers to enable implicit monitoring of execution paths, whereas PittyPAT [107] and $\mu$CFI [84] mainly depend on the Intel Processor Tracing (PT) technology [113] to explicitly monitor and verify the execution integrity at runtime. SHERLOC [114] uses ARM Micro Trace Buffer (MTB) and TrustZone for asynchronous detection of CFI violations.

CFA monitors the execution flow, recording transfers to be reported in some form to a $\mathcal{V}$rf. C-FLAT [33] was the first CFA and used software instrumentation to insert IRMs, which redirect each control flow transfer to a secure software routine housed within TrustZone. This routine extends branch destinations into a hash-chain before resuming the attested execution (and performing the branch). TinyCFA [34] shows an instrumentation-based approach to achieve CFA atop a *Proof of Execution* (PoX) architecture, called APEX [115]. Additionally, the work of Papamartzivanos et al. [116] utilizes Intel PT technology [113] for generating the runtime traces. LO-FAT [35] and ATRIUM [36] eliminate instrumentation requirements from C-FLAT by implementing custom hardware modules to detect control flow transfers and extend the hash-chain.

While these early approaches produce evidence that minimizes storage/transmission costs (to the size of one hash digest), they result in loss of information, requiring $\mathcal{V}$rf to use the received hash digest to derive the exact control flow path for inspection. The complexity of this task grows exponentially, leading to the well-known path explosion problem [117], [118].

To ease verification and inspection of CFA evidence, more recent techniques [37], [38], [40], [41], [43], [119] generate $CF_{Log}$ as a lossless trace containing all relevant control flow evidence. For instance, OAT [38], ARI [40], and TRACES [42] leverage TEEs to securely update and store the runtime evidence. LiteHAX [37] and ACFA [41] utilize custom hardware for recording a verbatim trace.

While CFA techniques are primarily focused on monitoring control flow events passively, recent techniques have proposed *hybrid* action mechanisms that monitor the control flow path while providing some enforcement capabilities. For instance, ISC-FLAT [43] creates a TEE-protected dispatcher to ensure that external system interrupts cannot stealthily modify the control flow of the application being attested. CFA+ [120] leverages ARMv8.5-A's landing pad instructions [98] in combination with selective software instrumentation to enforce a specific CFI policy and enable lightweight monitoring of the execution state, which is maintained in two reserved registers.

### 3.3. System Models & Execution Environments

CFA and CFI concepts can apply to both high-end systems and low-end embedded devices. Additionally, the requirements of execution environments within the target platform for CFI/CFA can be distinguished as: hardware-agnostic, extension-specific, and RoT-based.

CFI focuses mainly on the first two types of execution environments, and it has primarily been applied in high-end systems. Many earlier CFI approaches are hardware agnostic and utilize SWI to monitor control flow events and locally detect violations. These CFI mechanisms, e.g., LLVM-CFI [18] and Microsoft Control Flow Guard (MS-CFG) [121], are easily portable and can cover a variety of high-end targets regardless of the particular CPU type. As they are hardware-agnostic and instrumentation-based, their system models require the presence of a Memory Management Unit (MMU) or an Operating System (OS) to maintain the integrity of the instrumentation. When applied to bare-metal embedded systems that lack MMUs, a Memory Protection Unit (MPU) is used for similar guarantees [111], [122].

The portability of CFI in hardware-agnostic environments can come at the price of performance. Hence, several CFI approaches use specific architectural (hardware) extensions in their local environments. Some involve custom hardware extensions specifically designed to support CFI, while others are repurposed from their other goals and integrated as building blocks into CFI. Examples of the former include Intel CET [21] and ARM Pointer Authentication [20], along with the body of work built upon them [22], [23], [25], [27], [99], [100]. Examples of the latter category include CFI using Intel PT [84] and ARM Trace Macrocell (TMC) [123]. As extension-specific execution environments are available on both high-end and embedded platforms, CFI proposals using them can also be applied to both classes of target devices.

Most existing CFA approaches target embedded platforms, including those with low-power single-core MCUs (e.g., Atmel AVR ATMega, TI MSP430) with 1-16MHz CPUs, 8-16 bit instruction set architectures (ISAs), used to execute "bare-metal" software and commonly implement IoT sensors/actuators and control systems. Additionally, CFA has been proposed for 32-bit embedded platforms with TEE support (e.g., ARM Cortex-M MCUs) that execute real-time applications. As discussed in Section 2.3, CFA necessitates RoT support to generate (and sign) remotely verifiable evidence. RoTs in CFA for embedded devices are implemented via TEEs [33], [38], [42], [43] or custom hardware changes on $\mathcal{P}$rv [34]–[37], [41], [124]. Current CFA proposals aimed at user-space programs [119], [120], [125] either trust the OS (the code integrity of which can be verified using static RA as supported by commodity TPMs) or rely on enclaved execution TEEs [126].

## 4. Effects & Consequences

This section discusses the effects and consequences of design choices presented in Section 3.

### 4.1. Effectiveness

**4.1.1. Coverage.** In terms of coverage, CFI mechanisms offer varying degrees of protection, ranging from protecting all edges, i.e., all types of indirect control flow altering instructions [14], [88]–[91], [107], [127], [128], to partial coverage, targeting either forward-edges [18], [22], [98], [100], [109], [110], [112], [121], [129] or backward-edges [20], [21], [27], [99], [111], [122], [130], [131]. Forward-edge [9] schemes employ IRMs via SWI [18], [121], [129], hardware-assisted monitoring [84], [107], landing pads [22], [98], and pointer authentication [26], [92], [100]. Backward edge [8] schemes utilize software- [122], [132] or hardware-based [20], [21] shadow stacks, and architecture-specific features, such as branch history tables in the x86 processors [133], [134], static rewriting, e.g., to form jump tables [111], or pointer authentication [27], [99].

Additionally, CFI designs can offer partial protection/-coverage in specific scenarios. For example, some designs concentrate on protecting C-like applications [129] while leaving out relevant structures specific to C++, such as vtables, and vice-versa [18]. Conversely, other designs focus on statically linked applications [14], [88], [89], [92], [111], overlooking dynamic linking and associated concerns, such as protecting Procedure Linkage Tables (PLT) and Global Offset Tables (GOT) [91], [135].

The majority of CFI mechanisms (regardless of their coverage) are context insensitive [14], [18], [22], [88], [89], [91], [92], [121], [128], [136]. This can introduce gaps that

are challenging to detect [69]. Moreover, it limits the ability to detect non-control-data attacks [10].

CFA schemes can enhance coverage and expand (remote) detection capabilities beyond traditional control-flow hijacking attacks. As CFA evidence includes the executed control flow path, it (in principle) informs $\mathcal{V}$rf of any out-of-order execution, including DOP attacks which are oblivious to most CFI. Some approaches [37], [38], [76] include data inputs within CFA, augmenting produced evidence to also make DDMs observable. It is important to note, however, that CFA evidence is only truly useful if $\mathcal{V}$rf can effectively analyze it. This last aspect has been, for the most part, overlooked in the current literature. We revisit this point in Section 5.2.

**4.1.2. Compatibility.** Compatibility is a fundamental aspect to consider when evaluating the effectiveness of CFI and CFA mechanisms and manifests in various forms.

**Binary Support.** Despite the abundance of CFI mechanisms, few operate directly on binary using static binary analysis [88], [89], [95], [96] or specific hardware extensions [94], [107], [128], [130], [136], [143], [144]. While these have broader applicability, they suffer from false positives [103] typically employing ad-hoc approaches to recover CFGs or simply marking all address-taken functions and call-site preceded instructions as legitimate targets for indirect branches [17]. Conversely, CFI based on source code [18], [26], [84], [91], [100], [106], [109], [110] constructs more accurate CFGs. Access to source code typically allows for advanced static analysis techniques (e.g., points-to analysis as seen in $\mu$CFI [84] and multi-layer type analysis as seen in MLTA [86]), increasing coverage and precision. However, source-level schemes do not apply to commercial off-the-shelf (COTS) software, where only binary images are available [15].

CFA designs typically do not require source code knowledge to generate control flow evidence [33], whereas source code knowledge may assist $\mathcal{V}$rf in analyzing received evidence (see Section 5.2). Hardware-based CFA inherently supports binaries [35]–[37], [41] by integrating with the CPU core and detecting branch instructions at runtime. CFA relying on SWI [33], [34], [38], [39], [42], [43] can instrument control flow transfers in the binary without knowing the source. This is because required instrumentation is used only to log destination addresses rather than determining/enforcing policies in place. Exceptions to this include schemes mixing evidence generation with local integrity checks, e.g., [120], [124].

**Modular/Shared Object Support.** A limitation of many CFI mechanisms is the lack of support for external modules or dynamic shared objects (DSO). These mechanisms often rely on global information that may not always be available, making it challenging to implement globally compatible CFI. Abstractly speaking, support for external/shared modules involves (i) integrating multiple modules hardened by CFI separately and (ii) integrating CFI-protected modules with unprotected legacy code. Binary solutions such as CCFIR [89] attempt to address these issues by allowing more targets than necessary, striking a security-compatibility compromise. Although approaches such as MCFI [91] and RockJIT [137] tackle case (i) by independently instrumenting each module and generating new CFGs when modules are linked, recent CFI solutions that offer stronger security guarantees, exemplified by $\mu$CFI [84] and OS-CFI [109], do not provide modular support. Even contemporary solutions employing hardware features, e.g., PACStack [27] and PACTight [100], struggle to address both issues (i) and (ii).

While not explicitly discussed in prior work, the lack of modular support in CFA can be attributed to (i) most CFA proposals being aimed at simple embedded systems (as seen in Table 1) where applications are statically linked within a single module; and (ii) DSO support would have implications on $\mathcal{V}$rf evidence analysis, requiring careful consideration.

**Hardware Dependence.** Hardware-specific features can enhance CFI and CFA. However, they limit a scheme's compatibility to architectures that support them and introduce challenges for legacy systems. For instance, CFI like PittyPAT [107], GRIFFIN [128], $\mu$CFI [84], and PathArmor [106] utilize Intel PT and LBR to obtain runtime information and compute a smaller set of legitimate targets, striking a balance between accuracy and performance overhead. Similarly, OS-CFI [109], and CFI-LB [110] leverage Intel TSX (Transactional Synchronization Extensions) and MPX (Memory Protection Extensions) to safeguard instrumented code and metadata against malicious tampering. Approaches such as HCFI [127] propose custom hardware modifications.

TEE-based CFA schemes demonstrate how instrumentation can be used alongside RoT hardware support (e.g., ARM TrustZone [80], Intel MPK [145], or PoX architecture [115]) to implement CFA. Early hardware-based CFA, such as LO-FAT [35] and ATRIUM [36], add custom branch monitors and hash engines to detect and accumulate control flow transfers as a hash digest. LiteHAX [37] opts for more expressive evidence, using dedicated hardware to log and store all control flow transfers, aiming at easing $\mathcal{V}$rf subsequent analysis. ACFA [41] uses custom hardware for branch detection while eliminating the cost of hash engines to make instrumentation-less CFA feasible in budget-constrained micro-controllers. Instead, it incorporates components of a static RA architecture (VRASED [72]) and an active RoT (GAROTA [146]). The former is used to authenticate CFA evidence, while the latter is leveraged to ensure reliable delivery of evidence to $\mathcal{V}$rf (enabling auditing guarantees).

**Functionality.** Recent evaluations of various CFI defenses have highlighted compatibility issues that can compromise the intended functionality of the target application [69], [147]. Notably, the implementation approach in Lockdown [95] and OS-CFI [109] fails to correctly compile certain applications, e.g., nginx. Moreover, CFI mechanisms such as OS-CFI [109] and CFI-LB [110] have been found to generate false positives. Additionally, the analysis mechanism of LLVM-CFI [18] is incompatible with at least one application in the SPEC CPU2006 suite, as reported in [22]. CFI mechanisms that depend on reserving registers, e.g.,

TABLE 1. CATEGORIZATION OF CFI AND CFA SCHEMES, HIGHLIGHTING THEIR MAIN PROPERTIES AND REQUIREMENTS.

| | | Device Type/Target | | | | Mechanism | | | | Scope | | | | Overheads | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Year | Scheme | Embedded (bare-metal) | Embedded (OS) | High-end (User-space) | High-end (Kernel) | Type | Strategy | Sensitivity | System Support | Data-Only | ROP | JOP | Evidence Expressiveness | Runtime | Code Size | Custom Hardware | Network |
| **Control Flow Integrity (CFI) Approaches** | | | | | | | | | | | | | | | | | |
| 2013 | bin-CFI [88] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | ✗ | OS/MMU | ✗ | ○ | ○ | - | ● | ● | ✗ | - |
| 2013 | CCFIR [89] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+R/I | ✗ | OS/MMU | ✗ | ○ | ○ | - | ● | ● | ✗ | - |
| 2014 | **LLVM CFI** [18] | ✗ | ✗ | ✓ | ✓ | Enforcement | SWI | ✗ | OS/MMU | ✗ | ✗ | ◐ | - | ● | ● | ✗ | - |
| 2014 | KCoFI [90] | ✗ | ✗ | ✗ | ✓ | Enforcement | SWI | ✗ | MMU | ✗ | ◐ | ◐ | - | ● | ● | ✗ | - |
| 2014 | MCFI [91] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | CS | OS/MMU | ✗ | ◐ | ◐ | - | ● | ● | ✗ | - |
| 2014 | RockJIT [137] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | CS | OS/MMU | ✗ | ◐ | ◐ | - | ● | ● | ✗ | - |
| 2015 | CCFI [92] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | ✗ | OS/MMU | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2015 | HAFIX [130] | ✓ | ✗ | ✗ | ✗ | Enforcement | ISA | Path | C-HW | ✗ | ● | ✗ | - | ● | ● | ● | - |
| 2015 | CFCI [93] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | ✗ | OS/MMU | ✗ | ○ | ○ | - | ● | ● | ✗ | - |
| 2015 | O-CFI [94] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+R/I | ✗ | OS/MMU+MPX | ✗ | ◐ | ◐ | - | ● | ● | ✗ | - |
| 2015 | $\pi$CFI [85] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | ✗ | OS/MMU | ✗ | ◐ | ◐ | - | ● | ● | ✗ | - |
| 2015 | PathArmor [106] | ✗ | ✗ | ✓ | ✗ | Hybrid | SWI+ISA | Path | OS/MMU+LBR | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2015 | Lockdown [95] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+R/I | ✗ | OS/MMU | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2016 | TypeArmor [96] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | ✗ | OS/MMU | ✗ | ✗ | ◐ | - | ● | ● | ✗ | - |
| 2016 | FG-CFI [97] | ✗ | ✗ | ✗ | ✓ | Enforcement | SWI | ✗ | MMU | ✗ | ◐ | ◐ | - | ● | ● | ✗ | - |
| 2016 | HCFI [127] | ✗ | ✗ | ✓ | ✗ | Enforcement | ISA | ✗ | OS+C-HW | ✗ | ● | ● | - | ● | ● | ● | - |
| 2017 | PittyPAT [107] | ✗ | ✗ | ✓ | ✗ | Hybrid | ISA+HRM | Path | OS/MMU+PT | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2017 | GRIFFIN [128] | ✗ | ✗ | ✓ | ✗ | Hybrid | ISA+HRM | ✗ | OS/MMU+PT+TSX | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2017 | CFI-CaRE [136] | ✓ | ✗ | ✗ | ✗ | Enforcement | SWI+R/I | ✗ | TZ | ✗ | ● | ◐ | - | ● | ● | ✗ | - |
| 2017 | **Intel CET** [21] | ✗ | ✗ | ✓ | ✓ | Enforcement | ISA | ✗ | OS/MMU+CET | ✗ | ● | ○ | - | ● | ● | ✗ | - |
| 2018 | $\mu$CFI [84] | ✗ | ✗ | ✓ | ✗ | Hybrid | SWI+ISA | ✗ | OS/MMU+PT | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2018 | SCFP [108] | ✓ | ✗ | ✗ | ✗ | Enforcement | SWI+C-HW | Path | C-HW | ✗ | ● | ◐ | - | ● | ● | ● | - |
| 2018 | **ARM BTI** [98] | ✓ | ✓ | ✓ | ✓ | Enforcement | ISA | ✗ | BTI | ✗ | ✗ | ○ | - | ● | ● | ✗ | - |
| 2018 | **PAC-RET** [99] | ✓ | ✓ | ✓ | ✓ | Enforcement | ISA | ✗ | PA | ✗ | ● | ✗ | - | ● | ● | ✗ | - |
| 2019 | OS-CFI [109] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+R/I | Path | OS/MMU+MPX+TSX | ✗ | ✗ | ● | - | ● | ● | ✗ | - |
| 2019 | CFI-LB [110] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+R/I | CS | OS/MMU+TSX | ✗ | ✗ | ● | - | ● | ● | ✗ | - |
| 2019 | PARTS [25] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+ISA | ✗ | OS/MMU+PA | + | ● | ● | - | ● | ● | ✗ | - |
| 2020 | $\mu$RAI [111] | ✓ | ✗ | ✗ | ✗ | Enforcement | SWI+R/I | Path | MPU | ✗ | ● | ✗ | - | ● | ● | ✗ | - |
| 2020 | Silhouette [122] | ✓ | ✗ | ✗ | ✗ | Enforcement | SWI+R/I | ✗ | MPU | ✗ | ● | ✗ | - | ● | ● | ✗ | - |
| 2021 | VIP [112] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+R/I | Path | OS/MMU+MPK | + | ✗ | ● | - | ● | ● | ✗ | - |
| 2021 | PACStack [27] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+ISA | ✗ | OS/MMU+PA | ✗ | ● | ✗ | - | ● | ● | ✗ | - |
| 2022 | TyPro [129] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | ✗ | OS/MMU | ✗ | ✗ | ● | - | ● | ● | ✗ | - |
| 2022 | PAL [26] | ✗ | ✗ | ✗ | ✓ | Enforcement | SWI+ISA | ✗ | PA+MMU | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2022 | PACTight [100] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI+ISA | ✗ | OS/MMU+PA | ✗ | ● | ● | - | ● | ● | ✗ | - |
| 2023 | FineIBT [22] | ✗ | ✗ | ✓ | ✓ | Enforcement | SWI+ISA | ✗ | CET+MMU | ✗ | ✗ | ◐ | - | ● | ● | ✗ | - |
| 2023 | SHERLOC [114] | ✓ | ✗ | ✗ | ✗ | Hybrid | ISA+HRM | Path | TZ+MTB+DWT | ✗ | ● | ◐ | - | ● | ● | ✗ | - |
| 2023 | TypeSqueezer [138] | ✗ | ✗ | ✓ | ✗ | Enforcement | SWI | Path | OS/MMU | ✗ | ✗ | ● | - | ● | ● | ✗ | - |
| 2024 | HEK-CFI [139] | ✗ | ✗ | ✗ | ✓ | Enforcement | ISA | ✗ | CET+MMU | ✗ | ● | ◐ | - | ● | ● | ✗ | - |
| **Control Flow Attestation (CFA) Approaches** | | | | | | | | | | | | | | | | | |
| 2016 | C-FLAT [33] | ✓ | ✗ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | TZ | □ | ● | ● | △ | ● | ● | ✗ | ☆ |
| 2017 | LO-FAT [35] | ✓ | ✗ | ✗ | ✗ | Monitoring | C-HW | $\mathcal{V}$rf-based | C-HW | □ | ● | ● | △ | ✗ | ✗ | ● | ☆ |
| 2017 | ATRIUM [36] | ✓ | ✗ | ✗ | ✗ | Monitoring | C-HW | $\mathcal{V}$rf-based | C-HW | □ | ● | ● | △ | ✗ | ✗ | ● | ☆ |
| 2018 | LiteHAX [37] | ✓ | ✗ | ✗ | ✗ | Monitoring | C-HW | $\mathcal{V}$rf-based | C-HW | ⊞ | ● | ● | ▲ | ✗ | ✗ | ● | ★ |
| 2019 | DIAT [140] | ✓ | ✓ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | TZ | □ | ● | ● | △ | ● | ● | ✗ | ☆ |
| 2019 | ScaRR [119] | ✗ | ✗ | ✓ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | OS/MMU | □ | ● | ● | ▲ | ● | ● | ✗ | ★ |
| 2019 | RIM [124] | ✓ | ✗ | ✗ | ✗ | Monitoring | C-HW | Path | C-HW | ⊞ | ● | ● | △ | ✗ | ✗ | ? | ☆ |
| 2020 | OAT [38] | ✓ | ✓ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | TZ | ⊞ | ● | ● | ◭ | ● | ● | ✗ | ☆ |
| 2020 | LAHEL [141] | ✓ | ✓ | ✗ | ✗ | Monitoring | C-HW | $\mathcal{V}$rf-based | C-HW | □ | ◐ | ◐ | ◭ | ● | ● | debug HW | ☆ |
| 2020 | LAPE [142] | ✓ | ✗ | ✗ | ✗ | Monitoring | SWI+R/I | $\mathcal{V}$rf-based | MPU | □ | ◐ | ◐ | △ | ● | ● | ✗ | ☆ |
| 2021 | Tiny-CFA [34] | ✓ | ✗ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | C-HW | □ | ● | ● | ▲ | ● | ● | ✗ | ☆ |
| 2021 | DIALED [76] | ✓ | ✗ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | C-HW | ⊞ | ● | ● | ▲ | ● | ● | ✗ | ☆ |
| 2021 | ReCFA [125] | ✗ | ✗ | ✓ | ✗ | Monitoring | SWI+R/I | $\mathcal{V}$rf-based | OS+MPK | □ | ● | ● | ▲ | ● | ● | ✗ | ☆ |
| 2022 | GuaranTEE [126] | ✗ | ✗ | ✓ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | Intel SGX | □ | ● | ● | △ | ● | ● | ✗ | ☆ |
| 2023 | ACFA [41] | ✓ | ✗ | ✗ | ✗ | Monitoring | C-HW | $\mathcal{V}$rf-based | C-HW | □ | ● | ● | ▲ | ✗ | ✗ | ● | ★ |
| 2023 | ARI [40] | ✓ | ✓ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | TZ | □ | ◐ | ◐ | ◭ | ● | ● | ✗ | ☆ |
| 2023 | BLAST [39] | ✓ | ✓ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | TZ | □ | ● | ● | ▲ | ● | ● | ✗ | ☆ |
| 2023 | ISC-FLAT [43] | ✓ | ✗ | ✗ | ✗ | Hybrid | SWI | $\mathcal{V}$rf-based | TZ | □ | ● | ● | ▲ | ● | ● | ✗ | ☆ |
| 2024 | TRACES [42] | ✓ | ✗ | ✗ | ✗ | Monitoring | SWI | $\mathcal{V}$rf-based | TZ | □ | ● | ● | ▲ | ● | ● | ✗ | ★ |
| 2024 | CFA+ [120] | ✓ | ✓ | ✓ | ✗ | Hybrid | SWI+ISA | $\mathcal{V}$rf-based | OS/MMU+TPM | □ | ● | ● | ▲ | ● | ● | ✗ | ☆ |

**Legend:** ✓ Has this feature, ✗ Lacks this feature, - Feature is not applicable, **SWI**: Software Instrumentation, **HRM**: Hardware Reference Monitor, **R/I**: Randomization or Isolation, **ISA**: Instruction Set Architecture, **MMU:** Memory Management Unit, **C-HW**: Custom Hardware, **OS**: Operating System, **CS**: Context Sensitive, **Path**: Context- & path-sensitive **MPX**: Intel Memory Protection eXtensions **LBR**: Intel Last Branch Record **PT**: Intel Processor Trace **TSX**: Intel Transactional Synchronization Extension **TZ**: ARM TrustZone, **CET**: Intel Control-Flow Enforcement Tech., **BTI**: ARM Branch Target Identification, **PA**: ARM Pointer Authentication, **MPU**: Memory Protection Unit, **MPK**: Intel Memory Protection keys, **MTB**: ARM Macro Trace Buffer, **DWT**: ARM Data Warchpoint and Trace, + Definition-to-use data corruption, □ Path deviation DOP, ⊞ Definition-to-use data corruption & path deviation DOP, ● Fine-grained, ◐ Mixed granularity, ○ Coarse-grained, △ No path or hashed path, ◭ Partial path, ▲ Full path (lossless), ☆ All evidence stored and transmitted at once, ★ Evidence sliced and streamed, ✗ No overhead, ● <5% overhead, ● <20% overhead, ● Higher overhead, **?** Feature required but cost not reported, **debug HW** reliance on prototyping/debug features not meant for device deployment.

VIP [112], can corrupt functionality when targeting applications with inline assembly that utilize the same registers.

CFA mechanisms that instrument binaries may also encounter compilation failures due to instrumentation issues,

as observed in ReCFA with specific benchmarks [125]. As CFA is a newer concept, fewer studies exist on analyzing CFA instrumentation compatibility. At least in principle, issues presented in CFI schemes could also apply to CFA, depending on the instrumentation strategy used. (For instance, similar to VIP-CFI, TinyCFA and TRACES also employ reserved registers.)

**4.1.3. Feasibility.** CFI approaches are to a large extent feasible, despite inherent uncertainties around the robustness of policies due to granularity and context sensitivity (discussed further in Section 4.2). As with any attestation mechanism, CFA requires a secure RoT on $\mathcal{P}$rv to maintain and authenticate evidence, as discussed in Section 3.2. It also requires communication with an external $\mathcal{V}$rf. Naturally, custom hardware features (such as branch monitors) improve feasibility and reduce the cost of CFA. Being a relatively recent concept, we expect hardware features to support CFA to take longer to reach off-the-shelf devices.

**4.1.4. Performance & Scalability.** When comparing the scalability of CFI and CFA, it becomes evident that CFI generally encounters fewer or no scalability challenges due to their localized nature. Since the scope of CFI is confined to local decisions based on control flow policies, scalability issues revolve around code size and runtime of the individual applications being protected. A study dedicated to CFI performance can be found in [15].

Performance and scalability in CFA depend on the monitoring strategy. Techniques that utilize SWI incur runtime and code-size increases due to logging [34], context switching into secure environments [39], or both [33], [38], [40], [42], [119], [140], [142]. Alternatively, hardware-based CFA mechanisms trade the performance/scalability impact of SWI for additional hardware costs. Early hardware-based approaches impose expensive overheads due to large internal buffers and hardware hash-engines [35], [36]. Alternative techniques leverage minimal hardware [34], [37] or propose hardware/software co-designs [41] to lower the hardware cost. Regardless of their specificities, existing hardware-based CFA mechanisms are tailored to embedded platforms, leaving scalability in more complex computing platforms as an open challenge.

Unlike CFI, CFA faces further performance challenges in storing and transmitting runtime evidence. Schemes such as ScaRR [119], ACFA [41], and TRACES [42], which continuously report evidence to $\mathcal{V}$rf, may face challenges when attempting to cover multiple active applications on the same $\mathcal{P}$rv. This may impact availability, particularly when network communication is essential (e.g., cloud). In other schemes, $\mathcal{P}$rv may need to store a large $\mathrm{CF}_{Log}$ if attested operations are complex. The latter can limit CFA applicability to small and self-contained operations [34], [38]. Recent work has investigated dynamically configurable and application-specific optimizations for CFA evidence in MCUs [148]. Nevertheless, additional research is required to realize CFA (and related optimizations) in higher-end systems.

## 4.2. Attack Vectors

In this section, we explore how gaps or design choices (typically aimed at trading off performance for security) in CFI and CFA can lead to attack vectors.

**4.2.1. Pitfalls.** Attacks can exploit various well-known pitfalls or limitations, including, but not limited to:

**Granularity**: many past attacks have exposed the ineffectiveness of coarse-grained CFI defenses for both forward and backward edges [101]–[103].

**Implementation issues**: the implementation of defenses may deviate from their design specifications, leading to a larger number of allowed branch targets than necessary. For example, [69] highlighted implementation mistakes in multiple CFI defenses, including MCFI [91] and PARTS [25].

**Imprecise consideration of language semantics**: Attacks such as COOP [149] have affected T-VIP [150] and VTint [151] due to inadequate incorporation of language-specific semantics.

**Hardware design limitations**: Certain attacks have specifically targeted the hardware design of CFI mechanisms. For instance, the attack on HAFIX [130] highlighted vulnerabilities stemming from hardware limitations [152].

**Exploitation of Assumptions**: defenses always rely on assumptions within their threat models. Thus, attacks can exploit and falsify these trust assumptions to bypass the defense mechanisms. For instance, a widespread CFI assumption is W⊕X. The POP attack [70] serves as an example where this assumption was violated to bypass FineIBT [22] defense on the Linux kernel v6.2.8.

**Corner Cases**: Certain attacks exploit exceptional cases. For instance, the Control Jujutsu attack [153] highlighted the limitations of fine-grained CFI defenses with activated shadow stacks in complex code bases like Apache and nginx. Due to coding practices in these code bases, context-insensitive analysis, regardless of its intended robustness, creates over-approximated CFGs that render CFI ineffective. Another example is CHOP [154], which further undermines robust backward edge protection mechanisms, including hardware-based shadow stack implementations [21]. It leverages a specific corner case that enables manipulation of the stack unwinding path during exception handling to launch ROP-like attacks, using the unwinder as a confused deputy.

**CFA has not yet been extensively evaluated**: coarser-grained CFA [40] (or those based on attesting $\mathcal{P}$rv adherence to locally enforced CFI policies [124]) may be subject to the same attack vectors as coarse-grained/context-insensitive CFI, where certain attacks would not appear in the CFA evidence. Yet, CFA that monitors all indirect branches can withstand language semantic issues, enabling detection of attacks such as COOP [149]. Additionally, CFA can also provide evidence of logic implementation bugs that lead to unintended paths, in addition to attacks rooted in memory safety vulnerabilities. Naturally, the expressiveness of CFA evidence (i.e., whether it gives $\mathcal{V}$rf full path evidence or a subset) comes at the price of its (lossless) storage and transmission. Unsurprisingly, implementation deviations (from

intended specifications) and falsifiable assumptions would equally affect CFA and CFI.

**4.2.2. Control Flow Bending (CFB).** CFB attacks [19] generalize non-control data attacks targeting CFI schemes relying on statically generated CFGs. While many CFI attacks target weaker or sub-optimal implementations [101]–[103], CFB focuses on bypassing the most restrictive (or optimal) static CFI policies. CFB creates malicious (Turing-complete) paths that exist on the most strict CFG for a given program by exploiting specific functions, called *dispatchers*, which have the capability to modify their own return addresses. In other words, CFB can arbitrarily modify (bend) a program's behavior/path while staying within the confines of the imposed security policy.

This highlights that even fine-grained CFI can be bypassed if dynamic backward protection is not implemented (e.g., via a secure shadow stack). To mitigate CFB, certain CFI proposals incorporate dynamic analysis [85] or leverage hardware features that provide runtime information on execution status [84]. Additionally, context-sensitive CFI schemes have the potential to reduce the impact of CFB by maintaining an execution history and validating the execution of return instructions accordingly [109], [110].

Most CFA approaches log all dynamically defined branch targets within their execution scope. Thus, CFB path deviations appear in generated evidence, making CFB attacks apparent to $\mathcal{V}\mathsf{rf}$. That said, (similar to cases discussed above) the effectiveness $\mathcal{V}\mathsf{rf}$ in detecting CFB based on CFA evidence remains to be concretely evaluated.

**4.2.3. Race Conditions.** Many CFI methods overlook thread safety in multi-threaded applications. This can leave them vulnerable to Time-Of-Check-to-Time-Of-Use (TOCTOU) attacks. Software-based approaches such as LLVM-CFI [18] face challenges in ensuring thread safety, especially in the presence of blind compiler optimizations that can inadvertently expose sensitive variables used for security checks. This can create race conditions that enable TOCTOU attacks [155].

Additionally, WarpAttack [156] revealed that compiler optimizations could introduce double-fetch vulnerabilities, resulting in concurrency issues and TOCTOU, even with a strict static CFI policy that includes both forward and backward-edge protections. WarpAttack bypassed several CFI defenses, including LLVM-CFI [18], Lockdown [95], and MS-CFG [121]. To mitigate race conditions, contemporary CFI mechanisms rely on hardware support. For example, OS-CFI [109] and CFI-LB [110] utilize Intel TSX to safeguard intermediate values.

In CFA (and more broadly RA), resistance against TOCTOU attacks and race conditions often refers to achieving temporal consistency between when the executable binary is measured and when it is executed [34], [157]–[159]. Aside from modifications to code, the integrity of CFA evidence can be compromised by external interrupts that may stealthily modify the control flow path or the execution state, as shown and mitigated by ISC-FLAT [43].

**4.2.4. Side channels.** The emergence of microarchitectural attacks can affect CFI and CFA. While these defenses focus on memory corruption attacks, certain variants of Spectre [160] can affect them. For instance, Spectre v1 exploits misspeculation following a bounds-check prior to an array access, while Spectre v2 exploits misprediction of the target of an indirect call or jump. Both utilize a Flush+Reload channel [161] to leak data. Research has demonstrated that Spectre v1-like attacks can bypass software-based CFI defenses, such as LLVM-CFI [18], even in the presence of all default mitigations [162]. While specialized mitigations, such as SPECCFI [163] and MicroCFI [164], were proposed, Spectre v2 remains severe and yet to be fully mitigated.

Although contemporary CFI defenses, such as Intel CET [21], consider a post-Spectre threat model and are designed with built-in protection against Spectre v2 [165], recent attacks, such as InSpectre Gadget [166], have uncovered new types of exploitable gadgets that can successfully mount Spectre v2 attacks, even if the CET's Indirect Branch Tracking (IBT) feature or its recent fine-grained counterpart, FineIBT [22], are active. PACMAN [30] stands out as another recent attack that exploits speculative execution along with memory corruption to bypass ARM Pointer Authentication on Apple M1 SoCs.

Similar to CFI, CFA leveraging architectural components vulnerable to side channels could be equally vulnerable. On the other hand, several secret dependency-related time side channels [167] (that exploit software implementation bugs, rather than micro-architectural bugs) depend on differences in the target program's control flow path, opening opportunities for exploit identification based on $\mathrm{CF}_{Log}$ analysis. To our knowledge, the latter remains unexplored in prior work.

## 5. Takeaways and Paths Forward

We conclude this paper synthesizing insights from discussions presented in Section 3, Section 4, and Table 1. Based on these insights, we revisit questions **[Q1-Q4]** from Section 1.

### 5.1. Takeways

Considering question **[Q1]** posed in Section 1, this systematization presents several differences between CFI and CFA. The effectiveness of CFI mechanisms is intrinsically tied to the comprehensiveness and accuracy of a (statically-defined or dynamic) policy enforced locally. Most CFA techniques are policy-agnostic, passively monitoring execution to generate authenticated control flow reports. Contrary to CFI, CFA concerns convincing a remote party of trustworthy execution behavior, serving as a runtime analog to static attestation methods that prove the integrity of booted/loaded code. Thus, CFA reports are transmitted to a remote $\mathcal{V}\mathsf{rf}$ for analysis. These observations lead us to Takeaway 1.

Regarding **[Q2]**, we first examine CFA/CFI assumptions. Many CFI schemes assume the ability to apply W⊕X on memory to preserve the integrity of IRMs and avoid code injection. In Table 1, this is apparent from user-space CFI

**Takeaway 1: CFI and CFA have different goals**

CFI focuses on local detection of control-flow violations, whereas CFA provides remote evidence of execution behavior irrespective of underlying policy enforcement.

schemes frequently relying on OS/MMU system support to enforce the W⊕X policy. While CFA mechanisms need not impose W⊕X, they must rely on an attestation RoT in $\mathcal{P}$rv to attest that reported runtime evidence is authentic (this includes code integrity and instrumentation, when applicable). Furthermore, unlike CFI, CFA requires network connectivity between $\mathcal{V}$rf and $\mathcal{P}$rv. Despite these differences, we also observe that state-of-the-art techniques for CFI and CFA intersect in their mechanisms for monitoring control flow events. For instance, many schemes utilize IRMs via SWI as a mechanism while relying on hardware (whether commodity or custom) to protect or support their instrumentation, as shown in Table 1. Table 1 also shows that both CFI and CFA can optimize SWI using specific ISA extensions. This is summarized in Takeaway 2.

**Takeaway 2: Design intersections & differences**

Although CFI and CFA schemes share many commonalities in their strategies (as apparent in the `Mechanism` column of Table 1), they also have distinct requirements for their system models, e.g., as seen in the `System Support` and `Network Overhead` columns of Table 1.

A common misconception/over-simplification that relates to **[Q3]** is that CFA's entire purpose is to enable CFI checks to be outsourced to a resource-rich $\mathcal{V}$rf, avoiding CFI costs on $\mathcal{P}$rv. As extensively discussed in this systematization, CFA goals go beyond outsourcing CFI checks. As evidence of that, recent CFA methods have evolved to generate expressive (often lossless) control flow path evidence, as opposed to proving adherence to a locally enforced CFI policy (see `Evidence Expressiveness` column, in Table 1). This is subsumed by Takeaway 3.

Regarding **[Q4]**, given their distinct security goals, the coexistence of CFI and CFA on the same platform could be possible if the performance overhead is acceptable in the target domain. We believe the exploration of approaches that combine the strengths of CFI and CFA to be an intriguing avenue for further research. A potential hybrid design might include CFI building blocks that can be elegantly incorporated into CFA reports. Considering that many state-of-the-art CFI offers fine-grained local ROP detection with low overheads (as seen in `Scope` and `Overheads` columns in Table 1), a hybrid approach might implement CFI techniques for local ROP detection while utilizing CFA techniques for

**Takeaway 3: CFA goes beyond outsourced CFI**

While CFI is clearly the best choice for local detection of runtime attacks, CFA enables remote (and offline) control flow path analysis, giving remote visibility to complex path deviations (e.g., Control Flow Bending) that would often be oblivious to most CFI – see `Scope` column in Table 1. CFA evidence also makes logic control path bugs (other than memory corruption) observable. Finally, it facilitates auditing and root cause analysis if the evidence is reliably delivered to $\mathcal{V}$rf. On the other hand, remote observability in CFA comes at the cost of supporting communication and securely implementing an attestation RoT.

generating expressive evidence of path deviations due to JOP, and/or logic control bugs. Yet, CFI/CFA integration is non-trivial, as differences in designs and system assumptions should be considered and can contribute to overheads. As a first step in this direction, CFA+ [120] recently proposed a mechanism that combines CFI and CFA to locally enforce specific targets for certain control flow transfers, relying on ARMv8.5-A landing pad instructions [98] while leveraging minimal instrumentation to record path information in reserved registers. These observations are summarized in Takeaway 4.

**Takeaway 4: Coexistence merits investigation**

Given the trade-offs between CFI and CFA, a hybrid approach could offer both local responses to simpler runtime attacks and remote visibility to complex attacks and their root causes. On the other hand, overheads of both approaches on the same platform could challenge practical adoption.

## 5.2. Paths Foward

**Demand for Stronger Threat Models**. Currently considered threat models (in both CFI and CFA) can be limited in scope or may not adequately address the challenges posed by sophisticated adversaries (e.g., those capable of launching side-channel attacks). Next-generation mechanisms could consider stronger threat models to encompass new attack vectors that can lead to control-flow violations.

**CFA Support for Complex Software**. The current landscape of CFA mechanisms primarily focuses on addressing the needs of simple, specialized, bare-metal embedded software (see column `Device Type/Target` in Table 1). This limited scope poses challenges when it comes to applying these mechanisms to complex software scenarios with wider attack surfaces. To overcome this limitation, it is crucial to develop CFA mechanisms specifically tailored for complex software.

**CFA Evidence Verification & Practicality.** The majority of CFA literature focuses on $\mathcal{P}$rv, assuming a $\mathcal{V}$rf can interpret received evidence to detect attacks and identify root causes as long as the evidence is sufficiently expressive. Alas, there is a significant lack of concrete $\mathcal{V}$rf instances to substantiate postulated evidence analysis capabilities. Most of the CFA literature either leaves $\mathcal{V}$rf implementation as future work or implements simple remote checks based on received evidence, e.g., adherence to a CFG or emulated shadow stack (both of these could also be done locally by several CFI methods and still face the complex challenge of validating non-deterministic forward-edges). Only two studies have explored alternative approaches to simple remote checks. ZEKRA [168] suggests generating a zero-knowledge proof of CFG adherence for an untrusted $\mathcal{V}$rf, while RAGE [169] proposes training a Graph Neural Network (GNN) on previous runtime evidence for path verification. Yet, thus far, no prior work has concretely demonstrated CFA's postulated benefits in uncovering complex attacks (and their root causes) based on remotely analyzed evidence. Additionally, striking a balance between evidence expressiveness and overhead poses a challenge in achieving full-fledged CFA. Hashed paths compromise detailed runtime evidence in exchange for reduced storage and transmission costs. However, lossless path representations (and associated transmission to $\mathcal{V}$rf) remain costly. As the complexity of the applications increases, the importance of expressiveness/cost trade-offs becomes more pronounced. Within this realm, promising avenues for future work include the development of mechanisms to reduce evidence storage and transmission costs while maintaining relevance and expressiveness. In this direction, recent work in Spec-CFA [148] proposes architectural support for $\mathcal{V}$rf-defined application-specific optimizations based on likely control flow sub-paths, enabling reduced storage/transmission costs without compromising evidence expressiveness.

**CFI in Real-Time Systems and Other Niche.** Most CFI proposals in Table 1 are not well-suited for real-time systems where strict timing requirements and execution integrity must be simultaneously maintained. Several recent CFI proposals focus on this gap [170]–[174]. InsectACIDE [170] uses architectural support (ARM TrustZone and MTB) to record control flow events without adding intra-task delays. During idle periods in between the execution of tasks, InsectACIDE uses the recorded information to perform security checks and locally detect control flow violations. FastCFI [171] also uses ARM features for tracing control flow transfers but relies on Field Programmable Gate Arrays (FPGA) to store and traverse a CFG according to these transfers. ECFI [174] proposes a mechanism for real-time Programmable Logic Controllers (PLCs) in which indirect branches are instrumented, and a PLC's OS can schedule CFI checks. In ECFI, CFI checks are assigned a lower priority so that PLC tasks maintain their Worst-Case Execution Time (WCET). RECFISH [172] proposes CFI for ARM MCUs executing both bare-metal software or applications atop FreeRTOS. RECFISH utilizes SWI to insert trampolines to functionality enforcing indirect branch destinations according to function label sets or a shadow stack. When applied to FreeRTOS, RECFISH also saves the task's state to the shadow stack to ensure that CFI-critical data cannot be overwritten during a context switch.

Another challenge involves rethinking the conventional approach of terminating an exploited application upon detecting a CFI violation, especially in domains such as autonomous systems. Abruptly terminating an application can introduce system instability or disruptions, posing risks to critical operations. Current proposals for CFI in real-time systems generally focus on minimizing WCET while supporting local detection rather than delving into post-detection recovery strategies [170]–[172]. One alternative approach in ECFI [174] makes killing the violating process configurable via an optional flag and, by default, stores a log file of the violation details. Alternative strategies could be explored to recover from CFI violations and ensure system safety without unintended consequences. A promising direction to address this challenge is to design CFI schemes accommodating multi-variant execution that allows the containment of exploited applications while enabling the continuation of critical tasks. We note that devising CFI that tackles both aforementioned challenges is non-trivial and requires more careful consideration. This includes accounting for factors such as portability, adaptability, and scalability.

## Acknowledgements

The authors sincerely thank the anonymous shepherd and the anonymous reviewers for their guidance and constructive criticism.

## References

[1] TIOBE, "TIOBE Index for May 2023 of Programming Languages," https://www.tiobe.com/tiobe-index/, 2023, [Online; accessed 13-May-2023].

[2] Microsoft, "Data Execution Prevention," https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention, 2022, [Online; accessed 13-February-2023].

[3] H. Löhr *et al.*, "Patterns for secure boot and secure storage in computer systems," in *ARES*. IEEE, 2010.

[4] W. A. Arbaugh *et al.*, "A secure and reliable bootstrap architecture," in *S&P*. IEEE, 1997.

[5] LWN.net, "The Integrity Measurement Architecture," https://lwn.net/Articles/137306/, 2005, [Online; accessed 13-May-2024].

[6] M. Bires, "Upgrading Android Attestation: Remote Provisioning," https://android-developers.googleblog.com/2022/03/upgrading-android-attestation-remote.html, 2022, [Online; accessed 13-May-2024].

[7] P. Larsen *et al.*, *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018.

[8] R. Roemer *et al.*, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[9] T. Bletsch *et al.*, "Jump-oriented programming: a new class of code-reuse attack," in *CCS*, 2011.

[10] S. Chen *et al.*, "Non-control-data attacks are realistic threats." in *USENIX Security*, 2005.

[11] H. Hu *et al.*, "Data-oriented programming: On the expressiveness of non-control data attacks," in *S&P*. IEEE, 2016.

[12] M. Payer, "Control-flow hijacking: Are we making progress?" in *AsiaCCS*, 2017.

[13] L. Szekeres *et al.*, "Sok: Eternal war in memory," in *S&P*. IEEE, 2013.

[14] M. Abadi *et al.*, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[15] N. Burow *et al.*, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, 2017.

[16] R. De Clercq *et al.*, "A survey of hardware-based control flow integrity (cfi)," *arXiv preprint arXiv:1706.07257*, 2017.

[17] X. Xu *et al.*, "Confirm: Evaluating compatibility and relevance of control-flow integrity protections for modern software." in *USENIX Security*, 2019.

[18] C. Tice *et al.*, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX Security*, 2014.

[19] N. Carlini *et al.*, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security*, 2015.

[20] ARM, "Learn the architecture - Providing protection for complex software," https://developer.arm.com/documentation/102433/0100, 2022, [Online; accessed 18-February-2023].

[21] Tom Garrison, "Intel CET Answers Call to Protect Against Common Malware Threats," https://newsroom.intel.de/editorials/intel-cet-answers-call-to-protect-against-common-malware-threats/, 2020, [Online; accessed 13-February-2023].

[22] A. J. Gaidis *et al.*, "Fineibt: Fine-grain control-flow enforcement with indirect branch tracking," *arXiv preprint arXiv:2303.16353*, 2023.

[23] Apple, "Operating System Integrity," https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/web, 2021, [Online; accessed 18-February-2023].

[24] A. Sharma, "This new Android 14 feature may be meant for the Pixel 8," https://www.androidauthority.com/android-14-advanced-memory-protection-3281197/, 2023, [Online; accessed 18-February-2023].

[25] H. Liljestrand *et al.*, "Pac it up: Towards pointer integrity using arm pointer authentication." in *USENIX Security*, 2019.

[26] S. Yoo *et al.*, "In-kernel control-flow integrity on commodity oses using arm pointer authentication," in *USENIX Security*, 2022.

[27] H. Liljestrand *et al.*, "Pacstack: an authenticated call stack." in *USENIX Security*, 2021.

[28] G. Serra *et al.*, "Pac-pl: Enabling control-flow integrity with pointer authentication in fpga soc platforms," in *RTAS*. IEEE, 2022.

[29] H. Liljestrand *et al.*, "Color my world: Deterministic tagging for memory safety," *arXiv preprint arXiv:2204.03781*, 2022.

[30] J. Ravichandran *et al.*, "Pacman: attacking arm pointer authentication with speculative execution," in *ISCA*, 2022.

[31] B. Azad, "Examining pointer authentication on the iphone xs, 2019," *URl: https://googleprojectzero. blogspot. com/2019/02/examining-pointer-authentication-on. html (visited on 07/27/2021)*, 2021.

[32] ——, "iOS Kernel PAC, One Year Later," https://bazad.github.io/presentations/BlackHat-USA-2020-iOS_Kernel_PAC_One_Year_Later.pdf, 2020, [Online; accessed 18-February-2023].

[33] T. Abera *et al.*, "C-flat: control-flow attestation for embedded systems software," in *CCS*, 2016.

[34] I. D. O. Nunes *et al.*, "Tiny-cfa: A minimalistic approach for control-flow attestation using verified proofs of execution," *DATE*, 2021.

[35] G. Dessouky *et al.*, "Lo-fat: Low-overhead control flow attestation in hardware," in *DAC*, 2017.

[36] S. Zeitouni *et al.*, "Atrium: Runtime attestation resilient under memory attacks," in *ICCAD*. IEEE, 2017.

[37] G. Dessouky *et al.*, "Litehax: lightweight hardware-assisted attestation of program execution," in *ICCAD*. IEEE, 2018.

[38] Z. Sun *et al.*, "Oat: Attesting operation integrity of embedded devices," in *S&P*. IEEE, 2020.

[39] N. Yadav *et al.*, "Whole-program control-flow path attestation," in *CCS*, 2023.

[40] J. Wang *et al.*, "Ari: Attestation of real-time mission execution integrity," 2023.

[41] A. Caulfield *et al.*, "Acfa: Secure runtime auditing & guaranteed device healing via active control flow attestation," in *USENIX Security*. USENIX, 2023.

[42] ——, "Traces: Tee-based runtime auditing for commodity embedded systems," 2024. [Online]. Available: https://arxiv.org/abs/2409.19125

[43] A. J. Neto *et al.*, "Isc-flat: On the conflict between control flow attestation and real-time operations," in *RTAS*. IEEE, 2023.

[44] A. Azevedo de Amorim *et al.*, "The meaning of memory safety," in *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. Springer, 2018, pp. 79–105.

[45] H. Xu *et al.*, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–25, 2021.

[46] B. Qin *et al.*, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 763–779.

[47] D. Midi *et al.*, "Memory safety for embedded devices with nescheck," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 127–139.

[48] N. D. Matsakis *et al.*, "The rust language," *ACM SIGAda Ada Letters*, 2014.

[49] A. S. Elliott *et al.*, "Checked c: Making c safe by extension," in *SecDev*. IEEE, 2018.

[50] T. Nyman *et al.*, "Toward hardware-assisted run-time protection," 2020.

[51] R. N. Watson *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *S&P*. IEEE, 2015.

[52] R. Wahbe *et al.*, "Efficient software-based fault isolation," in *SOSP*, 1993.

[53] H. Shacham *et al.*, "On the effectiveness of address-space randomization," in *CCS*, 2004.

[54] D. Song *et al.*, "Sok: Sanitizing for security," in *S&P*. IEEE, 2019.

[55] P. Larsen *et al.*, "Sok: Automated software diversity," in *S&P*. IEEE, 2014.

[56] N. Burow *et al.*, "Sok: Shining light on shadow stacks," in *S&P*. IEEE, 2019.

[57] L. Cheng *et al.*, "Exploitation techniques for data-oriented attacks with existing and potential defense approaches," *ACM Transactions on Privacy and Security (TOPS)*, 2021.

[58] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, 2020.

[59] Microsoft, "Secure the Windows Boot Process," https://learn.microsoft.com/en-us/windows/security/operating-system-security/system-security/secure-the-windows-10-boot-process, 2023, [Online; accessed 18-February-2024].

[60] Z. Tao *et al.*, "DICE*: A Formally Verified Implementation of DICE Measured Boot," in *USENIX Security*, 2021.

[61] A. Steffen, "The linux integrity measurement architecture and tpm-based network endpoint assessment," *Linux Security Summit*, 2012.

[62] A. Holdings, "Armv8 architecture reference manual for a-profile architecture," 2022.

[63] Y. Chen *et al.*, "Norax: Enabling execute-only memory for cots binaries on aarch64," in *S&P*. IEEE, 2017.

[64] V. Kuznetzov *et al.*, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018.

[65] K. Serebryany *et al.*, "Memory tagging and how it improves c/c++ memory safety," *arXiv preprint arXiv:1802.09517*, 2018.

[66] G. S. Kc *et al.*, "Countering code-injection attacks with instruction-set randomization," in *CCS*, 2003.

[67] P. Rajasekaran *et al.*, "CoDaRR: Continuous Data Space Randomization against Data-only Attacks," in *AsiaCCS*, 2020.

[68] M. Castro *et al.*, "Securing software by enforcing data-flow integrity," in *OSDI*, 2006.

[69] Y. Li *et al.*, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *CCS*, 2020.

[70] S. Han *et al.*, "Page-oriented programming: Subverting control-flow integrity of commodity operating system kernels with non-writable code pages," in *USENIX Security*, 2024.

[71] PaX Team, "Non-Executable Pages Design & Implementation," https://pax.grsecurity.net/docs/noexec.txt, 2003, [Online; accessed 18-February-2023].

[72] I. D. O. Nunes *et al.*, "VRASED: A verified Hardware/Software Co-Design for remote attestation," in *USENIX Security*, 2019.

[73] M. Ammar *et al.*, "Simple: A remote attestation approach for resource-constrained iot devices," in *ICCPS*. IEEE, 2020.

[74] R. Sailer *et al.*, "Design and implementation of a tcg-based integrity measurement architecture." in *USENIX Security*, 2004.

[75] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2921–2936.

[76] I. D. O. Nunes *et al.*, "DIALED: Data Integrity Attestation for Low-end Embedded Devices," in *DAC*. IEEE, 2021.

[77] Trusted Computing Group, "Trusted Platform Module (TPM)," https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/, 2008, [Online; accessed 18-February-2024].

[78] ——, "Device Identifier Composition Engine (DICE)," https://trustedcomputinggroup.org/what-is-a-device-identifier-composition-engine-dice/, 2021, [Online; accessed 18-February-2024].

[79] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086, Tech. Rep.

[80] *ARM Security Technology - Building a Secure System using TrustZone Technology*, ARM Limited, 2009.

[81] A. Ltd, "Trustzone technology for armv8-m architecture version 2.1," https://developer.arm.com/documentation/100690/0201/, 2019.

[82] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[83] M. Armanuzzaman *et al.*, "Building your own trusted execution environments using fpga," in *AsiaCCS*, 2022.

[84] H. Hu *et al.*, "Enforcing unique code target property for control-flow integrity," in *CCS*, 2018.

[85] B. Niu *et al.*, "Per-input control-flow integrity," in *CCS*, 2015.

[86] K. Lu *et al.*, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *CCS*, 2019.

[87] Z. Ma *et al.*, "Return-to-non-secure vulnerabilities on arm cortex-m trustzone: Attack and defense," in *DAC*, 2023.

[88] M. Zhang *et al.*, "Control flow integrity for cots binaries," in *USENIX Security*, 2013.

[89] C. Zhang *et al.*, "Practical control flow integrity and randomization for binary executables," in *S&P*. IEEE, 2013.

[90] J. Criswell *et al.*, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *S&P*. IEEE, 2014.

[91] B. Niu *et al.*, "Modular control-flow integrity," in *PLDI*, 2014.

[92] A. J. Mashtizadeh *et al.*, "Ccfi: Cryptographically enforced control flow integrity," in *CCS*, 2015.

[93] M. Zhang *et al.*, "Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks," in *ACSAC*, 2015.

[94] V. Mohan *et al.*, "Opaque control-flow integrity." in *NDSS*, 2015.

[95] M. Payer *et al.*, "Fine-grained control-flow integrity through binary hardening," in *DIMVA*. Springer, 2015.

[96] V. Van Der Veen *et al.*, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *S&P*. IEEE, 2016.

[97] X. Ge *et al.*, "Fine-grained control-flow integrity for kernel software," in *EuroS&P*. IEEE, 2016.

[98] ARM, "BTI," https://developer.arm.com/documentation/ddi0602/2021-12/Base-Instructions/BTI--Branch-Target-Identification-, 2020, [Online; accessed 13-February-2023].

[99] ——, "Return Address Signing using ARM Pointer Authentication," https://gcc.gnu.org/legacy-ml/gcc-patches/2018-11/msg00104.html, 2018, [Online; accessed 13-February-2023].

[100] M. Ismail *et al.*, "Tightly seal your sensitive pointers with {PACTight}," in *USENIX Security*, 2022.

[101] N. Carlini *et al.*, "Rop is still dangerous: Breaking modern defenses," in *USENIX Security*, 2014.

[102] L. Davi *et al.*, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security*, 2014.

[103] E. Göktas *et al.*, "Out of control: Overcoming control-flow integrity," in *S&P*. IEEE, 2014.

[104] E. Göktaş *et al.*, "Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard," in *USENIX Security*, 2014.

[105] F. Schuster *et al.*, "Evaluating the effectiveness of current anti-rop defenses," in *RAID*. Springer, 2014.

[106] V. Van der Veen *et al.*, "Practical context-sensitive cfi," in *CCS*, 2015.

[107] R. Ding *et al.*, "Efficient Protection of Path-Sensitive Control Security," in *USENIX Security*, 2017.

[108] M. Werner *et al.*, "Sponge-based control-flow protection for iot devices," in *EuroS&P*. IEEE, 2018.

[109] M. R. Khandaker *et al.*, "Origin-sensitive control flow integrity," in *USENIX Security*, 2019.

[110] M. Khandaker *et al.*, "Adaptive call-site sensitive control flow integrity," in *EuroS&P*. IEEE, 2019.

[111] N. S. Almakhdhub *et al.*, "$\mu$RAI: Securing Embedded Systems with Return Address Integrity," in *NDSS*, 2020.

[112] M. Ismail *et al.*, "Vip: safeguard value invariant property for thwarting critical memory corruption attacks," in *CCS*, 2021.

[113] Intel, "Intel Processor Trace," https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/010/intel-processor-trace/, 2015, [Online; accessed 13-February-2024].

[114] X. Tan *et al.*, "Sherloc: Secure and holistic control-flow violation detection on embedded systems," in *CCS*, 2023.

[115] I. D. O. Nunes *et al.*, "{APEX}: A verified architecture for proofs of execution on remote devices under full software compromise," in *USENIX Security*, 2020.

[116] D. Papamartzivanos *et al.*, "Towards efficient control-flow attestation with software-assisted multi-level execution tracing," in *MeditCom*. IEEE, 2021.

[117] G. Ramalingam, "The undecidability of aliasing," *TOPLAS*, 1994.

[118] R. Baldoni *et al.*, "A survey of symbolic execution techniques," *CSUR*, 2018.

[119] F. Toffalini *et al.*, "{ScaRR}: Scalable runtime remote attestation for complex systems," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019.

[120] M. Ammar *et al.*, "On bridging the gap between control flow integrity and attestation schemes," in *USENIX Security*, 2024.

[121] Microsoft, "Control Flow Guard for platform security," https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2022, [Online; accessed 13-February-2023].

[122] J. Zhou *et al.*, "Silhouette: Efficient protected shadow stacks for embedded systems," in *USENIX Security*, 2020.

[123] D. Kuzhiyelil *et al.*, "Towards transparent control-flow integrity in safety-critical systems," in *ISC*. Springer, 2020.

[124] M. Geden *et al.*, "Hardware-assisted remote runtime attestation for critical embedded systems," in *PST*. IEEE, 2019.

[125] Y. Zhang *et al.*, "Recfa: resilient control-flow attestation," in *ACSAC*, 2021.

[126] M. Morbitzer *et al.*, "Guarantee: Introducing control-flow attestation for trusted execution environments," *arXiv preprint arXiv:2202.07380*, 2022.

[127] N. Christoulakis *et al.*, "Hcfi: Hardware-enforced control-flow integrity," in *ACM CODASPY*, 2016.

[128] X. Ge *et al.*, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, 2017.

[129] M. Bauer *et al.*, "Typro: Forward cfi for c-style indirect function calls using type propagation," in *ACSAC*, 2022.

[130] L. Davi *et al.*, "Hafix: Hardware-assisted flow integrity extension," in *DAC*, 2015.

[131] LLVM Community, "Safe Stack," https://clang.llvm.org/docs/SafeStack.html, 2014, [Online; accessed 18-February-2023].

[132] ——, "Shadow Call Stack," https://clang.llvm.org/docs/ShadowCallStack.html, 2012, [Online; accessed 18-February-2023].

[133] Y. Cheng *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," in *NDSS*, 2014.

[134] V. Pappas, "kbouncer: Efficient and transparent rop mitigation," *Apr*, 2012.

[135] G. F. Roglia *et al.*, "Surgically returning to randomized lib (c)," in *ACSAC*. IEEE, 2009.

[136] T. Nyman *et al.*, "Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers," in *RAID*. Springer, 2017.

[137] B. Niu *et al.*, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *CCS*, 2014.

[138] Z. Lin *et al.*, "Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis," in *CCS*, 2023.

[139] L. Maar *et al.*, "Beyond the edges of kernel control-flow hijacking protection with hek-cfi," in *AsiaCCS*, 2024.

[140] T. Abera *et al.*, "Diat: Data integrity attestation for resilient collaboration of autonomous systems." in *NDSS*, 2019.

[141] O. Arias *et al.*, "Lahel: Lightweight attestation hardening embedded devices using macrocells," in *HOST*. IEEE, 2020.

[142] D. Huo *et al.*, "Lape: A lightweight attestation of program execution scheme for bare-metal systems," in *HPCC/SmartCity/DSS*. IEEE, 2020.

[143] Y. Gu *et al.*, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *ACM CODASPY*, 2017.

[144] V. Pappas *et al.*, "Transparent ROP: exploit mitigation using indirect branch tracing," in *USENIX Security*, 2013.

[145] Intel, "Intel 64 and ia-32 architectures software developer's manual," 2018.

[146] E. Aliaj *et al.*, "Garota: generalized active root-of-trust architecture," *arXiv preprint arXiv:2102.07014*, 2021.

[147] T. Frassetto *et al.*, ""cfinsight: A comprehensive metric for cfi policies"," in *NDSS*, 2022.

[148] A. Caulfield *et al.*, "Speccfa: Enhancing control flow attestation/auditing via application-aware sub-path speculation," 2024. [Online]. Available: https://arxiv.org/abs/2409.18403

[149] F. Schuster *et al.*, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *S&P*. IEEE, 2015.

[150] R. Gawlik *et al.*, "Towards automated integrity protection of c++ virtual function tables in binary programs," in *ACSAC*, 2014.

[151] C. Zhang *et al.*, "Vtint: Defending virtual function tables' integrity," in *NDSS*, 2015.

[152] M. Theodorides *et al.*, "Breaking active-set backward-edge cfi," in *HOST*. IEEE, 2017.

[153] I. Evans *et al.*, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *CCS*, 2015.

[154] V. Duta *et al.*, "Let me unwind that for you: Exceptions to backward-edge protection." in *NDSS*, 2023.

[155] M. Conti *et al.*, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *CCS*, 2015.

[156] J. Xu *et al.*, "Warpattack: bypassing cfi through compiler-introduced double-fetches," in *S&P*. IEEE, 2023.

[157] X. Carpent *et al.*, "Temporal consistency of integrity-ensuring computations and applications to embedded systems security," in *AsiaCCS*, 2018.

[158] I. De Oliveira Nunes *et al.*, "On the toctou problem in remote attestation," in *CCS*, 2021.

[159] S. Hristozov *et al.*, "A toctou attack on dice attestation," in *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*, 2022, pp. 226–235.

[160] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, 2020.

[161] Y. Yarom *et al.*, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *USENIX Security*, 2014.

[162] A. Mambretti *et al.*, "Bypassing memory safety mechanisms through speculative control flow hijacks," in *EuroS&P*. IEEE, 2021.

[163] E. M. Koruyeh *et al.*, "Speccfi: Mitigating spectre attacks using cfi informed speculation," in *S&P*. IEEE, 2020.

[164] H. Jang *et al.*, "Microcfi: Microarchitecture-level control-flow restrictions for spectre mitigation," *IEEE Access*, 2023.

[165] V. Shanbhogue *et al.*, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *HASP*, 2019.

[166] S. Wiebing *et al.*, "Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2," in *USENIX Security*, 2024.

[167] R. Hund *et al.*, "Practical timing side channel attacks against kernel space aslr," in *S&P*. IEEE, 2013.

[168] H. B. Debes *et al.*, "Zekra: Zero-knowledge control-flow attestation," in *AsiaCCS*, 2023.

[169] M. Chilese *et al.*, "One for all and all for one: Gnn-based control-flow attestation for embedded devices," *arXiv preprint arXiv:2403.07465*, 2024.

[170] Y. Wang *et al.*, "Insectacide: Debugger-based holistic asynchronous cfi for embedded system," in *RTAS*. IEEE, 2024.

[171] L. Feng *et al.*, "Fastcfi: Real-time control-flow integrity using fpga without code instrumentation," *TODAES*, 2021.

[172] R. J. Walls *et al.*, "Control-flow integrity for real-time embedded systems," in *ECRTS*, 2019.

[173] M. Kadar *et al.*, "Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring," in *RTAS*. IEEE, 2021.

[174] A. Abbasi *et al.*, "Ecfi: Asynchronous control flow integrity for programmable logic controllers," in *ACSAC*, 2017.

# META-REVIEW

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as a part of the review process as detailed in the call for papers.

## Summary

This paper provides an SoK on CFI and CFA. The work is motivated by the increasing threat of memory corruption vulnerabilities and the wide scope of defenses that have been proposed to mitigate them. The defenses fall under two categories, CFI and CFA, but are riddled with various goals, assumptions, and implementation weaknesses. This paper disentangles the literature to provide a unified view of the two approaches, their strengths and weaknesses, and promising directions for future research.

## Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Provides a Valuable Step Forward in Established Field

## Reasons for Acceptance

1) The community is in need of a comparison and deep understanding of the difference and gap of CFI and CFA.
2) The paper does a great job at comparing and contrasting CFI and CFA and positioning them within the broader scope of runtime defenses.
3) It also does a good job of highlighting promising avenues for future work, including problem domains, high-level abstractions, and low-level techniques.