SoK: A Practical Guideline and Taxonomy to LLVM's Control Flow Integrity

Sabine Houy
Umeå University
sabine.houy@umu.se

Bruno Kreyssig
Umeå University
bruno.kreyssig@umu.se

Timothée Riom
Umeå University
timothee.riom@umu.se

Alexandre Bartel
Umeå University
alexandre.bartel@cs.umu.se

Patrick McDaniel
University of Wisconsin-Madison
mcdaniel@cs.wisc.edu

***Abstract*—Memory corruption vulnerabilities remain one of the most severe threats to software security. They often allow attackers to achieve arbitrary code execution by redirecting a vulnerable program's control flow. While Control Flow Integrity (CFI) has gained traction to mitigate this exploitation path, developers are not provided with any direction on how to apply CFI to real-world software. In this work, we establish a taxonomy mapping LLVM's forward-edge CFI variants to memory corruption vulnerability classes, offering actionable guidance for developers seeking to deploy CFI incrementally in existing codebases. Based on the Top 10 Known Exploited Vulnerabilities (KEV) list, we identify four high-impact vulnerability categories and select one representative CVE for each. We evaluate LLVM's CFI against each CVE and explain why CFI blocks exploitation in two cases while failing in the other two, illustrating its potential and current limitations. Our findings support informed deployment decisions and provide a foundation for improving the practical use of CFI in production systems.**

I. Introduction

Memory corruption vulnerabilities remain one of the most critical threats to software security, consistently ranking among the top vulnerabilities in various threat lists. For instance, one specific form of memory corruption is ranked as the second most dangerous issue in the *Most Dangerous Software Weaknesses* list [57], and memory corruption weaknesses occupy the top three spots in the *Top 10 KEV*[1] *Weaknesses* [56]. These rankings underline the urgent need for robust and effective defenses.

Control-Flow Integrity (CFI) is a well-established defense mechanism designed to protect against control-flow hijacking attacks. However, despite its theoretical effectiveness, the real-world adoption of CFI remains inconsistent. Operating systems like Android and Windows have successfully integrated CFI [4], [9], [42], but Linux struggles with widespread adoption. Similarly, open-source browsers, including Chromium [96] and Firefox [14], show limited support for CFI. One key challenge is that CFI can break functionality when policy violations occur, often triggered by factors such as C-standard mismatches or design decisions in legacy code. Additionally, while CFI research has produced various variants aimed at improving flexibility and efficiency [13], [15], [46], [68], [70], [78], [79], [82], there is still limited empirical data demonstrating how CFI performs against real-world exploits. This leaves developers uncertain about which CFI variant to deploy and how best to integrate it into existing projects.

The main CFI implementation applied in these cases is LLVM's; therefore, we focus on its seven CFI variants in this work. CFI was originally intended for the challenges faced by memory-unsafe languages, i.e., C and C++ [1]. LLVM also provides CFI for the "unsafe" code blocks in the otherwise memory-safe Rust language. While this highlights the ongoing relevance of CFI, in this work, we focus on the more imminent use case of memory-unsafe C/C++ code.

The main contribution of this work is a systematic guideline that maps distinct classes of memory corruption vulnerabilities to the CFI variants most likely to provide practical protection. This guideline aims to support developers in making informed decisions about CFI integration, addressing the challenges posed by CFI's incompatibility with existing software ecosystems [14], [41], [42], [96], [109]. To complement the guideline, we present four real-world exploits, each corresponding to a vulnerability type identified during its development. These examples help bridge the gap in the literature by showing where CFI mechanisms succeed or fail in practice, and offer insights into the reasons behind these outcomes. The examples are not meant to validate the guideline itself but serve to illustrate CFI's practical strengths and limitations in diverse contexts.

We propose a structured guideline that maps different types of memory corruption vulnerabilities to LLVM's CFI variants most likely to constrain control-flow manipulation. This mapping offers developers a practical reference for selecting CFI mechanisms that align with the specific vulnerabilities they aim to mitigate. To demonstrate the utility of the guideline, we select four recent high-impact CVEs[2] from different vulnerability classes. By analyzing these examples, we show how applying the appropriate CFI mechanisms could have prevented or significantly limited the exploitation of these vulnerabilities.

Our approach begins with a comprehensive review of prevalent memory corruption vulnerabilities to develop a structured taxonomy. This taxonomy serves as the foundation for mapping each vulnerability class to the CFI variants most likely to mitigate it. We then select four representative CVEs

[1]Known Exploited Vulnerabilities

[2]Common Vulnerabilities and Exposures

and compile the vulnerable software both with and without CFI enforcement. We test the publicly available PoCs/exploits against each compiled version to assess the impact of CFI on exploitability.

The real-world examples show that CFI mechanisms can successfully mitigate exploitation in some cases. In two out of four cases, CFI enforcement blocked exploit paths. In the remaining cases, CFI either partially restricted or failed to prevent exploitation, depending on the vulnerability type and CFI variant used. These examples underscore the need for developers to carefully consider the specific characteristics of the vulnerabilities they are targeting when applying CFI.

Main contributions:

1) We present a guideline that systematically maps LLVM CFI variants to memory corruption vulnerability types, offering developers a structured approach to CFI deployment.
2) We provide real-world examples that demonstrate when and why specific CFI variants succeed or fail in preventing exploitation. Additionally, we make the experimental environments and PoCs publicly available to facilitate reproducibility[3].

Our framework guides developers in selecting LLVM CFI variants that align with the specific memory corruption threats their applications face. By linking each variant to common vulnerability types, it helps prioritize protections based on application characteristics and deployment needs. Our findings support informed deployment decisions and provide a foundation for improving the practical use of CFI in production systems, especially by clarifying which variants are most relevant for mitigating specific classes of vulnerabilities.

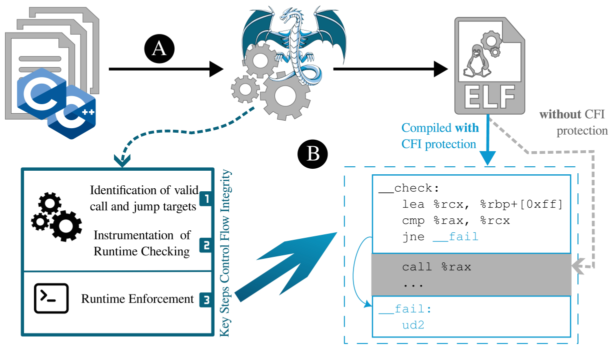
While CFI significantly complicates exploitation, it remains, like all mitigation techniques [17], [86], [87], ultimately bypassable [22], [30], [38], [51], [107]. Throughout this work, *prevent* refers to preventing specific exploit instances or triggers rather than achieving complete elimination of all vulnerability risks.

II. Background

A. Control Flow Integrity (CFI)

CFI is a fundamental defense mechanism against control-flow hijacking (CFH) attacks, such as Return-Oriented Programming (ROP) [25], [112], Jump-Oriented Programming (JOP) [21], and `vtable` corruption [12], [95]. CFI enforces that program control transfers adhere to a valid control flow graph (CFG), preventing attackers from redirecting execution to unintended code paths. Since many exploits rely on manipulating indirect control flow (e.g., function pointers, virtual table calls, type casting), various CFI variants exist to enforce integrity at different execution levels. CFI mechanisms are typically classified as either *coarse-grained* or *fine-grained*, depending on how strictly they restrict indirect control flow transfers. Coarse-grained CFI groups many legitimate targets into broad equivalence classes and allows indirect branching within that set. This approach is often vulnerable to advanced attacks that remain within these loose boundaries [18]. In contrast, fine-grained CFI uses more precise static analysis to limit targets based on context, type information, or the position of the call site, significantly reducing the attack surface [101]. As LLVM's CFI enforces type signature matching on the call side by verifying function prototypes, it is thus considered fine-grained [94], [101].

[3]Artifacts available at https://github.com/software-engineering-and-security/cfi-practical-guideline.git, including the resulting exploits and PoCs from our collection process (Section IV-A).


A
B
ELF
Compiled with CFI protection
without CFI protection
Identification of valid call and jump targets 1
Instrumentation of Runtime Checking 2
Runtime Enforcement 3
Key Steps Control Flow Integrity

```
__check:
  lea %rcx, %rbp+[0xff]
  cmp %rax, %rcx
  jne __fail
  call %rax
  ...
__fail:
  ud2
```

Fig. 1: Control Flow Integrity simplified example

Figure 1 illustrates the concept of CFI in a simplified example. Since CFI is a compiler-based mitigation technique, the compiler (in our case, LLVM clang/clang++ [94]) introduces the CFI-related assembly instructions. This requires two steps (A): First, the compiler identifies all valid call and jump targets (1). Then, it generates the Control Flow Graph (CFG), determining all potential vulnerable jumps and calls. Subsequently, it creates a set of all **valid** call and jump targets, meaning an equivalence set of all targets with the same signature as the intended destination. LLVM uses a type-based signature relying on the return and parameter types. Then, for each potentially vulnerable jump and call, clang adds binary instructions validating the call target before the actual call is executed (2). These additional checks are verified during runtime (3).

A simplified code example is shown in B. Only the `call %rax` instruction is part of the original code (no CFI enforcement). The code following the `__check` and `__fail` labels is the runtime enforcement, added in step (2). Register `%rax` contains the address of the to-be-performed call. CFI aims to verify that this address is valid within the CFG, meaning the signatures match. To ensure this, the predefined signature is loaded into register `%rcx`. Then, the `%rax`'s and `%rcx`'s values are compared, determining whether the signatures are identical. If the comparison is successful, the call is executed, and the program's control flow proceeds as intended. Otherwise, the program jumps to `__fail`, executing an undefined instruction (`ud2`) that leads to an `Illegal Instruction (SIGILL)` signal, causing the program to crash and preventing redirection to malicious code.

The CFG- and type-based CFI approaches have certain limitations. One key challenge is the *over-approximation* of legitimate control-flow destinations. For instance, if a function to be protected has a `void*` return type and accepts an `int` as a parameter, any function in the CFG that matches this pattern will be considered a valid target. This can lead to a significant number of potential valid targets, which may allow an excessive number of jumps or function calls, reducing the effectiveness of CFI in preventing CFH attacks. In static analysis and program verification, over-approximation is recognized as an open research challenge [18], [30], [90].

LLVM CFI combines CFG- and type-based implementation and offers four call-focused and three cast-focused variants, explained in II-A1 and II-A2 respectively.

1) CFI Variants for Function Call Integrity: **Non-virtual call CFI** (`cfi-nvcall`), ensures that function calls made through object pointers in non-virtual contexts cannot be redirected to unintended targets. Unlike indirect function calls, which resolve dynamically at runtime, non-virtual function calls are known at compile time. However, if an attacker corrupts an object pointer, execution may jump to arbitrary locations. Consider the following example depicted in Listing 1: an attacker could manipulate `obj` (line 23) to point to a maliciously crafted memory region, causing `nonVirtualMethod()` to redirect the execution flow to unintended code. CFI verifies that `obj` remains a valid `NonVirtualClass` object before execution, preventing unauthorized function calls [94], [101], [112].

Similarly, **virtual function call CFI** (`cfi-vcall`) ensures that virtual function calls always resolve to the correct method in a class's virtual table (`vtable`). C++ relies on `vtable`-based dynamic dispatch for polymorphism, making virtual function calls a prime target for attackers who overwrite `vtable` pointers [27], [84], [94].

```
1   #include <iostream>
2
3   class BaseClass {
4   public:
5       virtual void virtualMethod() {
6           std::cout << "BaseClass::virtualMethod\n";
7       }
8   };
9   class DerivedClass : public BaseClass {
10  public:
11      void virtualMethod() override {
12          std::cout << "DerivedClass::virtualMethod\n";
13      }
14  };
15  class NonVirtualClass {
16  public:
17      void nonVirtualMethod() {
18          std::cout << "Legitimate function call\n";
19      }
20  };
21  // cfi-nvcall
22  void callNonVirtualMethod(NonVirtualClass* obj) {
23      obj->nonVirtualMethod();
24  }
25  // cfi-vcall
26  void callVirtualMethod(BaseClass* obj) {
27      obj->virtualMethod();
28  }
```

Listing 1: C++ code example clarifying functionality of `cfi-nvcall` and `cfi-vcall`.

In Listing 1 line 27, if `obj` 's virtual table pointer is overwritten, an attacker could redirect execution to an arbitrary function. CFI ensures that the virtual function call remains within the valid class hierarchy, mitigating `vtable` hijacking attacks [15], [38]. In other words, CFI would determine that the `virtualMethod` in `BaseClass` (line 5) and the overridden one in `DerivedClass` (line 11) are valid call targets within `obj`'s `vtable`.

Another common target is indirect function calls, which are protected by **indirect function call CFI** (`cfi-icall`). Indirect calls occur via function pointers, making them a prime vector for hijacking attacks [1], [94].

```
void safeFunction() {
    std::cout << "Safe function executed\n";
}
void callFunction(void (*func)()) {
    func();
}
int main() {
    void (*functionPointer)() = safeFunction;
    callFunction(functionPointer);
}
```

Listing 2: C++ code example clarifying functionality of `cfi-icall`.

If an attacker overwrites `functionPointer` with a rogue address, shown in Listing 2, execution may jump to malicious shellcode, ROP gadgets, or system functions. CFI restricts `functionPointer` to known valid function addresses within the program's CFG, blocking control flow hijacking [101], [112].

A related mechanism, **member-function call CFI** (`cfi-mfcall`), protects member function pointers in C++. Unlike "*regular*" function pointers, member function pointers require both a valid object and a valid method, increasing the attack surface [49], [94], [113].

```
void callMemberFunction(BaseClass* obj,
            void (BaseClass::*func)()) {
    (obj->*func)();
}
int main() {
        DerivedClass obj;
        void (BaseClass::*functionPointer)() =
           &BaseClass::memberFunction;
        callMemberFunction(&obj, functionPointer);
}
```

Listing 3: C++ code example clarifying functionality of `cfi-mfcall`.

If an attacker corrupts `functionPointer` or `obj`, see Listing 3, execution may be redirected to arbitrary memory locations. CFI ensures that `functionPointer` remains within the correct class hierarchy, preventing exploits that target `vtable` corruption.

While both `cfi-icall` and `cfi-nvcall` enforce function call integrity, they protect against distinct attack vectors. `cfi-icall` applies to indirect function calls via function pointers. Since function pointers resolve dynamically, attackers can overwrite them to execute arbitrary code. CFI ensures that only legitimate function addresses are called, blocking hijacking attempts. `cfi-nvcall` applies to non-virtual function calls, typically resolved at compile time. However, if an attacker corrupts an object pointer, the function call could target unintended memory regions. CFI prevents this by ensuring the object pointer remains valid in its expected class.

2) CFI Variants for Type Casting Protection: Beyond function calls, CFI also enforces type-casting constraints to prevent type confusion and object corruption attacks. The `cfi-unrelated-cast` variant ensures that **casts between unrelated types** are prohibited, preventing memory corruption vulnerabilities which affect the control flow [84], [94].

```cpp
int number = 42;
float* invalidPointer =
    reinterpret_cast<float*>(&number);
```

Listing 4: C++ code example clarifying functionality of `cfi-unrelated-cast`.

If an attacker misuses `reinterpret_cast`, they may introduce data corruption or arbitrary code execution. CFI blocks such casts at runtime, ensuring type safety.

In contrast, `cfi-derived-cast` ensures that **casting between *base* and *derived* classes** remains valid, mitigating type confusion attacks [94], [112].

```cpp
void testCast(BaseClass* basePtr) {
    DerivedClass* derivedPtr =
        dynamic_cast<DerivedClass*>(basePtr);
}
```

Listing 5: C++ code example clarifying functionality of `cfi-derived-cast`.

An unsafe cast may occur if `basePtr` is manipulated to point to an unrelated object; see Listing 5. CFI prevents such abuses by validating inheritance relationships. A stricter enforcement mechanism, `cfi-cast-strict`, extends `cfi-derived-cast` by covering cases where a derived class introduces no new virtual functions and has an identical memory layout to its base class [15].

B. CFI Relevant Memory Corruption Vulnerabilities

CFI is specifically designed to prevent CFH attacks. The most relevant vulnerabilities to CFI are those that allow attackers to directly divert the program's control flow, typically by corrupting code pointers, return addresses, or function calls. There are many different types and subcategories of memory corruption vulnerabilities. In general, CFI can only protect against attacks that lead to a modified control flow. We choose the vulnerabilities based on the current Top 10 KEV list, ranking *use-after-free* and *heap-based buffer overflows* first and second, respectively. Place three is "Out-of-bounds Write", the heap- and stack-based buffer overflows parent category, thus adding *stack-based buffer overflows*. *Type confusion* reaches rank number eight. All other listed vulnerabilities cannot directly corrupt the memory (e.g, Server-Side Request Forgery -SSRF) and are thus not directly relevant to CFI and the scope of this paper. However, they could affect the memory by triggering, for instance, a buffer overflow, which would be covered by *heap- stack-based buffer overflows*.

1) Stack-based Buffer Overflow: A stack-based buffer overflow is a common vulnerability in C and C++ programs. It arises when data is written beyond the boundaries of a buffer allocated on the stack. This issue often occurs when functions such as `strcpy`, `sprintf`, or `gets` are used without proper bounds checking, enabling more data to be written than the buffer can accommodate. Consequently, this excess data can overwrite adjacent memory on the stack, including critical control information, e.g., return addresses, function pointers, and local variables [10], [72], [104]. This leads to unpredictable behavior, crashes, or, in some cases, exploitation by attackers who deliberately craft inputs to overwrite control structures, redirecting execution to malicious code. Stack-based buffer overflows are particularly dangerous because they can be exploited for arbitrary code execution, privilege escalation, or denial-of-service attacks [105].

Modern compiler protections and operating systems offer enhanced defense mechanisms, including stack canaries, Address Space Layout Randomization (ASLR), and non-executable stacks. However, as all these techniques are bypassable in different ways [11], [31], [45], [81], it is essential to use multiple at once and add as many security layers as possible. An additional layer of security, such as enforcing CFI, can improve resistance to stack-based buffer overflows [23], [104], [105].

2) Heap-based Buffer Overflow: A heap-based buffer overflow occurs when a buffer allocated in the heap overflows, resulting in the overwriting of adjacent memory regions. This can corrupt heap structures, alter program data, or facilitate arbitrary code execution. A successful attack may overwrite return addresses or critical metadata, allowing attackers to redirect control flow. Heap-based buffer overflows typically arise when functions like `malloc`, `calloc`, or `realloc` are used to allocate memory, and the data is written into the buffer without adequately checking its size. This oversight can cause the buffer to overflow, potentially corrupting adjacent heap structures or metadata used by the memory allocator. Attackers can exploit these vulnerabilities to manipulate the heap layout, overwrite critical data, such as function pointers or object metadata, and achieve arbitrary code execution or privilege escalation [17], [48], [110].

Additionally, runtime protections like heap hardening mechanisms, memory sanitizers, and allocators with built-in integrity checks, such as LLVM's CFI, offer necessary safeguards [48].

3) Use-After-Free (UAF): A use-after-free (UAF) vulnerability arises when a program continues to access memory after it has been freed, resulting in undefined behavior. This allows attackers to manipulate freed memory for malicious purposes, such as executing arbitrary code or causing crashes. UAF vulnerabilities are particularly prevalent in languages like C and C++, where memory management is manual. These vulnerabilities occur when a pointer continues to reference a memory location after it has been freed using `free` in C or `delete` in C++. If the pointer is not nullified, it becomes a "dangling pointer", which can be dereferenced to access reallocated or corrupted memory, leading to potential exploits [20], [40], [54].

Techniques like CFI can help detect and prevent such vulnerabilities by ensuring valid control flow paths.

4) Type Confusion: A type confusion vulnerability occurs when a program mistakenly treats an object in memory as a different type than it is, often leading to unsafe memory operations. This can result in undefined behavior, program crashes, or allow attackers to execute arbitrary code. Type confusion vulnerabilities are common in object-oriented programming and commonly lead to type mismatches in virtual tables (`vtable`), altering the intended control flow. In C and C++, type confusion typically arises when an object of one type is misinterpreted as another, often due to improper type casting or unsafe memory manipulation. For example, if a pointer to an object of type `A` is cast to a pointer of type `B`, the program may misinterpret the underlying memory, resulting in undefined behavior or security breaches [47], [114]. Attackers can exploit this vulnerability to corrupt memory, bypass security checks, or gain control of the program's execution path.

Static analyzers, runtime type verification tools, and techniques like Control Flow Integrity are also critical in ensuring that type assumptions are correctly enforced.

III. Implementation of Systematization

In the following, we discuss to what extent LLVM's CFI implementation can potentially mitigate the mentioned vulnerabilities (described in detail in Section II).

CFI protects against specific types of *buffer overflows* if data related to the control flow is corrupted. If a heap or stack corruption modifies data fields unrelated to control flow, CFI cannot detect the exploit when no protected function or `vtable` pointers are corrupted on the way. In the case of *use-after-free (UAF)*, CFI can mitigate vulnerabilities if a freed object's `vtable` or function pointers are reused inappropriately. CFI can effectively only prevent *type confusion* from malicious casts or directly altering function pointers, thus resulting in a mismatched type.

To summarize, CFI checks are effective against memory corruptions that involve altering function pointers or `vtables`. In the following section, we elaborate on how LLVM's implementation of CFI can affect the four top 10 KEV vulnerabilities.

A. Mapping of Relevant Memory Corruption Vulnerabilities to CFI Types

The following explains why specific CFI variants are tailored to specific threat models (types of memory corruption vulnerabilities) rather than being universal solutions. We provide a mapping for each vulnerability category and its expected effective CFI variants [1], [94], [101] (see Section II-A), summarized in Table II.

Our rating system uses a three-point scale: low (○), moderate (◑), and high (●) mitigation potential. It is important to note that *high* does not mean that CFI will protect against all exploits based on this vulnerability type. Rather, it indicates which CFI variants most effectively block common exploitation vectors of the respective vulnerabilities. The overall score for each vulnerability is calculated as the mean of all CFI variants.

Each variant is assigned one of these levels based on an approximation based on a summary of facts extracted from existing scientific literature. We performed a keyword search (based on the considered vulnerabilities and CFI) in 2024 in major scientific library databases (such as ACM and IEEE) to find relevant literature. Then, we filtered based on the title and then on the abstract. Thus, literature focusing solely on other languages, such as Java, would be excluded. When identifying relevant work, we performed backward- and forward-referencing to find additional literature.

TABLE I: Literature used for Mapping

Vulnerability Type	Literature
Heap-based Buffer Overflow	[1], [3], [15], [22], [23], [38], [48], [59], [74], [83], [94], [101], [110]
Stack-based Buffer Overflow	[1], [3], [11], [15], [23], [55], [58], [63], [74], [76], [83], [94], [101], [108], [111]
Use-After-Free (UAF)	[1], [3], [22], [48], [50], [51], [60], [75], [83], [94], [101], [110]
Type Confusion	[7], [23], [28], [33], [34], [47], [51], [61], [62], [77], [94], [101], [114]

The resulting literature is depicted in Table I. In addition, we provide a code book containing direct quotes of the relevant literature used to create the mapping, available as artifacts[3]. The extracted quotes do not mean that they are the only relevant ones, but the most relevant ones.

1) Heap-based Buffer Overflow: Heap overflows can overwrite function pointers, vtable pointers, and other heap-resident control-flow structures. As several works highlight forward-edge hijacks, LLVM's `icall`[4] and `vcall` CFI are particularly relevant.

● *High* – `icall`, `vcall`: Tice et al. [101] and Sayeed et al. [83] explicitly describe heap corruption of indirect call targets and `vtable` pointers, both of which are precisely the targets protected by `icall` and `vcall` CFI. Abadi et al. [1] also show CFI's effectiveness in preventing jump-to-libc and ROP-based heap attacks.

◑ *Medium* – `cast-strict`, `derived-cast`, `unrelated-cast`: Literature, e.g., Abadi et al. [1] and Burow et al. [15], link heap overflows to type confusion and control-flow subversion. These cast-based protections enforce type soundness, helping mitigate, but not fully prevent, such attacks. Sayeed et al. mention IFCC protecting "*forward-edge*" heap resident data, which aligns partially with cast variants.

○ *Low* – the variants `nvcall`, `mfcall` were not directly implicated in any reviewed heap-based buffer overflow scenarios. The literature does not indicate widespread use of heap corruption to subvert static member functions or non-virtual calls.

2) Stack-based Buffer Overflow: Stack overflows have long targeted return addresses and now commonly enable Return-Oriented Programming (ROP). Although CFI aims to protect backward edges, return instructions are typically outside the scope of LLVM's standard CFI variants.

[4]We omit `cfi-` in the following for readability.

TABLE II: Mapping of Memory Corruption Vulnerabilities to LLVM's CFI Mechanisms

CFI Variants	**Memory Corruption Vulnerability**			
	HEAP-BASED BUFFER OVERFLOW	STACK-BASED BUFFER OVERFLOW	USE-AFTER -FREE (UAF)	TYPE CONFUSION
`cfi-cast-strict`	◑	○*	◑	●
`cfi-derived-cast`	◑*	○*	○*	●
`cfi-unrelated-cast`	◑*	○*	○*	●
`cfi-nvcall`	○*	◑	○	○*
`cfi-vcall`	●	○	◑	◑*
`cfi-icall`	●	●	◑	○*
`cfi-mfcall`	○*	○*	○*	○*
Average *mitigation potential*	*moderate*	*low*	*low*	*moderate*

Legend: ○ = *low* , ◑ = *moderate*, ● = *high* mitigation potential
*on entries where the CFI variant was not intended to mitigate the respective vulnerability type.

● *High* – `icall`: Abadi et al. [1], Burow et al. [15], and Padaryan et al. [76] describe how stack smashing can affect function pointers stored in stack memory. Protecting such pointers falls under the domain of `icall` if used in indirect calls.

◑ *Medium* – `cast-strict`: Abadi et al. [1] describe how CFI prevents redirection via corrupted function pointers (even when originating from stack misuse), offering limited coverage depending on pointer usage. `cast-strict` offers partial defense if the vulnerability chain includes illegitimate pointer casting.

○ *Low* – all others: Most variants are not relevant in typical stack-based attacks. Stack-based buffer overflows typically do not involve `vtables`, dynamic casts, or member-function pointers, leading to low mitigation relevance.

3) Use-After-Free (UAF): UAF bugs can reuse dangling pointers to overwrite or fake object layouts—especially `vtable` or function pointers—making CFI for dynamic call targets essential.

◑ *Medium* – `vcall`, `cast-strict`, `derived-cast`: Tice et al. [101] describe how attackers reuse freed objects to inject malicious `vtables`, making `vcall` CFI highly relevant. However, Lee et al. [50] and Sayeed et al. [83] note that cast-based defenses help restrict control flow in UAF exploitation, but attackers may still perform non-control-data attacks. Badoux et al. [7] and Yuan Li et al. [51] describe the use of type-enforcing CFI (like LLVM's cast protections) to mitigate attacks relying on type confusion induced by UAF.

○ *Low* – all others: Despite their use in dispatching calls, `icall`, `nvcall`, and `mfcall`, these variants were not directly associated with UAF attack mechanisms in the literature. While function and member-function pointers might be targets in UAF, no sources documented exploits using them via freed memory.

4) Type Confusion: Type confusion occurs when an object is misinterpreted as another type, often via unsafe casting. This can lead to arbitrary memory access, misused `vtables`, or invalid calls—making type- and callsite-aware CFI essential.

● *High* – `cast-strict`, `unrelated-cast`: Multiple papers (e.g., Farkhani et al. [34], Pang et al. [77], Fan et al. [33]) describe runtime type enforcement (RTC-based CFI) as the most effective strategy to prevent control-flow hijacks via type confusion. These works directly support high mitigation potential for LLVM's `cast-strict` and `unrelated-cast`, both of which enforce type compatibility at runtime. Badoux et al. [7] explain that LLVM-CFI targets polymorphic casts, mapping closely to these variants.

◑ *Medium* – `derived-cast`: Unsafe downcasts are a common source of confusion, for instance, Kim and Kim [28], and `derived-cast` specifically protects against misuse of base-to-subclass conversions. However, it does not address all cases, such as unrelated casts or deep type violations.

○ *Low* – `vcall`, `nvcall`, `icall`, `mfcall`: Although type confusion can lead to hijacked call targets, e.g., forged `vtables`, the function-call variants only mitigate attacks when the call itself is reached. They do not enforce type correctness at the point of cast, making them insufficient against the root cause of confusion-based exploits.

IV. APPLICATION & CVE EXAMPLE EVALUATION

A. CVE Collection Process

We systematically filter and collect Common Vulnerabilities and Exposures (CVEs) to identify relevant and potentially exploitable vulnerabilities.

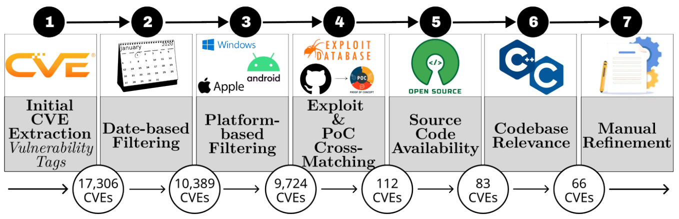


Fig. 2: CVE Collection Process

Figure 2 illustrates these steps. In the first step, **❶-Initial CVE Extraction**, we gather all CVEs from the MITRE [100] database that are tagged with vulnerability types pertinent to our research as discussed in Section III-A and based on the *Top 10 KEV*[5] *Weaknesses* list [56]. Next, we use ❷**-Date-based Filtering** to exclude CVE entries that predate the year 2018. This ensures focus on contemporary security challenges and, more importantly, to excludes CVEs before the introduction of LLVM's CFI in Clang 3.7 [98]. It additionally decreases incompatibilities with older C/C++ standards [53], [102]. Following this, ❸**-Platform-based Filtering** is performed to eliminate CVEs that exclusively affect Windows, Android, MacOS, or iOS. Since we focus on vulnerabilities impacting open-source software ecosystems, MacOS and iOS are irrelevant. These operating systems implement their own CFI mechanisms. Windows [2] and Android [4] fork the implementation of LLVM and add their features, whereas MacOS [5] uses Pointer Authentication Codes, a separate LLVM mitigation technique [99]. These aspects place all of them outside the scope of our study. The fourth step, ❹**-Exploit & PoC Cross-Matching**, involves identifying CVEs with publicly available exploits by cross-referencing ExploitDB [32] and extensive PoC repositories on GitHub [71]. As shown in Figure 2, this step led to the removal of the highest number of CVEs. We then apply ❺**-Source Code Availability** filtering, removing all CVEs not pertaining to open-source software, as access to source code is necessary for further security assessments. Next, ❻**-Codebase Relevance** filtering ensures that only C or C++ code vulnerabilities are retained, as CFI applies to these languages.

Finally, ❼**-Manual Refinement** ensures that CVEs related to Just-in-Time (JIT) compiled code are excluded, as CFI cannot effectively mitigate threats in such dynamically generated environments [8], [69]. This results in excluding almost all type confusion vulnerabilities identified in web browsers such as Firefox and Chrome. Most type confusion vulnerabilities are found in the JavaScript engine of the browser (Firefox's SpiderMonkey [88] and Chrome's V8 [39]) and rely heavily on JITed code, resulting in excluding an additional 28 CVEs. Current implementations of CFI cannot protect it since the JITed code is unavailable during compilation. Moreover, we also omit all vulnerabilities identified within the Linux kernel as LLVM provides a separate implementation of CFI for the kernel (`kcfi`) [44], [97] (removing five CVEs), which is out-of-scope for this work.

This structured multi-step process identifies only the most relevant and applicable CVEs. While this process strongly limits the amount of real-world exploits we can find, it is necessary for targeted research and analysis.

B. Experiment Setup

For each CVE, we set up a podman [80] container with the most compatible OS and version. We compile the vulnerable software without CFI using clang/clang++ to confirm that the PoC or exploit functions as expected. This serves as the baseline for comparison with builds using each of the seven CFI variants. If behavior diverges, we recompile only the vulnerable function with CFI to confirm the violation. We also adjust build configurations as needed for clang and CFI compatibility.

[5]Known Exploited Vulnerabilities

C. Example *of Guideline Application*

The specific security requirements and prioritization strategies inherently depend on the characteristics of the target project and the associated threat model. In this section, we present a representative use case scenario for our taxonomy.

Suppose a developer wants to improve the overall security of an application by integrating CFI mechanisms. Based on Table II, a first step could be to deploy `icall`, as it has a high impact (●) in mitigating heap and stack-based buffer overflows, while only having a low score (○) for type confusion vulnerabilities.

After deployment, two outcomes are possible: (i) the application remains functionally correct and allows the gradual introduction of additional CFI protections, or (ii) runtime errors occur that require debugging or re-evaluating the approach. In both cases, the developer must make informed decisions about the subsequent integration steps. If the application is prone to type confusion, it is recommended to prioritize the implementation of CFI cast variants.

The proposed methodology is in line with practices used by mature projects such as Chromium, which emphasizes the protection of indirect and virtual function calls as a key strategy to increase security [96].

D. Evaluation of CVEs with CFI

TABLE III: Example CVEs

CVE ID	Affected Project	Affected Version	PoC/Exploit	Severity
CVE-2021-3156 [67]	sudo [89]	`<1.9.5p2`	Heap Overflow* Privilege escalation [24]	7.8 (High)
CVE-2023-49992 [65]	espeak-NG [29]	`1.52-dev`	Stack Overflow* corrupting the stack [85]	5.3 (Medium) 7.8 (High) [35]
CVE-2022-3666 [64]	Bento4 [6]	`1.6.0-639`	UAF double-free [16]	7.8 (High)
CVE-2024-34391 [66]	libxmljs [52]	`<=1.0.11`	Type Confusion type misinterpretation [103]	N/A 9.2 (Critical) [36]

*refers to heap- and stack-based buffer overflow respectively.

We identified four relevant CVEs matching all necessary requirements described in Section IV-A. An overview of these CVEs is displayed in Table III. None of the projects containing the vulnerabilities (CVEs) support CFI by default. Table IV depicts the outcomes of our experiments based on the CVE collection process.

In a nutshell, the heap- and stack-based buffer overflow exploits were successfully mitigated by `icall`. While `unrelated-cast` prevented the UAF-PoC, the CFI violation occurred due to the library using an old C/C++ standard and not exploiting the vulnerability in itself. CFI was not able to prevent the chosen type confusion exploit. The following explains the technical details of the chosen CVEs, the exploit process, and CFI's impact. We provide more in-depth technical details for each of the exploits as artifacts[3].

TABLE IV: Results of CVEs compiled with LLVM's CFI Mechanisms

CFI Variants	**Memory Corruption Vulnerability**			
	HEAP-BASED BUFFER OVERFLOW *CVE-2021-3156* sudo [89]	STACK-BASED BUFFER OVERFLOW *CVE-2023-49992* espreak-NG [29]	USE-AFTER -FREE (UAF) *CVE-2022-3666* Bento4 [6]	TYPE CONFUSION *CVE-2024-34391* libxmljs [52]
`cfi-cast-strict`	↻	↻	↻	↻
`cfi-derived-cast`	↻	↻	↻	↻
`cfi-unrelated-cast`	↻	↻	↻*	↻
`cfi-nvcall`	↻	↻	↻	↻
`cfi-vcall`	↻	↻	↻	↻
`cfi-icall`	✓	✓	↻	↻
`cfi-mfcall`	↻	↻	↻	↻

↻ = compiling **but no** prevention of the specific CVE based PoC/exploit, ✓ = compiling **and** prevention
**compiling but CFI violation in code not connected to CVE (mismatch of old C/C++ standard [53], [102])*

1) Heap-based Buffer Overflow - CVE-2021-3156 *[67]:* or "*Baron Samedit*" is a heap overflow in sudo [89] allowing privilege escalation via malformed arguments ending with backslashes. The vulnerability bypasses sudo's argument validation when using `sudoedit -s` instead of `sudo -e` [24]. The bug occurs in `set_cmnd()` (technical details available as artifacts[3]) where backslash handling increments pointers past buffer boundaries.

CFI Impact: `register_hooks()` causes a CFI violation of **`icall` preventing** the exploit in `load_plugins.c`. `register_hooks()` is responsible for registering certain hooks involving function pointers needed to operate sudo. The heap buffer overflow caused by malformed input can overwrite these function pointers, effectively altering the execution flow. When enabled, CFI ensures that indirect calls only jump to valid addresses explicitly registered or verified during the program's initialization. Since an attacker has altered the function pointer, CFI detects that the call is attempting to jump to an address that is not allowed.

2) Stack-based Buffer Overflow - CVE-2023-49992 [65]: describes a stack overflow in eSpeak-NG [29]'s `RemoveEnding` function (technical details available as artifacts[3]). A crafted input exploiting multibyte UTF-8 processing causes a buffer overflow in `ending[50]`, allowing out-of-bounds writes that corrupt stack variables and potentially hijack control flow, as demonstrated in the PoC for CVE-2023-49992.

CFI Impact: **`icall` prevents** this attack by enforcing strict runtime checks that verify whether an indirect function call actually points to a valid function of the expected type. A `Runtime Error in src/libespeak-ng/speech.c:473` in the function `synth_callback` triggered by a CFI violation occurs. When the corrupted `synth_callback` function pointer is used at `speech.c`, the runtime detects that the pointer does not match a valid function of the expected type `int (short *, int, espeak_EVENT *)`. Instead of allowing execution to proceed to an unintended location, `icall` immediately traps and aborts execution, preventing any malicious code from running. This stops the exploit before the attacker can gain control over the process. In summary, `icall` (i) shows a potential exploitation vector for the PoC and (ii) prevents it from being viable.

As predicted by our mapping in Table II, the other CFI variants for function call integrity are less effective. `nvcall` protects non-virtual function calls, which are direct and unaffected by function pointer overwrites. `vcall` enforces integrity for virtual function calls in C++ objects, ensuring that calls to virtual functions only resolve to valid virtual table entries, which is irrelevant here since the exploit does not involve C++ virtual functions. `mfcall` protects member function calls on objects, ensuring that function pointers within objects are valid before invocation, but the function pointer, in this case, is not a member function but rather a standalone function pointer used in the callback mechanism.

3) Use-After-Free - CVE-2022-3666 [64]: describes a UAF vulnerability in Bento4's [6] `AP4_LinearReader` class (still unpatched). When handling failed sample reads, the issue occurs in `Advance()` (technical details available as artifacts[3]).

CFI Impact: In Bento4, the vulnerability occurs when an object is deleted, but a dangling pointer to it remains accessible. If the same memory is later reallocated for a different object, an attacker can manipulate this stale pointer to interact with unintended memory contents. CFI primarily ensures that function calls and type casts follow the expected class hierarchy and valid control flow. Still, it does not prevent access to a freed object if the memory has been repurposed. In the available PoC, the stale pointer is not used to perform an exploit. Thus, the `vtable` is not corrupted, nor is an invalid cast performed. Moreover, the PoC leads to Denial-of-Service, which CFI does not cover. These are the reasons why **none** of the CFI variants detects a violation.

A CFI `unrelated-cast` violation occurs in `Ap4DArray.h`, which is not connected to CVE-2022-3666 but rather an incompatibility of CFI with older C/C++ standards [53], [102].

4) Type Confusion - CVE-2024-34391 [66]: describes a type confusion vulnerability in libxmljs [52], the Node.js [73] bindings for `libxml2` [37]. The issue occurs during XML entity reference processing when SWIG-generated bindings incorrectly interpret an `xmlEntity` as an `xmlNode` (technical details available as artifacts[3]). <u>CFI Impact:</u> While all seven CFI variants compiled successfully, **none** of them was able to prevent the DoS exploit. The most likely variant to mitigate this kind of vulnerability would be one of the cast-based ones (see Table II). These techniques require the presence of an actual cast, meaning a `dynamic_cast` or `static_cast` in C++. However, type confusion is not caused by an incorrect cast operation but rather by a misinterpretation of the type itself, more precisely, a misassumption of the memory layout of the accessed object. The problem arises when C function pointers are cast into C++ objects, circumventing proper type checks. Since the C++ compiler does not have full visibility on the memory layout and function pointer integrity of the underlying C code, it cannot enforce the expected control flow restrictions. This allows an attacker to exploit the type confusion and execute unintended behavior, bypassing CFI.

V. Discussion

A. Effectiveness & Limitations of CFI Preventing Exploitation

In the cases where CFI prevents the exploitation of a specific vulnerability (Sections IV-D1 and IV-D2), it may be possible to create a different exploit to circumvent CFI. However, we are not trying to show that CFI prevents exploitation altogether. Instead, our results show that in-the-wild exploits lose viability if CFI is used during compilation. In conjunction with other mitigation techniques, engineering an exploit to bypass all of them becomes increasingly complex. We also point out that in the cases of IV-D3 and IV-D4, even though CFI cannot stop these specific vulnerabilities, it does not mean CFI is unable to prevent exploitation of the underlying vulnerability type in general. Rather, our provided mapping (Table II) depicts a general guideline.

B. Practical Guidance for Incremental CFI Deployment

In practice, this means developers should not rely on a specific CFI variant to guarantee protection against exploitation of a vulnerability type. However, our mapping allows strategic planning of gradual CFI adoption to existing code bases. For example, one could start to integrate `vcall` and `icall` (as done by Chromium [96]), which already mitigates *heap-based* and *stack-based buffer overflow* exploits to a substantial degree while providing some protection against UAF.

An alternate strategy is to focus on the CFI variants, which affect the usability of the target software the least. Let us assume compiling a piece of software with all CFI variants for type casting protection (Section II-A) works out of the box, while the other variants cause errors breaking the software's functionality due to compatibility issues, which is a common challenge of CFI [14], [41], [53], [102]. Our mapping can help to find the best CFI option still offering the next most protection. Generally speaking, the more CFI variants are added, the better the protection. However, due to its highly complex implementation and compatibility challenges, it is recommended to start with the most useful ones, depending on the target project's vulnerabilities and needs.

C. Challenges in Real-World CFI Application

When applying CFI to real-world scenarios, unforeseen challenges may arise, as exemplified by the type confusion vulnerability discussed earlier in Section IV-D4. In this case, the issue originated from the interaction and, more specifically, the translation between multiple programming languages (C, C++, and JavaScript). Such challenges are less likely to manifest when working solely with synthetic or carefully constructed case-based examples, as these are designed to isolate specific factors rather than account for the complexities of real-world implementation. Existing literature has primarily focused on such controlled examples (see Section VII), leaving gaps in understanding how these issues manifest in practical, real-world settings.

D. Comparison with other CFI Implementations

Windows Control Flow Guard (CFG) [2] provides a coarse-grained implementation of forward-edge CFI. It restricts indirect calls to a set of valid function entry points, but does not enforce additional rules for signature matching or class hierarchies. GCC offers only limited CFI support compared to LLVM, as it focuses primarily on virtual table verification (VTV) [91]–[93], [101], [106]. This mechanism helps detect specific type confusion vulnerabilities in C++. However, it does not cover other indirect control transfers, nor does it provide granularity of CFI variants. In contrast, LLVM implements a comprehensive CFI implementation that supports multiple variants, including checks on indirect calls, virtual functions, and various forms of type casting.

Intel's Control Flow Enforcement Technology (CET) [43] extends software CFI with hardware-based shadow stacks and indirect branch tracking, thereby strengthening protection against backward branches and improving security with minimal performance overhead. However, CET focuses primarily on backward-edge control flow integrity and does not replace the checks provided by software solutions such as LLVM's CFI. Given LLVM's flexible and detailed software-based options, it provides an ideal implementation for analyzing practical trade-offs in CFI deployment, which motivated our work on LLVM's CFI.

VI. Threats to Validity

A. Vulnerability Selection

We selected one representative CVE per vulnerability category from the KEV Top 10 list to ensure relevance and real-world impact. However, this small sample size may not capture the full extent of each category, as it is limited by the restricted availability of working PoCs and exploits. Thus, our conclusions may not be generalizable to all potential exploits in each class. Additionally, there may be other relevant memory corruption vulnerabilities (e.g., Time-of-check Time-of-use (TOCTOU) Race Condition) related to CFI that are less likely to occur in the real world and are therefore not included in the KEV list (and thus not in this work).

B. Mapping from CFI Variants to Vulnerability Types

Our taxonomy is based on the publicly available LLVM documentation, publications, and empirical testing. However, the literature is occasionally ambiguous or outdated, and not all implementation details are disclosed. Therefore, some mappings between CFI variants and vulnerability types required informed interpretation.

C. Limitations of Experiment Setup

Our evaluation used CVE PoCs and exploits executed on modified builds using CFI instrumentation. Although the only changes to the build options were related, we cannot guarantee that they did not have any other effects or that other potential modifications (e.g., using alternative optimization levels) could have an impact. Some exploits required minor adjustments to function in our testbed. Additionally, we did not include performance or usability evaluations, which are critical for real-world deployment but were outside the scope of this work.

D. Human Intervention by the Researchers

Significant manual effort was required to reproduce CVEs and determine whether CFI blocked exploitation. These judgments, although validated through debugging and reverse engineering, may still introduce human bias or error. In addition, we had to determine whether the identified exploits/PoCs fit our purpose, which could introduce an additional layer of bias, even though we made every effort to find representative and available options. Moreover, the mapping process, as described in Section VI-B, required an informed interpretation of the source at specific points. We attempted to minimize bias by providing the codebook as part of the artifacts[3].

VII. Related Work

A. CFI Deployment Challenges

Several studies have identified key challenges in applying CFI in practice. Becker et al. [9] highlight incomplete instrumentation, unprotected dynamic libraries, and partial enforcement that weaken security guarantees. Although this work is a SoK paper, it focuses solely on the actual deployment of CFI on Android and Windows rather than providing a guide on how and when to apply CFI. Houy and Bartel [41], [42] emphasize performance overhead, compatibility with legacy code, and difficulties handling dynamic features like just-in-time compilation in complex runtimes. Xu et al. [109] point out frequent compatibility issues, false positives, and limited support for complex software patterns, revealing trade-offs between security precision and deployability. These challenges collectively hinder broad and effective CFI adoption.

B. CFI Working Examples

Nearly all of the papers rely on synthetic PoCs rather than real-world documented attacks. Evans et al. [30], Niu et al. [68], [70], Sayeed et al. [83], and Xu et al. [109] evaluate LLVM's CFI using controlled experiments and synthetic exploit scenarios to demonstrate its effectiveness. Burow et al. [15] focus mainly on empirical testing with real-world applications and benchmark suites designed to simulate practical workloads and measure performance and security outcomes. In contrast, our work aims to provide a practical deployment guideline for developers, supporting informed and incremental adoption of CFI in existing code bases. Similarly, Conti et al. [22], and Li et al. [51] analyze CFI's impact on various attack vectors through constructed test cases. While Farkhani et al. [34] discuss attack techniques observed in practical settings, the evaluation still primarily relies on PoCs rather than exploits seen in the wild. These PoCs are carefully constructed experiments that illustrate how specific bypass techniques can be employed against LLVM's CFI, thereby exposing potential weaknesses in its implementation. Although the scenarios mimic real-world conditions to some extent, they remain controlled demonstrations rather than real-world attacks.

C. CFI Bypasses and Limitations

Many attack vectors bypass coarse-grained CFI [19], [26], [38]. Carlini et al. showed that dispatcher functions (e.g., `memcpy`) can enable a *return-to-libc* attack. However, CFI blocked arbitrary code execution in some cases [18], with further reductions when combined with a shadow stack. ROP and JOP attacks use small code fragments for arbitrary execution. Studies [15], [30], [68], [70] indicate that LLVM's CFI limits naive ROP/JOP attacks by enforcing a fixed CFG, yet advanced memory disclosure bypasses remain. COOP attacks, which manipulate virtual function calls, also partially evade CFI protection [22], [83], [84]. Although enhanced per-input CFI improves forward-edge protection, LLVM's CFI still struggles against speculative execution-based attacks [109]. Similarly, shadow stacks strengthen defenses against stack-based and heap vulnerabilities [34], [51] but do not entirely prevent return address manipulation.

VIII. Conclusion

This work presents a structured guideline that maps the most critical memory corruption vulnerabilities to LLVM's forward-edge CFI variants, offering developers a practical reference for informed CFI deployment. While CFI is a powerful mitigation, its adoption remains limited due to integration challenges, such as runtime violations caused by benign code patterns or mismatches with existing design decisions. Our guideline addresses these obstacles by clarifying which CFI variants are most suitable for which vulnerability types. To illustrate its relevance, we analyze four recent, high-impact vulnerabilities. In two cases, the mapped CFI variant would have blocked the exploitation, underscoring the guideline's practical utility.

We advocate for broader CFI adoption in memory-unsafe projects and call for future work to focus on easing its integration. Our guideline is a step toward making CFI more accessible and deployable in real-world systems.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. In addition, this work was supported in part by the Semiconductor Research Corporation (SRC) and DARPA.

References

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

[2] alvinashcraft, alexbuckgit, mattwojo, v-kents, DCtheGeek, drewbatgit, knicholasa, ErikSwan, mijacobs, msatranjr. Control Flow Guard for platform security - Win32 apps | Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, Dec 2024. Accessed 2025-02-18.

[3] Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu. On bridging the gap between control flow integrity and attestation schemes. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6633–6650, 2024.

[4] AOSP | Android Open-Source Project. Control Flow Integrity | Android Open Source Project. https://source.android.com/docs/security/test/cfi, Aug 2024. Accessed 2024-11-01.

[5] Apple Developer. Improving control flow integrity with pointer authentication | Apple Developer. https://developer.apple.com/documentation/browserenginekit/improving-control-flow-integrity-with-pointer-authentication, Dec 2024. Accessed 2025-02-18.

[6] axiomatic-systems. Bento4. https://www.bento4.com/, Dec 2024. Accessed 2025-03-08.

[7] Nicolas Badoux, Flavio Toffalini, Yuseok Jeon, and Mathias Payer. Type++: prohibiting type confusion with inline type information. NDSS, 2025.

[8] Erick Bauman, Jun Duan, Kevin W Hamlen, and Zhiqiang Lin. Renewable just-in-time control-flow integrity. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 580–594, 2023.

[9] Lucas Becker, Matthias Hollick, and Jiska Classen. {SoK}: On the effectiveness of {Control-Flow} integrity in practice. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 189–209, 2024.

[10] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[11] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. Smashing the stack protector for fun and profit. In *ICT Systems Security and Privacy Protection: 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings 33*, pages 293–306. Springer, 2018.

[12] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS*, 2016.

[13] Cyril Bresch, David Hély, Roman Lysecky, Stéphanie Chollet, and Ioannis Parissis. Trustflow-x: A practical framework for fine-grained control-flow integrity in critical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(5):1–26, 2020.

[14] Bugzilla. 510629 - (cfi)[meta] Ship Control Flow Integrity. https://bugzilla.mozilla.org/show_bug.cgi?id=510629, Aug 2009. Accessed 2024-11-01.

[15] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[16] burymyname. heap-use-after-free bug in mp42ts - Issue #793 · axiomatic-systems/Bento4. https://github.com/axiomatic-systems/Bento4/issues/793, Oct 2022. Accessed 2025-03-04.

[17] Muhammad Arif Butt, Zarafshan Ajmal, Zafar Iqbal Khan, Muhammad Idrees, and Yasir Javed. An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences*, 12(13):6702, 2022.

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

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

[20] Zeyu Chen, Daiping Liu, Jidong Xiao, and Haining Wang. All use-after-free vulnerabilities are not created equal: An empirical study on their characteristics and detectability. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 623–638, 2023.

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

[22] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963, 2015.

[23] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[24] CptGibbon. CptGibbon/CVE-2021-3156: Root shell PoC for CVE-2021-3156. https://github.com/CptGibbon/CVE-2021-3156, 2022. Accessed 2025-02-28.

[25] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[26] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of {Coarse-Grained}{Control-Flow} integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, 2014.

[27] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51, 2011.

[28] Dongjoo Kim and Seungjoo Kim. Bintyper: Type confusion detection for c++ binaries. https://i.blackhat.com/eu-20/Thursday/eu-20-Kim-BinTyper-Type-Confusion-Detection-For-C-Binaries-wp.pdf, Dec 2020. Accessed 2024-11-17.

[29] espeak-ng. espeak-ng/espeak-ng: eSpeak NG is an open source speech synthesizer that supports more than hundred languages and accents. https://github.com/espeak-ng/espeak-ng, Mar 2025. Accessed 2025-03-08.

[30] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[31] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[32] Exploit-DB. Exploit Database - Exploits for Penetration Testers, Researchers, and Ethical Hackers. https://www.exploit-db.com/, Feb 2025. Accessed 2025-02-18.

[33] Xiaokang Fan, Sifan Long, Chun Huang, Canqun Yang, and Fa Li. Accelerating type confusion detection by identifying harmless type castings. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 91–100, 2023.

[34] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 28–39, 2018.

[35] GitHub Advisory. Espeak-ng 1.52-dev was discovered to contain a Stack... · CVE-2023-49992 · GitHub Advisory Database. https://github.com/advisories/GHSA-2g42-jgwr-h29g, Jan 2024. Accessed 2025-03-08.

[36] GitHub Advisory. libxmljs vulnerable to type confusion when parsing specially crafted XML · CVE-2024-34391 · GitHub Advisory Database. https://github.com/advisories/GHSA-6433-x5p4-8jc7, Nov 2024. Accessed 2025-03-08.

[37] GNOME. GNOME / libxml2 · GitLab. https://gitlab.gnome.org/GNOME/libxml2, Mar 2025. Accessed 2025-03-08.

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

[39] Google V8. V8 JavaScript engine. https://v8.dev/, 2025. Accessed 2025-03-08.

[40] Binfa Gui, Wei Song, Hailong Xiong, and Jeff Huang. Automated use-after-free detection and exploit mitigation: How far have we gone? *IEEE Transactions on Software Engineering*, 48(11):4569–4589, 2021.

[41] Sabine Houy and Alexandre Bartel. Lessons learned and challenges of deploying control flow integrity in complex software: the case of openjdk's java virtual machine. In *2024 IEEE Secure Development Conference (SecDev)*, pages 153–165. IEEE, 2024.

[42] Sabine Houy and Alexandre Bartel. Twenty years later: Evaluating the adoption of control flow integrity. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–30, 2025.

[43] intel. A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html, 2020. Accessed 2025-07-15.

[44] Jake Edge. Control-low integrity for the kernel [LWN.net]. https://lwn.net/Articles/810077/, January 2020. Accessed 2025-02-28.

[45] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. Return-oriented programming on risc-v. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 471–480, 2020.

[46] Hyerean Jang, Moon Chan Park, and Dong Hoon Lee. Ibv-cfi: Efficient fine-grained control-flow integrity preserving cfg precision. *Computers & Security*, 94:101828, 2020.

[47] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387, 2017.

[48] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards efficient heap overflow discovery. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 989–1006, 2017.

[49] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 195–211, 2019.

[50] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[51] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, 2020.

[52] libxmljs. libxmljs. https://github.com/libxmljs, Dec 2024. Accessed 2025-03-08.

[53] llvmbot. CFI (Control Flow Integrity) emits invalid checks when using std::make_shared (C++11) · Issue #34779 · llvm/llvm-project. https://github.com/llvm/llvm-project/issues/34779, Nov 2017. Accessed 2025-03-06.

[54] Faming Lu, Mengfan Tang, Yunxia Bao, and Xiaoyu Wang. A survey of detection methods for software use-after-free vulnerability. In *International Conference of Pioneering Computer Scientists, Engineers and Educators*, pages 272–297. Springer, 2022.

[55] Nicoló Maunero, Paolo Prinetto, and Gianluca Roascio. Cfi: Control flow integrity or control flow interruption? In *2019 IEEE East-West Design & Test Symposium (EWDTS)*, pages 1–6. IEEE, 2019.

[56] MITRE. CW -2023 CWE Top 10 KEV Weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html, Dec 2023. Accessed 2024-11-01.

[57] MITRE. CWE -2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html, Nov 2024. Accessed 2025-05-17.

[58] MITRE - Common Weakness Enumeration. CWE - CWE-121: Stack-based Buffer Overflow (4.15). https://cwe.mitre.org/data/definitions/121.html. Accessed 2024-11-18.

[59] MITRE - Common Weakness Enumeration. CWE - CWE-122: Heap-based Buffer Overflow (4.15). https://cwe.mitre.org/data/definitions/122.html. Accessed 2024-11-18.

[60] MITRE - Common Weakness Enumeration. CWE - CWE-416: Use After Free (4.15). https://cwe.mitre.org/data/definitions/416.html. Accessed 2024-11-18.

[61] MITRE - Common Weakness Enumeration. CWE - CWE-843: Access of Resource Using Incompatible Type ('Type Confusion'). https://cwe.mitre.org/data/definitions/843.html. Accessed 2024-11-18.

[62] Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. Castsan: Efficient detection of polymorphic c++ object type confusions with llvm. In *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I 23*, pages 3–25. Springer, 2018.

[63] Ștefan Nicula and Răzvan Daniel Zota. Exploiting stack-based buffer overflow using modern day techniques. *Procedia Computer Science*, 160:9–14, 2019.

[64] NIST | National Vulnerability Database. NVD - CVE-2022-3666. https://nvd.nist.gov/vuln/detail/CVE-2022-3666, Nov 2024. Accessed 2025-03-04.

[65] NIST | National Vulnerability Database. NVD - cve-2023-49992. https://nvd.nist.gov/vuln/detail/CVE-2023-49992, Nov 2024. Accessed 2025-02-28.

[66] NIST | National Vulnerability Database. NVD - CVE-2024-34391. https://nvd.nist.gov/vuln/detail/CVE-2024-34391, Nov 2024. Accessed 2025-02-28.

[67] NIST | National Vulnerability Database. NVD - cve-2021-3156. https://nvd.nist.gov/vuln/detail/CVE-2021-3156, Feb 2025. Accessed 2025-02-28.

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

[69] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328, 2014.

[70] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 914–926, 2015.

[71] nomi-sec. nomi-sec/PoC-in-GitHub: PoC auto collect from GitHub. Be careful Malware. https://github.com/nomi-sec/PoC-in-GitHub, Feb 2025. Accessed 2025-02-18.

[72] Aleph One. Smashing the stack for fun and profit. https://phrack.org/issues/49/14, 1996. Accessed: 2025-02-17.

[73] OpenJS Foundation. Node.js — Run JavaScript Everywhere. https://nodejs.org/en, 2025. Accessed 2025-03-08.

[74] OWASP®Foundation. Buffer Overflow | OWASP Foundation. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow. Accessed 2024-11-17.

[75] OWASP®Foundation. Using freed memory | OWASP Foundation. https://owasp.org/www-community/vulnerabilities/Using_freed_memory. Accessed 2024-11-17.

[76] Vartan A Padaryan, VV Kaushan, and AN Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. *Programming and Computer Software*, 41:373–380, 2015.

[77] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. Mapping to bits: Efficiently detecting type confusion errors. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 518–528, 2018.

[78] Emanuele Parisi, Alberto Musa, Simone Manoni, Maicol Ciani, Davide Rossi, Francesco Barchi, Andrea Bartolini, and Andrea Acquaviva. Titancfi: Toward enforcing control-flow integrity in the root-of-trust. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[79] Moon Chan Park and Dong Hoon Lee. Bgcfi: Efficient verification in fine-grained control-flow integrity based on bipartite graph. *IEEE Access*, 11:4291–4305, 2023.

[80] podman. Podman. https://podman.io/, 2025. Accessed 2025-03-08.

[81] Gerardo Richarte et al. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1(7), 2002.

[82] Isaac Richter, Jie Zhou, and John Criswell. Detrap: Risc-v return address protection with debug triggers. In *2024 IEEE Secure Development Conference (SecDev)*, pages 166–177. IEEE, 2024.

[83] Sarwar Sayeed, Hector Marco-Gisbert, Ismael Ripoll, and Miriam Birch. Control-flow integrity: Attacks and protections. *Applied Sciences*, 9(20):4229, 2019.

[84] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[85] SEU-SSL. Poc/espeak-ng at main · SEU-SSL/Poc. https://github.com/SEU-SSL/Poc/tree/main/espeak-ng, 2023. Accessed 2025-02-28.

[86] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. CCS '04, page 298–307, New York, NY, USA, 2004. Association for Computing Machinery.

[87] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE symposium on security and privacy*, pages 574–588. IEEE, 2013.

[88] SpiderMonkey Developers. Home | SpiderMonkey JavaScript/WebAssembly Engine. https://spidermonkey.dev/, 2025. Accessed 2025-03-08.

[89] Sudo Project. Sudo. https://www.sudo.ws/, Mar 2025. Accessed 2025-03-08.

[90] Gang Tan and Trent Jaeger. Cfg construction soundness in control-flow integrity. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 3–13, 2017.

[91] GNU Compiler Collection team. Instrumentation Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html, 2025. Accessed 2025-07-15.

[92] GNU Compiler Collection team. Vtable Verification Feature Proposal. https://docs.google.com/document/d/1EsIEkOUHMrTmgRnFNEe9ZupXPYq1wPliuFQB3GPJtzs/pub, 2025. Accessed 2025-07-15.

[93] GNU Compiler Collection team. Vtable Verification User's Guide. https://docs.google.com/document/d/1wN-uygC0hicLe1dyAGCvtn_tJhnwFer0Nsy56b84doY/pub, 2025. Accessed 2025-07-15.

[94] The Clang Team. Control Flow Integrity – Clang 15.0.0.git documentation. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2024. Accessed 2024-11-19.

[95] The Clang Team. Control Flow Integrity design documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html, 2025. Accessed 2025-03-08.

[96] The Chromium Projects. Control Flow Integrity. https://www.chromium.org/developers/testing/control-flow-integrity/, Nov 2023. Accessed 2024-11-01.

[97] The Clang Team. Control Flow Integrity – Clang 15.0.0.git documentation. https://clang.llvm.org/docs/ControlFlowIntegrity.html#fsanitize-kcfi. Accessed 2025-02-28.

[98] The Clang Team. bcain-llvm-readthedocs-io-clang-en-release_37.pdf. https://bcain-llvm.readthedocs.io/_/downloads/clang/en/release_37/pdf/, Aug 2017. Accessed 2025-03-06.

[99] The Clang Team. Pointer Authentication — Clang 21.0.0git documentation. https://clang.llvm.org/docs/PointerAuthentication.html, 2025. Accessed 2025-03-08.

[100] The MITRE Corporation. CVE - CVE. https://cve.mitre.org/, Oct 2024. Accessed 2025-03-08.

[101] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}. In *23rd USENIX security symposium (USENIX security 14)*, pages 941–955, 2014.

[102] Tom_Roeder. [RFC] Simple control-flow integrity - Project Infrastructure / LLVM Dev List Archives - LLVM Discussion Forums. https://discourse.llvm.org/t/rfc-simple-control-flow-integrity/31010, Feb 2014. Accessed 2025-03-06.

[103] uriyay-jfrog. Type confusion in _wrap__xmlNode_properties_get leads to an RCE · Issue #645 · libxmljs/libxmljs. https://github.com/libxmljs/libxmljs/issues/645, Nov 2023. Accessed 2025-02-28.

[104] Perry Wagle, Crispin Cowan, et al. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, volume 1. Citeseer, 2003.

[105] Wang Wei. Survey of attacks and defenses on stack-based buffer overflow vulnerability. In *7th International Conference on Education, Management, Information and Computer Science (ICEMC 2017)*, pages 324–328. Atlantis Press, 2016.

[106] GCC Wiki. vtv - GCC Wiki. https://gcc.gnu.org/wiki/vtv, 2025. Accessed 2025-07-15.

[107] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. Warpattack: Bypassing cfi through compiler-introduced double-fetches. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1271–1288, 2023.

[108] Shenglin Xu and Yongjun Wang. Bofaeg: Automated stack buffer overflow vulnerability detection and exploit generation based on symbolic execution and dynamic analysis. *Security and Communication Networks*, 2022(1):1251987, 2022.

[109] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. Confirm: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1805–1821, 2019.

[110] Qiang Zeng, Mingyi Zhao, and Peng Liu. Heaptherapy: An efficient end-to-end solution against heap buffer overflows. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 485–496, 2015.

[111] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE symposium on security and privacy*, pages 559–573. IEEE, 2013.

[112] Mingwei Zhang and R Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 91–100, 2015.

[113] Changwei Zou, Dongjie He, Yulei Sui, and Jingling Xue. Tips: Tracking integer-pointer value flows for c++ member function pointers. *Proceedings of the ACM on Software Engineering*, 1(FSE):1609–1631, 2024.

[114] Changwei Zou, Yulei Sui, Hua Yan, and Jingling Xue. Tcd: Statically detecting type confusion errors in c++ programs. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 292–302. IEEE, 2019.