applied sciences

MDPI

Article

Control-Flow Integrity: Attacks and Protections

Sarwar Sayeed [1], Hector Marco-Gisbert [1,*], Ismael Ripoll [2] and Miriam Birch [1]

[1] School of Computing, Engineering and Physical Sciences, University of the West of Scotland, High Street, Paisley PA1 2BE, UK; Sarwar.Sayeed@uws.ac.uk (S.S.); miriam.Birch@uws.ac.uk (M.B.)

[2] Department of Computing Engineering, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain; iripoll@disca.upv.es

* Correspondence: hector.marco@uws.ac.uk; Tel.: +44-141-849-4418

Received: 23 August 2019; Accepted: 3 October 2019; Published: 10 October 2019

Abstract: Despite the intense efforts to prevent programmers from writing code with memory errors, memory corruption vulnerabilities are still a major security threat. Consequently, control-flow integrity has received significant attention in the research community, and software developers to combat control code execution attacks in the presence of type of faults. Control-flow Integrity (CFI) is a large family of techniques that aims to eradicate memory error exploitation by ensuring that the instruction pointer (IP) of a running process cannot be controlled by a malicious attacker. In this paper, we assess the effectiveness of 14 CFI techniques against the most popular exploitation techniques, including code reuse attacks, return-to-user, return-to-libc, and replay attacks. We also classify these techniques based on their security, robustness, and implementation complexity. Our study indicates that the majority of the CFI techniques are primarily focused on restricting indirect branch instructions and cannot prevent all forms of vulnerability exploitation. We conclude that the performance overhead introduced, jointly with the partial attack coverage, is discouraging the industry from adopting most of them.

Keywords: CFI protections; CFI attacks; memory errors; security; exploitation

1. Introduction

Cyber Security is a fast-evolving environment. One of the most active research areas is the one that tries to address the Control Flow Integrity problem. Memory corruption errors were one of the first family of programming faults exploited by early attackers, which still have not been properly addressed. In general "Memory error" is a wide family of programming errors that can be abused by the attacker to alter the low-level behavior (sequence of processor instructions being executed) of a running process by changing the content of some memory locations. In the CFI literature, the causes of a memory error is a software fault, i.e., it is not considered the physical integrity of the memory device, but the logical content on the memory. Although most CFI solutions can also address physical faults such as RowHammer [1,2], we will not consider those class of physical attacks that modify the content of the memory by physical means.

In most cases, the manifestation of many memory errors is the abnormal termination of the process due to illegal access to memory or instruction; but advanced hackers can take the control flow and put the vulnerable process under their command. The final result is a RCE (Remote Code Execution), which gives the control of the system to the attacker.

Due to continuous improvements in the attacking methods and techniques, a protection technique often becomes outdated and having been required to come up with something more advanced. Over the years the set of exploiting techniques and the defensive countermeasures required have increased significantly. The early attacker just uploaded (injected) in the target process the code that they wanted to execute and redirected the control flow to that code. Presently, attackers can use a wide arsenal of techniques, from the more generic ones as ROP [3,4] (Return Oriented Programming) or JOP [5] (Jump Oriented Programming); through the ones that exploit design choices as the Sigreturn [6], which depends on the way Unix handles signals; to the implementation-specific ones, as for example offset2lib [7] or ret2csu [8] which exploits weaknesses on the implementation of the `mmap()` syscall on Linux and the run-time support appended to ELF executables respectively. Although at first glance, most of these techniques may seem to be hard to understand and even harder to use in real scenarios, there good tutorials, videos, and even tools (for example, ropshell) to help security researchers, as well as attackers, to effectively use them.

The term CFI is used to refer to any solution that can deter or mitigate the severity of the attacks that exploit memory errors by trying to prevent the effective control of flow (program counter). To accomplish an attack, an adversary goes through various attacking stages where obtaining control over the IP is the very first step of the vulnerability exploitation. Protection techniques such as Stack Smashing Protector (SSP) [9–11], Address Space Layout Randomization (ASLR) [12–14] and Non-executable (NX) [15] bit are present in all modern systems, but unfortunately, recent attacking techniques have improved as to bypass all those protection mechanisms [16]. In this work, we consider only control flow integrity techniques, i.e., methods that try to preserve the flow of execution initially programmed by the developer. These techniques do not remove the vulnerabilities but make it harder or even impossible to take advantage of the faults to create exploits. For example, most CFI techniques would force a crash when the attacker tries to abuse the vulnerability, this way, a remote code execution would be converted into a denial of service, which is much less dangerous.

Obviously, the desirable way to avoid memory error attacks is by removing at once the root cause of the vulnerabilities by writing code free of faults. Unfortunately, the complexity of the software makes it virtually an impossible goal. In this work, we will not consider the software engineering techniques, as robust languages and syntactical constructions. Also is out of the scope of this paper code integrity [17,18] techniques (we will consider that the attacker cannot alter the original code of the application).

The remainder of the paper is structured as follows. First, we describe the type of programming errors that may result from the generation of vulnerable code, followed by the way the attackers can exploit these errors. The top-level classification of the CFI techniques is presented in Section 4. Sections 5 and 6 describes and analyzes the various CFI techniques respectively. The work is completed with the discussion of the results, followed by the conclusions.

2. Memory Errors

According to the CWE [19] taxonomy, there is no such thing as a memory error weakness fault, but a large set of weaknesses that may cause a memory error.

One of the main causes of memory overrun errors is unsafe languages, such as C and C++. The errors already exist in vast amounts of software that are currently being used widely. Unsafe languages are frequently used to write operating system kernels and libraries, whereas the safe languages, for instance, Java, highly relies on an interpreter that is developed on unsafe languages [20], various C/C++ functions do not check for buffer lengths, boundaries and may result in buffer overflow, string manipulation or other memory errors. In this section, we discuss and present various memory error vulnerabilities which can lead to control flow hijacking. We will not consider faults that always cause a process crash and where the attacker has no possibility to interact or manipulate the cause of the error.

2.1. Stack-Based Buffer Overflow

A stack-based buffer overflow fault is a type of out-of-bounds write where the buffer being overwritten is allocated on the stack (i.e., a local variable or, rarely, a parameter to a function) [21].

The overflow occurs when data passes over its boundary and writes over the memory, which is not allocated for itself. This error mainly occurs in programming languages that do not use bound checking mechanisms, like C and C++; thus attackers are able to perform such attacks by assigning more inputs than required. Numbered as CWE-121 in the MITRE's CWE list. An example of code containing this fault is presented in Listing 1.

Listing 1. Stack-based buffer overflow.

```
int main(int argc, char *argv[]) {
    char buff[256];
    strcpy(buff, argv[1]);
}
```

The most frequent fault is caused when manipulating strings because the length of strings (in C and related languages) is not an attribute of the string, but strings are terminated by a delimiter (the null character byte). Therefore, most programmers assume normal or "well intended" inputs and do not check the boundaries. It is also hard (in more complex code, not in the one of the example) for the compiler to figure out potential overflows.

As we will see later, the exploitability of a fault greatly depends on the type of data that can be overwritten; stack buffer overflow is one of the simplest to exploit due to the presence of the return address.

2.2. Heap-Based Buffer Overflow

A heap overflow fault is a type of buffer overflow where the buffer being overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as `malloc()` [22,23]. Numbered as CWE-122 in the MITRE's CWE list. Listing 2 shows code containing a heap overflow vulnerability. If the length of `argv[1]` is greater than 256, then the overflow will occur, overwriting heap memory.

Listing 2. An example of heap-based buffer overflow.

```
int main(int argc, char *argv[]) {
    char *buf = malloc(256);
    strcpy(buf, argv[1]);
}
```

The only difference with respect to the stack-based overflow is the zone or area of memory where the overflow occurs. The heap is where long term global variables are allocated. While the stack is well defined and follows an invariant layout, the heap is a much more dynamic structure. The content of the heap depends on the data types used by the application (for example, an application written in C++ with a lot of virtual classes will have a heap full of function pointers), also the exact arrangement will depend on the DSA (Dynamic Storage Allocator [24]) that implements the `malloc()` service. Similarly, the longer the application runs, the more scrambled will be the objects allocated in the heap. Although triggering the error is pretty much the same on the stack as on the heap, all this "entropy" makes far more challenging the exploitation.

2.3. Use of Externally Controlled Format String

Also known as *Format String Vulnerability*, it is caused when the attacker can control the format string (i.e., the first parameter to the `printf()`) to an advanced printing function [25,26]. This memory vulnerability is listed on Common weakness Enumeration list as CWE-134. This type of weakness was discovered at the beginning of this century [27].

The attacker can read and write from any address at will; also, contrary to other forms of overflows, the attacker can peek and/or poke individual memory position, which makes the exploitability of this fault much simpler and reliable. It was compared to having a full debugger attacked on the target process. The `printf()` in listing 3 contains a format string vulnerability.

Listing 3. Format string vulnerability.

```
int main(int argc, char *argv[]) {
    printf(argv[1]);
}
```

2.4. Integer Overflow or Integer Wraparound

An integer overflow or wraparound occurs when the outcome of an arithmetic operation is too large to store in the associated representation [28]. This is related to the way numbers are represented internally. For example, a 32 bit unsigned integer can represent up to 2^{32} values. If the output of an arithmetic operation is larger than the maximum (results in a negative value), the result is truncated. This memory vulnerability is listed as CWE-190 by MITRE for strict overflows and CWE-191 when the error is an underflow.

Many languages consider that integer arithmetic is by default modular and so it is not a bug [29]. If `x` is `unsigned int` then `(x+c)` shall be considered interpreted as `((x+c) mod` 2^{32}`)`, which never causes an exception. An example of a program that causes integer overflows is shown in Listing 4.

Listing 4. Integer Overflow Vulnerability.

```
int  main(int argc, char *argv[]) {
   char buf[50];
   int i = atoi(argv[1]);
   unsigned short s = i;
   if (s > 10) {
       return;
   }
   memcpy(buf, argv[2], i);
}
```

The fault is caused when the programmer does not take into account this behavior and the overflown or overwrapped variable is used to access an array or data structure out of its bounds. On the other hand, some strongly typed programming languages (for example Ada) emit code that checks for overflows and raises the corresponding exception. Although it may seem that this solution produces robust and reliable software, it is not always the case. The first launch of the Ariane rocket by ESA was a failure due to an unnecessary integer overflow exception. The exception occurred in one of the tasks (Ada tasks) (threads) that were no longer necessary, but that was left active to avoid problems stopping individual tasks on a concurrent system. Unfortunately, the exception got propagated to the whole module, which contained more necessary tasks (the fly control), and resulted in a fatal loss of control [30].

2.5. Use After Free

This is also referred to as a dangling pointer [31]. The process uses dynamic memory after it was freed. In order to manifest the error, the program must free a valid object, then allocate a new one (it does not need to be of the same type) and then use the former pointer. Although it may seem an unlikely scenario, there are two common situations: Error conditions and other exceptional circumstances that free memory but still perform some actions with the data or confusion over which part of the program is responsible for freeing the memory. This memory vulnerability is listed on the Common weakness Enumeration list as CWE-416, and an example of use after free vulnerable code is showed in listing 5.

Listing 5. Example of use after free exploitation.

```
int main(void){
    char *pt_1,*pt2;
    pt_1 = malloc(100);
    free(pt_1);
    pt_2 = malloc(100);
    strncpy(pt_1,"HELLO",100); <- Use pt_1 after being freed.
    printf("%s\n", pt_2);
}
```

In this example, in most cases (depending on the DSA algorithm) the output will be the string "HELLO". Dynamic memory is allocated in the heap, in this sense it can be considered to be a form of heap buffer overflow with additional limitations due to the amount of data that can be written (only the size of the new allocated object).

2.6. NULL Pointer Dereference

The application access (eg, dereferences) a pointer that points to the 0x0 address [32]. If this pointer is used to redirect the execution flow, then on modern systems it will raise a processor trap terminating the execution. This is because the page zero is not mapped for security reasons to prevent attackers from abusing null pointer dereference to control the execution flow. This memory vulnerability is listed on Common weakness Enumeration list as CWE-476.

3. Exploitation

In this section, we present several attacking techniques used by adversaries to subvert the control-flow. The nature of exploitation strategies and their ability to perform the attacks are discussed. First, we summarize **how** the attacker takes control of the instruction pointer, and then **what** can be executed.

3.1. How to Subvert the Control Flow?

We will focus on control flow instructions that could be manipulated by the attacker. For example, constant (direct or relative to instruction pointer) jumps or calls are not considered because they always follow the code flow programmed by the developer. Our target are instructions that use data to determine the actual destination.

3.1.1. Indirect Jump

The destination address is not embedded in the opcode but read from a memory address or is the value of a processor register.

This kind of instruction is commonly found in the supporting structures (PLT Procedure Linkage Table) used by the loader to bind the application with the system libraries. All applications that use dynamic libraries call the library services using indirect jumps. Listing 6 shows examples of indirect jump instructions. These instructions are also used to implement context switches in user space. For example, signal and setjmp/longjmp mechanisms.

Listing 6. Indirect jump instructions.

```
jmp     QWORD PTR [rip+0x21c4cc]
jmp     QWORD PTR [rax+0x10]
jmp     QWORD PTR [rdx]
```

Indirect jumps are prone to abuse since the destination address can be manipulated by the attacker. For example, an indirect jump or call that reads the destination from a read only memory area cannot be exploited. In the previous example, `[rip+0x21c4cc]` must point to a writable area in order to be exploitable.

3.1.2. Indirect Call

Indirect calls are the equivalent of the previous ones but rather than a jump, the opcode corresponds to a call instruction. Listing 7 shows three indirect call instructions.

They are used to implement virtual functions in "C++". Function pointers are stored in the *vtable* array, which points to the actual methods of the object. This way, a class can inherit some methods from the base class and override virtual functions.

Listing 7. Indirect call instructions.

```
call    QWORD PTR [rip+0x28e529]
call    QWORD PTR [rax+0xe0]
call    QWORD PTR [r12+rbx*8]
```

3.1.3. Return

Overwriting the return address is the classic exploitation technique for most stack-buffer overflows. The Application Binary Interface (ABI) of most systems define that the return address shall be stored in the stack when a function is called, and retrieved and used to come back to the caller upon termination of the function. For this reason, the stack was, and still is, both very appealing to attackers and security researchers to develop protection techniques. Listing 8 shows the end of a function with a return (`retq`) at the end. A complete example of this exploitation was presented at the Black Hat conference in 2018 [8].

Listing 8. Return instruction.

```
pop     %r12
pop     %r13
pop     %r14
pop     %r15
retq    <-- Jumps to the address located on the stack
```

3.2. What Can Be Executed?

Once the attackers know how to manipulate the control flow, the second step is to use it to execute their malicious code. There are basically two strategies: (1) inject code and (2) reuse code.

3.2.1. Code Injection

The attacker uploads or injects malicious code into the target process and transfers the control flow to the entry point to its code [33]. Most processors differentiate between accesses for reading or writing data and accesses for executing code (the NX protection technique, which will be described). Although it is possible to change the memory access permissions, in order to do it, the attacker must have already full control of the process. Code injection is commonly used as the second stage of the intrusion to upload the advanced payload. There are still a few places where the attacker can upload and execute code directly:

- Just in Time Compilers (JIT). Most web browsers do not interpret JavaScript code but compile it, on the fly, to native instructions and execute the resulting code. This is done using pages that have read/write/exec permissions.
- Backward compatible applications. Some applications use old versions of libraries that need to execute code snippets on the stack. There are also one advanced "C" constructions that force the compiler to execute code on the stack.

3.2.2. Code Reuse Attack

Code-reuse attacks (CRA) are a family of techniques which rely on reusing the already existing on the target process [5]. Typically, in these types of attacks, the attacker has the opportunity to inject or modify data on the target and redirect the control flow to code that is already in the target.

Return-to-libc: The attackers redirect the control flow of the libc function of their choice (typically `system()`) [34]. As well as the instruction pointer, the attacker also needs to pass the desired parameters to the function. The way the parameters are passed depends on the ABI, and although the ABI plays an important role in the exploit development (and the feasibility of the exploit itself), it is beyond the scope of this paper.

Return Oriented Programming: The attacker redirects the control flow to sequences of instructions ending in a `ret` instruction. If the attacker is able to retake the control flow using the ending return instruction, then they can chain multiple sequences in the precise order to execute the desired payload. The sequence of instructions that ends in a `ret` instruction is called a *gadget*. ROP programming consist of finding and arranging gadgets [3,4]. It has been shown that the Gnu libc library contains enough gadgets to build a Turing-complete machine. In other words, it is possible to build any program using glibc gadgets. In fact, there exists multiple open source projects to extract useful gadgets (those that perform single and easy to use instructions). As we will see in the protection techniques section, ROP is so powerful that it has influenced the type of code that the compiler emits at the end of the functions.

GOT Overwriting: To build a ROP, the attacker needs to have access to a large amount of code (the attacker needs to have enough gadgets). This code is typically located in the libc. However, if the library is protected with the ASLR, then it is not possible to use it. Global Offset Table (GOT) overwriting [35] uses only a few ROP gadgets from the application code (which in some cases is not randomized) to allow the attacker to use the return-to-libc strategy. The idea is to de-randomize the library, i.e., discover the actual location of the library, using information that is already available to the application code. This information is located in the GOT. The GOT is used for both determining the address of the libc, and as a trampoline to jump to it (typically the `system()` function).

Jump Oriented Programming: To build a ROP program, the attacker needs to control the content of the stack, or use some form of stack pivoting. To overcome this limitation JOP [5] replaces the use of `ret` instructions by a dispatcher gadget, based on indirect jump instructions, to dispatch and execute the sequence of gadgets.

Sigreturn: Another Turing-complete technique for reusing snippets of code. The Sigreturn [6] technique relays the `sigreturn()` system cal, which is used by the glibc to restore the state of the processor after servicing a signal. Loading the content of all the processor registers with attacker data is very convenient, but the information about the state of the process is lost (stack pointer or other registers containing pointers), because all the registers are restored. As the authors [6] pointed out: " we believe that sigreturn oriented programming is a powerful addition to the attackers' arsenal".

Speculate execution: In 2018 a new range of weaknesses were discovered that affected all advanced processors [36–38]. These vulnerabilities can be used to create attacks which are very similar to those that can be used against classical memory errors. Also, most of the generic CFI techniques are also effective. These weaknesses can be abused by the attackers to force the processor to execute unintended code using various forms of speculative execution and retrieve the results of the execution via timing side channels (on the caches or some internal elements of the processor). The only difference with respect to the classic control flow attacks is that these are *transient* control flow attacks. Since the processor retakes the correct control flow of execution after the speculated or transient sequence of code, these attacks cannot be used to take the control of the process or system but only exfiltrate information [39].

4. Classification of CFI Techniques

Control-Flow Integrity is a defense policy which restricts unintended control flow transfers to unauthorized locations [40]. It is able to defend against various types of attacks whose primary intention is to redirect the execution flow elsewhere. Many CFI techniques [41–48] were proposed over the past few years. However, they were not fully adopted due to practical challenges and significant limitations.

The CFI techniques are primarily classified as fine-grained and coarse-grained. Both classifications comprise advantages and drawbacks in-terms of control-flow security. The development and research of each particular CFI techniques were mainly focused on either fine or coarse-grained approach with justification why a particular approach was adopted. In this section, we discuss both approaches to understand the security level.

4.1. Fine-Grained CFI

Fine-grained CFI is referred to as the strict type CFI [48]. Labeling is one of the most usual approaches to implement fine-grained CFI. The edges of a control-flow transfer contain labels, and they are checked for integrity on each control transfer. Prior to control flow transfer, a validation process ensures that the transfer is passed to a valid destination which contains a valid label.

Figure 1 shows an execution flow where the fine-grained approach is initiated. The execution flow consists of both direct calls and indirect calls. The direct call is fixed and cannot be altered, whereas the target address is calculated before returning the indirect call. Indirect return 4 is valid and also returns to the valid destination. However, an attack is initiated at the same return point overwriting the return address, and the attacker manages to divert the flow elsewhere.

Fine-grained CFI usually enforces a shadow stack which incurs intense overhead [47]. Therefore although it enhances sturdy security, the drawback is a high-performance overhead. It can be complicated to achieve the fine-grained CFI because the destination of indirect branches needs to be determined statically. Fine-grained CFI does strict checking of each call and jump instruction inside of an application; therefore, it lacks the performance and is very expensive at the same time. To cope with the performance issues, a weaker form of CFI, coarse-grained, was proposed.

4.2. Coarse-Grained CFI

Coarse-grained CFI is referred to as loose CFI [48]. Coarse CFI distinguishes indirect branch instructions by their classification and imposes policies on each type. Loose CFI checks when a control-flow transfer commences with a return instruction so that the destination can be targeted right after the call instruction. This CFI approach is more relaxed and secures desirable performance, but proven to be dangerous [47]. Though it does not experience high-performance overhead; however, coarse-cfi has a limitation in obtaining details of application [49].

Figure 2 shows a coarse-grained CFI approach where the execution flow consists of direct and indirect control-flow transitions. Both direct and indirect call/ret instructions from 1–4 are valid and verified by coarse CFI. However, the indirect call, `call *eax`, is a pointer; therefore, besides going to its valid destination function `f2()`, attackers are also able to divert the flow by illegal jumps to function `f1()` and initiate a control-flow attack.

Coarse-grained focuses on if many unessential edges exist, whereas a fine-grained CFI intensifies a fully precise static CFI. A recent investigation reveals that coarse-grained CFI is not fully defensive against control-flow attacks, whereas fine-grained CFI is presumed to be more secure. Carlini et al. [45] demonstrate that coarse-grained CFI is not protective enough to defend against small or even simple applications. However, Muench et al. [48] and Davi et al. [50] argue that compare to fine-grained CFI approach; the coarse-grained enforcement method is enough to protect against CRA.

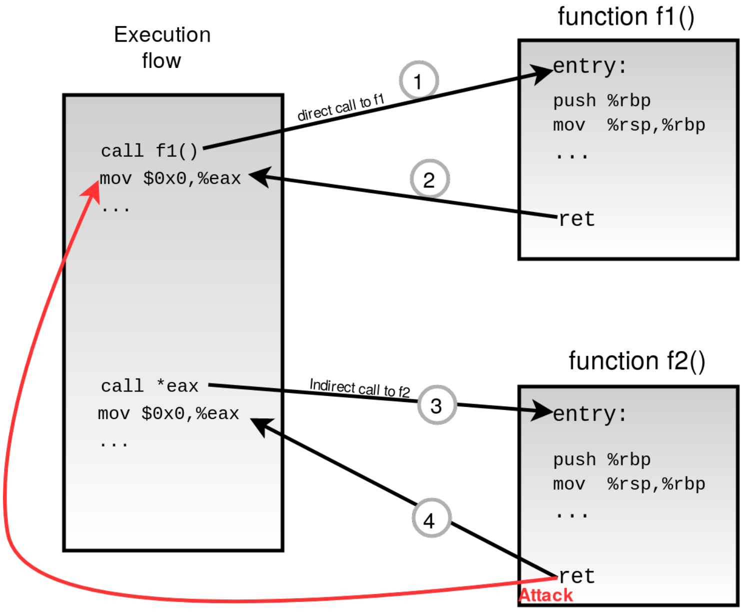

Figure 1. Fine-grained CFI approach bypass example. The attacker can successfully redirect (arrow in red) the control flow from the ret instruction in step 4 to the mov instruction next to `call f1()`.

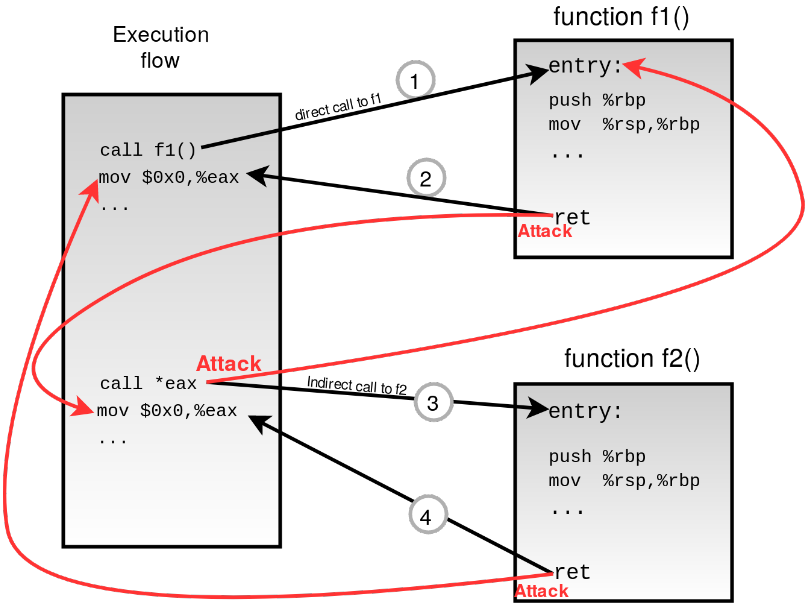

Figure 2. Coarse-grained CFI approach bypass example. The attacker is able to redirect the control flow from multiple `call()`s and `ret` points. Arrows in red show the valid destinations where attackers can redirect the flow.

5. Protection Techniques

In this section, we discuss the most relevant CFI techniques that can be used to prevent control-flow hijacking. In our discussion, we also point out the limitations associated with individual techniques.

5.1. Control-Flow Integrity (CFI1)

Abadi et al. [51] proposed CFI, the first CFI proposal for control-flow security implementation. To avoid confusion we call this technique as CFI1. They have implemented inlined CFI for windows on the ×86 platform. Their work [52] suggests initiating a Control flow graph (CFG) before program execution. The CFG monitors run-time behavior; hence, any inconsistency in the program results in a CFI exception being called and terminate the application. The CFI instrumentation does not affect direct function calls, but only indirect calls require an ID-check. Functions that are called indirectly, such as virtual methods, need insertion of an ID. This CFI technique ensures that program control is transferred to a valid destination. Thus, whenever a machine code transfers its control, the destination needs to be valid as well as predefined by CFG.

However, Davi et al. [50] pointed out three important limitations of this technique. First, the source code is not always available. Second, binaries lack the required debug information and finally, it causes high execution overhead because of dynamic rewriting and run-time checks. It is also unable to cope with dynamic code or binaries which contravene compiler conventions [53]. In addition, Clercq et al. [20] shows that the CFG is unable to determine if a function returns to the current caller. The problem also occurs when a function is called from more than one place, for instance, in a regular application, a function can be called from one place but if the return address is overwritten then the called function may return together with another CFG edge to an unknown call point.

5.2. CCFI: Cryptographically Enforced CFI (CCFI)

CCFI possesses new pointer arrangements, which cannot be imposed with static approaches [46]. It comprises two prime attributes. First, it recategorizes function pointers at run-time to boost typecasting. Second, it restricts swapping of two valid pointers which consist of the same type. Static analysis does not resemble dynamic characteristics; hence, CCFI involves dynamic CFI mechanism which uses cryptographic primitives. It is a fine-grained compiler enhancement where Message authentication codes (MACs) are implemented on every control flow pointer and impose various run-time, as well as language attributes. CCFI is designed to protect return addresses, frame pointer, function pointers, vtable pointers, and exception handlers. A MAC key produces random values at the beginning of a program to be stored in the registers. Random offsets are added to each malloc and stack frame in order to randomize each memory and stack frame.

Nevertheless, with all the implemented features, CCFI comprises an average overhead of 52% on all benchmarks and is vulnerable to replay attacks [48]. CCFI only detects calls to print doubles function. It is not address taken and comprises dissimilar non-variadic prototype than square. CCFI permits all types of forbidden calls [54]. Besides reporting the highest overhead compared to other CFI techniques, it requires the Operating Systems to maintain and restore the MAC key while the context switches take place [55]. CCFI is unable to protect function pointers studded in C structures [56]. It also fails to identify structure pointers, and it is possible to disrupt the control flow by altering the current pointer with the old pointer. CCFI mainly focuses on defending the user-level program and does not include kernel-level security [46]. The security features are not focused on protecting the heap and stack addresses. However, to overcome such attacks, randomness can be implemented by injecting entropy, causing a high memory consumption.

5.3. CFI for COTS Binaries (BinCFI)

In this technique, CFI is applied to stripped binaries on ×86/Linux architecture. It is experimented over 300 MB of binaries to verify the reliability and does not require symbol, debug or relocation information. Instrumentation method was applied to every shared library and executables in a way that full transparency could be ensured. Since obtaining full transparency can be challenging, a new optimization technique was developed. A simple metric, Average Indirect target reduction(AIR), has also been proposed to measure the elimination of indirect transfers. Another major aspect of binCFI is that it does not make any assumptions in regards to indirect Control-Flow targets, rather it is based on static analysis so that it can enumerate large executables and binaries.

binCFI involves implementing CFI to the shared libraries; for instance, glibc [42]. It focuses on overcoming the drawbacks, which are highlighted by the static analysis technique. According to Niu et al. [44], bin-CFI permits a function to return to every viable return address; hence, the accuracy of this CFI is fragile to Return-Oriented programming (ROP)-based attacks. AIR is not a very suitable metric while considering it from a security perspective [57]. Two main reasons are; First, every CFI mechanism produces similar AIR numbers; hence, making the AIR improper when compared with other CFI mechanisms. Second, a high reduction in the number of targets still comprise enough targets which give attackers a chance to exploit. Various experiments demonstrated that call-preceded gadgets are able to bypass the CFI technique that limits itself to only control-flow transfer checks prior to unsafe function calls [47]. binCFI enhances strong compatibility; however, it fails to restrict all types of control-flow attacks [58].

5.4. Practical CFI and Randomization for Binary Executables (CCFIR)

CCFIR gathers the legitimate target of indirect branch instructions and places them randomly in a "Springboard Section" [59]. CCFIR restricts indirect branch instructions and permits them only towards white-list destinations. It is able to differentiate between calls and returns and also restricts undefined return to critical functions. In order to protect return addresses and function pointers, the binaries require to be disassembled first and then figure out the source and implementation of targets so that further protection can be enhanced.

The average execution overhead of CCFIR is from 3.6% to 8.6%. CCFIR uses three ID's for each branch instruction, excluding the shadow stack. Unfortunately, CCFIR is not free from flaws, for example, attacks using ROP chains can bypass it [47]. Security drawbacks in CCFIR allow attackers to use the control flow transitions to execute CRA. In addition to that, the springboard consumes a lot more memory [60]. Although CCFIR comprises better performance but unable to provide enough security to mitigate control-hijacking attacks [58].

5.5. Hardware (CFI) for an IT Ecosystem (HW-CFI)

HW-CFI is a security proposal by NSA information assurance [41]. They put forward two notional features to enhance CFI. One of the features recommends implementing CFG to hardware. It pre-calculates a bitmap to overlay the address space where a 1 will be considered to be an authorized branching root, and a 0 will be defined as unauthorized. The other feature protects the dynamic control-flow by a protected shadow stack. The return address of a function call will be saved at both the stack area and shadow stack. While returning to the caller, both of the addresses will be matched for verification. Any dissimilarity will result in a CPU fault and cause the application to terminate. Since software bugs are likely to corrupt data and result in possible threats, the notable aspect of this CFI technique is that data in the shadow stack will be protected by hardware features.

This CFI proposal is a notional CFI design and might not be compatible with all system architecture. Although shadow stack can be an excellent option, monitoring shadow stack explicitly might not often be possible.

5.6. Per-Input CFI (PICFI)

PICFI imposes computed CFG to each input. It is certainly difficult to consider all inputs of an application and CFG for each of the inputs. Therefore, PICFI runs an application with empty CFG and lets the program discover the CFG by itself [44]. The main idea behind this empty CFG approach is to affix edges on run-time prior to being used for branch instructions. PICFI restricts adding edges which were not defined during the static computation of CFG. To deal with the performance overhead PICFI comprises performance optimization techniques. PICFI consists of an overall run-time overhead as low as 3.2%. PICFI statically computes CFG to determine the edges that will be added on run-time and implements DEP to defend against code injection.

Statically computed CFG does not produce a proper result, and various experiments prove that DEP is by-passable. PICFI only relies on underlying static analysis and can be affected by label creep by obtaining a vast number of labels from CFG [57].

5.7. KCoFI: Complete CFI for Commodity Operating System Kernels (KCoFI)

KCoFI ensures protection for commodity operating systems from attacks, such as ret2usr, code segment modification [61]. KCoFI performs its tasks in-between the stack and the processor. KCoFI includes a conventional label-based approach to deal with indirect branch instructions. KCoFI is not based on full program analysis or sophisticated total memory protection. A methodical model of kernel execution, which includes small-step semantics, was developed to facilitate tasks such as address translation, trap handling, and context switching.

KCoFI introduces about 27% of overhead when transferring files between 1 KB and 8 KB. On smaller files, the average overhead is around 23%. KCoFI fulfills all the requirements of managing event handling, but the outcome of this CFI enforcement is too expensive, over 100% [62]. KCoFI is unable to mitigate the control-flow attacks to the kernel, and it is demonstrated that the adopted coarse-grained approach is not enough to enhance proper control-flow security [63].

5.8. Fine-Grained CFI for Kernel Software (Kernel CFI)

Kernel CFI implements retrofitting approach entirely to FreeBSD, the MINIX microkernel system, MINIX's user-space server and partially to BitVisor hypervisor [62]. This CFI technique follows two main approaches for CFI implementation. First, Kernel source code contains function pointer usage patterns, which can be used to produce a technique that helps compute CFG for kernel software. Second, the execution of two methods in order to expel CFI instrumentation that is connected to earlier optimization technique. The average performance overhead ranges from 51% to 25% for MINIX and from 12% to 17% for FreeBSD systems.

Although Kernel CFI technique cuts down the indirect transfer up to 70%, unfortunately, there is still a chance for indirect branch instructions to transfer control to an unintended destination.

5.9. Enforcing Forward-Edge CFI in GCC & LLVM (IFCC)

Forward-Edge CFI is a fine-grained enforcement CFI implementation which involves analysis for GCC and LLVM compiler to meet particular CFI requirements. Indirect Function-Call Checks (IFCC) is a CFI transformation mechanism which is put into practice over LLVM 3.4 compiler. IFCC introduces a dynamic tool which can be used to analyze CFI and locate forward edge CFI vulnerabilities [43]. Its implementation policy primarily concentrates on three compiler-based mechanisms. First, it focuses on protecting data that is stored in the heap, second, it enforces forward edge CFI by enhancing two of the implemented mechanisms, and finally, it takes advantage of the run-time analysis tool which is designed to detect control-flow hijacking in the early stage of software development life cycle.

Nevertheless, IFCC fails to protect against control Jujutsu attack, a fine-grained attacking technique which aims to execute malicious payload [64]. Another major limitation of this CFI mechanism is that it fails to verify when a function returns back to a destination [49]. An experiment was conducted using ASICS discovery tool [64]. The exploit was expanded as well as a full exploit was developed. However, IFCC was unable to detect that exploit.

5.10. Control-Flow Bending: On the Effectiveness of CFI (CFB)

CFB comprises static CFI implementation [45]. The authors introduced a generalization of non-control-data attacks are referred to as CFB. For instance; if arguments are overwritten directly, then that is considered to be a data-only attack as it did not require to invade the control-flow for such operation, but if the overwritten data is non-control-data, then it has affected the control-flow. The non-control-data attacks do not involve in modifying control-flow data; hence, this CFI technique allows such modifications until the targeted indirect branch is viable and follows the CFI policy. CFB can be bind to an attack which establishes the control-flow transfer within a legitimate CFG.

CFB implements a fully precise static CFG, which can be undecidable [64]. Static CFI analysis comprises significant drawbacks as they fail to determine every feasible value of a function pointer [46]. CFB also violates certain functions at a high level and execution of such functions likely to alter the return address and corrupt control-flow [44].

5.11. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks

SAFEDISPATCH provide defenses against vtable hijacking. It examines C++ programs statically and carries out a run-time check to ensure that the control-flow at virtual method call sites is not hijacked by attackers [65]. SAFEDISPATCH is an enhanced C++ compiler and is built based on Clang++/LLVM. To identify the effectiveness, SAFEDISPATCH was examined by implementing on Google Chromium web browser code base. This CFI technique conducts static class hierarchy analysis (CHA) to ensure that each C class of a program is invoked by a static object which is type C. The information is then used to perform instrumentation for dynamic checks. The run time overhead of SAFEDISPATCH is 2.1% and the memory overhead, around 7.5%.

SAFEDISPATCH is unable to protect binaries, which makes it vulnerable to ROP-based attacks [47]. Since various programs frequently use shared library or dynamic loaded libraries, all the compiler-based fine-grained illustration undergoes common security problems associated with shared libraries [48]. SAFEDISPATCH can be vulnerable to various attacks as it is unable to protect binaries [47].

5.12. Control Flow Guard (C-Guard)

Control Flow Guard is a highly implemented security mechanism developed by Microsoft to defend against different types of memory error vulnerabilities [66]. It protects against memory error exploitations by enhancing strong security where the program executes its codes. While Control Flow Guard is placed, arbitrary code execution is quite hard to initiate through common vulnerabilities such as buffer overflows.

C-Guard is unable to verify when a function returns to some unauthorized destination [49].

5.13. Reuse Attack Protector (RAP)

RAP is preventing return-oriented programming technique, implemented as a GCC compiler plugin. Developers do not have to use a reformed compiler to use RAP [49]. RAP has a commercial version, which comes with two prime defense mechanisms to protect against control-flow attacks. First, it limits what kind of functions can be called from a particular place as well as what locations a function can be returned to from that place (function). Second, a function will only be able to return to a location from which it was called. RAP uses Linux kernel at compile time to implement strict CFI at run-time and assures that code pointers are not corrupted by the adversary. However, it does not have a solid protection against attacks, such as ret2usr [67].

RAP's implemented approach is very much similar to the traditional label-based approach. Label based CFI suffers from security issues as a function could return to any call site. Experiments show that the open-source version of RAP comprises limitations and can be exploited successfully [68].

5.14. Opaque CFI (O-CFI)

O-CFI comprises binary software randomization for CFI enforcement [47]. It protects legacy binaries without even accessing the source code. O-CFI adopts a novel combination of fine-grained and coarse-grained approach to deal with the control-flow challenges. It is able to obscure the graph of control-flow edges from adversaries who gains full access to view stack, heap, and other program information. An experiment demonstrates that O-CFI is able to provide significant protection against code-reuse attacks. CFI checks are done on Intel ×86/×64 memory protection extensions, MPX [69], which is hardware driven. It consists of a performance overhead of only 4.7%.

O-CFI involves static binary approach; therefore, it fails to protect code that is generated dynamically. It is incompatible with the Windows Component Object Model(COM). Moreover, MPX is much slower than software-based implementation and not protective against memory-based errors. O-CFI fails to commit a precise CFI. It is also unable to solve the challenges related to CFI implementation for COTS modern windows application [53].

5.15. CFI Key Features Summary

As we have previously analyzed, each CFI protection technique can be implemented on hardware and software, use different approaches such as shadow stack, use labels and even be coarse and fine grained. In order to summarize all those features, we present in Table 1 the key features being adopted by the 14 CFI protection techniques analyzed.

Table 1. Key features of CFI techniques.

CFI Techniques	Based on		Compiler Modified	Shadow Stack	CFG	Label	Coarse Grained	Fine Grained	Backward Edge	CFI Enforcement
	HW	SW								
CFI1 [51,52]		✓		✓	✓	✓	✓		✓	Inlined CFI
CCFI [46]	✓	✓	✓					✓	✓	Dynamic Analysis
binCFI [42]		✓				✓	✓			Static Binary Rewriting
CCFIR [59]		✓				✓	✓		✓	Binary Rewriting
HW-CFI [41]	✓	✓	✓	✓	✓	✓		✓	✓	Landing Point
PICFI [44]		✓	✓		✓			✓	✓	Static Analysis
KCoFI [61]		✓	✓			✓	✓		✓	SVA Instrumentation
Kernel CFI [62]		✓			✓			✓	✓	Retrofitting Approach
IFCC [43]		✓	✓		✓			✓		Dynamic Analysis
CFB [45]		✓		✓	✓			✓	✓	Precise Static CFI
SAFEDISPATCH [65]		✓	✓				✓			Static Analysis
C-Guard [66]		✓	✓		✓		✓			Dynamic Instrumentation
RAP [49]		✓	✓		✓			✓	✓	Type Based
O-CFI [47]	✓	✓					✓	✓	✓	Static Rewriting

The summary reveals that all CFI protection techniques are based on software and only three of them are also implemented in hardware. In addition, most of them need compilers modifications and perform an offline Control-Flow-Graph (CFG) of the application. Regarding coarse vs fine-grained approaches, around 50% of techniques adopted coarse-grained and the other 50% adopted fine grain. This shows that there is not a clear inclination about any of those by the security community.

6. Analysis

In this section, we assess the effectiveness of both software and hardware-based CFI techniques against various attack vectors. Our assessment involves analyzing 11 software and 3 hardware based CFI techniques which predominantly focused on protecting the program control but thoroughly failed to protect from exploitation. We evaluate whether the CFI techniques are by-passable by narrowing our focus towards a few attack vectors such as CRA, code-injection, disclosure, ret2usr, ret-to-libc, and replay attack.

A technique with extensive weaknesses likely to receive denial from the industry and in order to identify whether the implementations are affordable, realistic and based on practical implication, it is essential to classify them based on hardware and software as they follow dissimilar approaches in terms of implementation, security and performance. Our assessment involves surveying over security experiments done by various research groups, and then putting them together to define the flaws in particular CFI techniques.

6.1. Software-Based CFI

Software-based CFI enforcement primarily focuses on program instructions, which are corrupted by indirect branch instructions.

Table 2 shows that CFI [51] does not comprise strong protection, and according to Chen et al. [70], it is possible to execute ROP-based attacks while this technique is deployed. It permits a function to return to any call site; hence, it can be exploitable [47]. Although bin-CFI does not comprise a shadow stack policy and uses ID/label for each branch instructions. However, Mohan et al. [47] illustrate that the most recent experiments prove that label-based approaches are also vulnerable to ROP-based attacks. It can also be bypassed by gadget synthesis attack. Although CCFIR enhances a security policy that restricts indirect branch instructions to predefined functions; however, it misuses external library call dispatching policy in Linux and also causes boundless direct calls to critical functions in windows libraries which can be exploited. Moreover, the springboard section can be exploited by disclosure attacks [50]. CCFIR can also be exploited by Loop attack and Gadget Synthesis attack. PICFI lacks security, and the control-flow can be compromised by performing three different attacking stages illustrated by [71]. KCoFI and Kernel CFI are both kernel focused CFI techniques. KCoFI depends only on the source code; therefore, ensuring minimal protection to binaries. It also does not provide stack protection; hence, it is exploitable by various memory error vulnerabilities, such as CRA and memory disclosure [70]. Besides that, Kernel CFI is a fine-grained CFI implemented on MINIX and FreeBSD. This technique builds a minimal challenge to defend against ROP-based attacks by having the control-flow diverted to a valid execution site [72]. Table 2 also shows that IFCC is unable to mitigate control-flow exploitation and can be by-passable by Control Jujutsu Attack, which allows an attacker to perform arbitrary code execution [64]. Although CFB enhances strong CFI enforcement having shadow stack implemented; however, it is shown that it can be by-passable by CFG-Aware Attack [73]. SAFEDISPATCH is a compiler-based CFI enforcement focusing on securing indirect calls to virtual methods in C++; however, it can be subverted by CRA such as ROP [50]. Control Flow Guard has been a very significant implementation of Microsoft. It was deployed as an effective security technique to defend against memory corruption attacks; however, Control Flow Guard fails to protect against indirect jumps. Moreover, it is fully by-passable by Back To The Epilogue (BATE) attack [74]. BATE can corrupt the control-flow and direct it to an undefined location. RAP a fine-grained CFI technique for linux kernel, has already been adopted by the industry. RAP makes it very hard for a ROP chain to be built up; however, it is unable to provide security against ret2usr attacks [67].

A weak CFI enforcement is prone to various types of exploitation and results in potential control-flow attacks. We show that all the 11 software-based CFI approaches are by-passable; however, each technique has imposed a different degree of security policy. Hence, the effort needed to bypass individual techniques is not the same.

Table 2. State of the art of the attacks bypassing CFI.

CFI Tech.	Code Reuse	Code-Injection	Disclosure	Return2user	Replay	Gadget Synthesis	Loop Attack	Jujutsu
CFI1	[70]		[70]					
CCFI	[75,76]				[48,77,78]			
binCFI	[44,47,79]					[53]		
CCFIR	[60,79,80]		[50,81]			[53]	[82]	
HW CFI	[83]							
PICFI	[71]							
KCofi	[70]		[70]					
Kernel CFI	[64], [72]							
IFCC								[64]
CFB		[73]						
SafeDispatch	[47,50]							
C-Guard	[84]	[74]						
RAP				[67]				
O-CFI	[85]							

6.2. Hardware-Based CFI

Hardware-based CFI enforcement requires the system to have hardware-based components in place for deployment. Our assessment involves 3 CFI techniques, which are implemented in both hardware and software and shows that they can be subverted by replay attack and code-reuse attacks. Hardware implementation is an expensive option as it might not be compatible with the running system; hence, the transformation of the whole system may be necessary. CCFI requires AES-NI implementation besides compiler fulfillment. A security experiment suggests that AES-NI can be exploited by replay attack [48]. Transformation of the whole IT ecosystem is essential to implement HW CFI for IT ecosystem. However, such implementation is near to impossible and too expensive for deployment. This hardware-based technique comprises shadow stack to protect backward edge and landing point instructions for indirect branch transfers. In the context of security, this CFI technique will fail to mitigate indirect branch transfers when implemented. An adversary will be able to direct forward edge to any landing point instruction causing control-flow corruption [83]. Memory Protection Extension(MPX) is adopted by O-CFI; however, MPX is not a quick approach and hits 4× slow down compared to the software approach. O-CFI can also be exploited by function-reuse attacks [85].

6.3. Overhead

Overhead plays a significant part in CFI implementation, and, in most cases, helps determine whether a particular feature will be adopted for CFI implementation. Besides complexity, inefficiency, and weak features, most CFI techniques reported very high-performance overhead. Figure 3 presents the maximum performance overhead of 10 CFI techniques along with the number of discrete exploitations can be executed.

Distinct CFI techniques use different platforms to measure execution, performance, and space overhead. However, we narrow our focus only towards the maximum performance overhead. Both CFI, KCofi causes overhead of 45% and 27%. Both CFI techniques can also be exploited by 2 different attack vectors. Kernel CFI causes the highest overhead among all the software-based CFI techniques. We show that Kernel CFI can be exploited by only 1 attack vector. Although bin-CFI consists of a considerable amount of overhead but 2 different attacking techniques can bypass this CFI technique. Hardware-based enforcement, CCFI, comprises 52% overhead raising the question if hardware implementation is cost-effective beside software implementation. CCFI can be exploited by more than one attacking technique, and its effectiveness has also been experimented by various research groups. CFI enforcement with such amount of overhead is not accepted and must receive a denial.

However, O-CFI another hardware-based implementation comprises only 4.7% overhead and can be bypassed by only 1 exploitation type. Compiler implemented CFI approaches, such as PICFI, IFCC, SAFEDISPATCH comprise very low overhead too and also be bypassed by 1 exploitation. CCFIR also consists of very low overhead, 8.6%. However, this technique can be exploited by 4 different attacking techniques such as Code reuse attack, Disclosure attack, Gadget Synthesis, and Loop attack. Our research indicates that CCFIR has also been experimented by the most number of research groups to show that it is vulnerable and can be bypassed by different attack vectors.

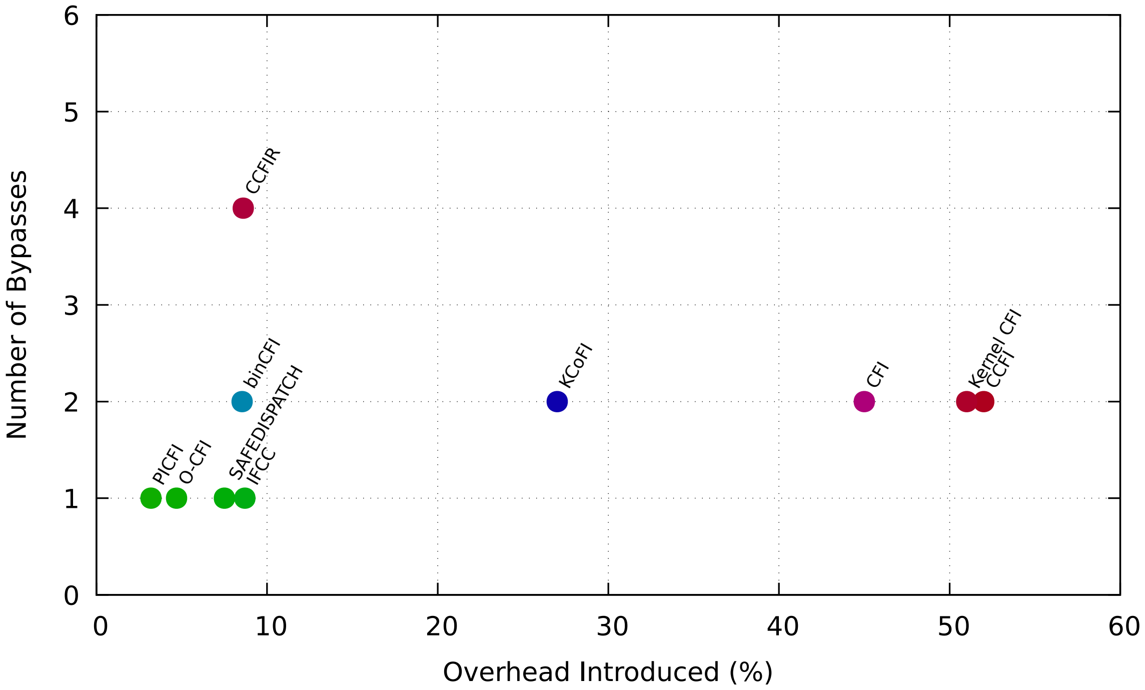

Figure 3. CFI techniques with attack vector counts versus overhead.

7. Discussion

It is clear that CFI is a strong security mechanism against control-flow hijacking. However, we show that the available CFI techniques are bypassable and comprise major limitations; hence, they are very much prone to being compromised. Besides security, overhead plays an important role in deciding whether a particular technique will be adopted. Our analysis also shows that most of the techniques comprise high overhead and also can be bypassed by different types of attack.

In this section, we discuss the potential reasons for all the major CFI techniques being bypassable. In Section 6, we classify them revealing that software CFI approaches are much more simple, and can be deployed without making any deep architectural changes. Whereas hardware CFI must meet architectural requirements, and mostly based on impractical implication. However, regardless of hardware or software, both approaches are based on similar security methods. For instance, static and dynamic, shadow stack approach is adopted by the majority of the CFI techniques.

Static analysis occurs before the execution, in a phase which writes object code, unable to verify if a function pointer is pointing to some destination on run-time. Also, dynamic analysis observes the run-time state of the program [86]; however, it often generates false positives and false negatives. Most software-based CFI techniques are dependent on the dynamic instrumentation where the source code to be inserted into a program to carry out CFI assessment for indirect branch instructions [20]. The process is likely to involve compiler modification or dynamic library translation.

On the other hand, shadow stack was adopted by major CFI techniques. Its implementation enhances the ability to protect a backward edge and has hardware support by Intel [87]. However, shadow stack suffers from serious compatibility issues. It requires having an additional stack and also it consumes extra space resulting in an expensive approach to deploy. Another example that can introduce serious limitations to the shadow stack approach is when the application uses the `longjump` function. The return address is saved in a buffer that is not visible by the shadow stack and therefore the `longjump` return will be detected as an attack but actually it is not. Signal handlers such as `sigreturn` will also introduce a challenge to protection mechanisms based on hardware that hid secondary stacks.

Moreover, several CFI techniques have been implemented in compilers such as GCC and LLVM (low-level virtual machine). LLVM is a compiler framework and written in C++. Clang is related to LLVM and uses LLVM as its back-end. LLVM is a library which is used to build, optimize, and create binary machine code. Clang comprises distinguish CFI schemes implemented on itself and each scheme are outlined to perform specific tasks and terminate a program when unrecognized behavior is detected. Experiments suggest that GCC5.4.0 and clang 3.8.0 fail to detect control-flow exploitation [54].

Beside the efforts to have a robust CFI technique, unfortunately most of them have not been adopted to fight against control flow hijacking mainly because the overhead introduced and the lack of full protection.

8. Conclusions

The CFI issue has been an active research area since the beginning of this century, and it is still an open issue. In this paper, we surveyed 14 major CFI techniques.

Each technique analyzed comprises severe limitations and can be subverted by various attack vectors. Common features such as shadow stack, CFG, code analysis, and label are adopted in some major techniques; hence, the flaw still remains. We identified that software-based techniques are more secure compared to hardware-based techniques and also based on practical implications. They are easier to implement and do not require an improper architectural requirement.

Based on the performance overhead and number of bypasses of each CFI technique, it was identified that there are some techniques that should be avoided in favour of a few that introduce much less overhead and are less bypassable. Most CFI techniques are dis-functional to provide proper security, and as a result, they were not fully acquired by the industry and hence, there are not any particular technique that provides full defense.

Therefore, based on our analysis, we conclude that there is still great room for innovation and research in the near future and it is expected that new solutions or a combination of existing ones will be presented to successful and efficiently provide a strong protection.

Author Contributions: Writing—original draft, S.S., H.M.-G., I.R. and M.B.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kim, Y.; Daly, R.; Kim, J.; Fallin, C.; Lee, J.H.; Lee, D.; Wilkerson, C.; Lai, K.; Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, 14–18 June 2014; pp. 361–372. [CrossRef]
2. Kwong, A.; Genkin, D.; Gruss, D.; Yarom, Y. RAMBleed: Reading Bits in Memory Without Accessing Them. In Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 18–20 May 2020.
3. Buchanan, E.; Roemer, R.; Shacham, H.; Savage, S. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In Proceedings of the 15th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 27–31 October 2008; pp. 27–38. [CrossRef]

4. Checkoway, S.; Davi, L.; Dmitrienko, A.; Sadeghi, A.R.; Shacham, H.; Winandy, M. Return-oriented Programming Without Returns. In Proceedings of the 17th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 4–8 October 2010; pp. 559–572. [CrossRef]
5. Bletsch, T.; Jiang, X.; Freeh, V.W.; Liang, Z. Jump-oriented Programming: A New Class of Code-reuse Attack. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, Hong Kong, China, 22–24 March 2011; pp. 30–40. [CrossRef]
6. Bosman, E.; Bos, H. Framing Signals—A Return to Portable Shellcode. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, 18–21 May 2014; pp. 243–258. [CrossRef]
7. Marco Gisbert, H.; Ripoli, I. On the Effectiveness of Full-ASLR on 64-bit Linux. In Proceedings of the In-depth Security Conference 2014 (DeepSec), Vienna, Austria, 18–21 November 2014.
8. Marco-Gisbert, H.; Ripoll, I. Return-to-csu: A new method to bypass 64-bit Linux ASLR. In Proceedings of the Black Hat Asia 2018, Singapore, 20–23 March 2018.
9. Cowan, C. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, 26–29 January 1998.
10. Etoh, H. GCC Extension for Protecting Applications from Stack-Smashing Attacks (ProPolice). 2003. Available online: http://www.trl.ibm.com/projects/security/ssp/ (accessed on 1 December 2018) .
11. Wang, Z.; Ding, X.; Pang, C.; Guo, J.; Zhu, J.; Mao, B. To detect stack buffer overflow with polymorphic canaries. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018; pp. 243–254.
12. The PaX Team. Address Space Layout Randomization. 2001. Available online: http://pax.grsecurity.net/docs/aslr.txt (accessed on 1 August 2018).
13. Bhatkar, S.; DuVarney, D.C.; Sekar, R. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA, 4–8 August 2003.
14. Lu, K.; Song, C.; Lee, B.; Chung, S.P.; Kim, T.; Lee, W. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 280–291. [CrossRef]
15. van de Ven, A. New Security Enhancementsin Red Hat Enterprise Linux. 2004. Available online: http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf (accessed on 3 March 2019).
16. Marco-Gisbert, H.; Ripoll, I. On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows. In Proceedings of the 2014 IEEE 13th International Symposium on Network Computing and Applications, Cambridge, MA, USA, 21–23 August 2014; pp. 145–152.
17. Wehbe, T.; Mooney, V.; Keezer, D. Hardware-Based Run-Time Code Integrity in Embedded Devices. *Cryptography* **2018**, 2, 20. [CrossRef]
18. Nanda, S.; Li, W.; Lam, L.; Chiueh, T. Foreign Code Detection on the Windows/X86 Platform. In Proceedings of the 2006 22nd Annual Computer Security Applications Conference (ACSAC'06), Miami Beach, FL, USA, 11–15 December 2006; pp. 279–288. [CrossRef]
19. The MITRE Corporation. CWE Category. 2017. Available online: http://cwe.mitre.org/ (accessed on 6 March 2019).
20. De Clercq, R.; Verbauwhede, I. A survey of Hardware-based Control Flow Integrity (CFI). *arXiv* **2017**, arXiv:1706.07257.
21. Watters, B. Stack-Based Buffer Overflow Attacks: What You Need to Know. 2019. Available online: https://blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/ (accessed on 7 April 2019).
22. Novark, G.; Berger, E.D. DieHarder: Securing the Heap. In Proceedings of the 17th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 4–8 October 2010; pp. 573–584. [CrossRef]
23. Jia, X.; Zhang, C.; Su, P.; Yang, Y.; Huang, H.; Feng, D. Towards Efficient Heap Overflow Discovery. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 989–1006.
24. Wilson, P.R.; Johnstone, M.S.; Neely, M.; Boles, D. Dynamic storage allocation: A survey and critical review. In *Memory Management*; Baler, H.G., Ed.; Springer: Berlin/Heidelberg, Germany, 1995; pp. 1–116.
25. Payer, M.; Gross, T.R. String oriented programming: When ASLR is not enough. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, Rome, Italy, 26 January 2013; p. 2.

26. Kilic, F.; Kittel, T.; Eckert, C. Blind format string attacks. In Proceedings of the International Conference on Security and Privacy in Communication Networks, Beijing, China, 24–26 September 2014; pp. 301–314.
27. Newsham, T. *Format String Attacks*; Guardent, Inc.: Waltham, MA, USA, 2000.
28. Dietz, W.; Li, P.; Regehr, J.; Adve, V. Understanding integer overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.* **2015**, *25*, 2. [CrossRef]
29. Dietz, W.; Li, P.; Regehr, J.; Adve, V. Understanding Integer Overflow in C/C++. In Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, 2–9 June 2012; pp. 760–770.
30. Dowson, M. The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes* **1997**, *22*, 84. [CrossRef]
31. PURE-HACKING. An Introduction to Use After Free Vulnerabilities. 2016. Available online: https://www.purehacking.com/blog/lloyd-simon/an-introduction-to-use-after-free-vulnerabilities (accessed on 1 March 2019).
32. OWASP.ORG. Null Dereference. 2017. Available online: https://www.owasp.org/index.php/Null_Dereference (accessed on 2 April 2019).
33. OWASP. Code Injection. 2013. Available online: https://www.owasp.org/index.php/Code_Injection (accessed on 28 September 2017).
34. Shellblade.net. Performing a Ret2libc Attack. 2018. Available online: https://www.shellblade.net/docs/ret2libc.pdf (accessed on 25 May 2017).
35. Roglia, G.F.; Martignoni, L.; Paleari, R.; Bruschi, D. Surgically Returning to Randomized Lib(C). In Proceedings of the 2009 Annual Computer Security Applications Conference, Honolulu, HI, USA, 7–11 December 2009; pp. 60–69. [CrossRef]
36. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; et al. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018.
37. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre Attacks: Exploiting Speculative Execution. In Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19), San Francisco, CA, USA, 19–23 May 2019.
38. Canella, C.; Bulck, J.V.; Schwarz, M.; Lipp, M.; von Berg, B.; Ortner, P.; Piessens, F.; Evtyushkin, D.; Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv* **2018**, arXiv:1811.05441 .
39. Guan, L.; Lin, J.; Luo, B.; Jing, J.; Wang, J. Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 3–19. [CrossRef]
40. Pappas, V. Defending against Return-Oriented Programming. 2015. Available online: https://www.cs.columbia.edu/~angelos/Papers/theses/vpappas_thesis.pdf (accessed on 21 February 2018).
41. Assurance, N.I. *Hardware Control Flow Integrity CFI for an IT Ecosystem*; NSA: Fort Meade, MD, USA, 2015.
42. Zhang, M.; Sekar, R. Control Flow Integrity for COTS Binaries. In Proceedings of the 22nd USENIX Conference on Security, Washington, DC, USA, 14–16 August 2013; pp. 337–352.
43. Tice, C.; Roeder, T.; Collingbourne, P.; Checkoway, S.; Erlingsson, U.; Lozano, L.; Pike, G. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In Proceedings of the 23rd USENIX Conference on Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 941–955.
44. Niu, B.; Tan, G. Per-Input Control-Flow Integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 914–926. [CrossRef]
45. Carlini, N.; Barresi, A.; Payer, M.; Wagner, D.; Gross, T.R. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In Proceedings of the 24th USENIX Conference on Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 161–176.
46. Mashtizadeh, A.J.; Bittau, A.; Boneh, D.; Mazières, D. CCFI: Cryptographically Enforced Control Flow Integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 941–951. [CrossRef]
47. Mohan, V.; Larsen, P.; Brunthaler, S.; Hamlen, K.W.; Franz, M. Opaque Control Flow Integrity. In Proceedings of the 22nd Annual Network and Distributed System Security Symposium, San Diego, CA, USA, 8–11 February 2015.
48. Muench, M.; Pagani, F.; Yan, S.; Kruegel, C.; Vigna, G.; Balzarotti, D. Taming transactions: Towards hardware-assisted control flow integrity using transactional memory. In Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses, Paris, France, 19–21 September 2016. [CrossRef]

49. Grsecurity. How Does RAP Works. Available online: https://grsecurity.net/rap_faq.php (accessed on 3 February 2018).
50. Davi, L.; Sadeghi, A.R.; Lehmann, D.; Monrose, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection. In Proceedings of the 23rd USENIX Conference on Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 401–416.
51. Abadi, M.; Budiu, M.; Erlingsson, U.; Ligatti, J. Control-flow Integrity. In Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 7–11 November 2005; pp. 340–353. [CrossRef]
52. Abadi, M.; Budiu, M.; Erlingsson, U.; Ligatti, J. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* **2009**, *13*, 4:1–4:40. [CrossRef]
53. Hawkins, B.; Demsky, B.; Taylor, M.B. BlackBox: Lightweight Security Monitoring for COTS Binaries. In Proceedings of the 2016 International Symposium on Code Generation and Optimization, Barcelona, Spain, 12–18 March 2016; pp. 261–272. [CrossRef]
54. Biswas, P.; Federico, A.D.; Carr, S.A.; Rajasekaran, P.; Volckaert, S.; Na, Y.; Franz, M.; Payer, M. Venerable Variadic Vulnerabilities Vanquished. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 186–198.
55. Zhang, J.; Hou, R.; Fan, J.; Liu, K.; Zhang, L.; McKee, S.A. RAGuard: A Hardware Based Mechanism for Backward-Edge Control-Flow Integrity. In Proceedings of the Computing Frontiers Conference, Siena, Italy, 15–17 May 2017; pp. 27–34. [CrossRef]
56. Liljestrand, H.; Nyman, T.; Wang, K.; Perez, C.C.; Ekberg, J.E.; Asokan, N. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 177–194.
57. Burow, N.; Carr, S.A.; Nash, J.; Larsen, P.; Franz, M.; Brunthaler, S.; Payer, M. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* **2017**, *50*, 16:1–16:33. [CrossRef]
58. Hu, H.; Qian, C.; Yagemann, C.; Chung, S.P.H.; Harris, W.R.; Kim, T.; Lee, W. Enforcing Unique Code Target Property for Control-Flow Integrity. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 1470–1486. [CrossRef]
59. Zhang, C.; Wei, T.; Chen, Z.; Duan, L.; Szekeres, L.; McCamant, S.; Song, D.; Zou, W. Practical Control Flow Integrity and Randomization for Binary Executables. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 559–573. [CrossRef]
60. Qiu, P.; Lyu, Y.; Zhang, J.; Wang, D.; Qu, G. Control Flow Integrity Based on Lightweight Encryption Architecture. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *37*, 1358–1369. [CrossRef]
61. Criswell, J.; Dautenhahn, N.; Adve, V. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014; pp. 292–307. [CrossRef]
62. Ge, X.; Talele, N.; Payer, M.; Jaeger, T. Fine-Grained Control-Flow Integrity for Kernel Software. In Proceedings of the IEEE European Symposium on Security and Privacy, Saarbrucken, Germany, 21–24 March 2016; pp. 179–194.
63. Moreira, J.; Rigo, S.; Polychronakis, M.; Kemerlis, V.P. DROP THE ROP fine-grained control-flow integrity for the Linux kernel. In Proceedings of the Black Hat Asia, Singapore, 28–31 March 2017.
64. Evans, I.; Long, F.; Otgonbaatar, U.; Shrobe, H.; Rinard, M.; Okhravi, H.; Sidiroglou-Douskos, S. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 901–913. [CrossRef]
65. Jang, D.; Tatlock, Z.; Lerner, S. *SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks*; Internet Society: San Diego, CA, USA, 2014.
66. Microsoft.com. Control Flow Guard. 2013. Available online: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf (accessed on 29 March 2018).
67. Li, J.; Tong, X.; Zhang, F.; Ma, J. FINE -CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 1535–1550. [CrossRef]
68. Farkhani, R.M.; Jafari, S.; Arshad, S.; Robertson, W.K.; Kirda, E.; Okhravi, H. On the Effectiveness of Type-based Control Flow Integrity. *arXiv* **2018**, arXiv:1810.10649.

69. Introduction to Intel® Memory Protection Extensions. 2013. Available online: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions (accessed on 1 February 2019).
70. Chen, X.; Slowinska, A.; Andriesse, D.; Bos, H.; Giuffrida, C. *StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries*; Internet Society: San Diego, CA, USA, 2015.
71. Ding, R.; Qian, C.; Song, C.; Harris, B.; Kim, T.; Lee, W. Efficient Protection of Path-Sensitive Control Security. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 131–148.
72. Pomonis, M.; Petsios, T.; Keromytis, A.D.; Polychronakis, M.; Kemerlis, V.P. kRX: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In Proceedings of the Proceedings of the Twelfth European Conference on Computer Systems, Belgrade, Serbia, 23–26 April 2017; pp. 420–436.
73. van der Veen, V.; Andriesse, D.; Göktaş, E.; Gras, B.; Sambuc, L.; Slowinska, A.; Bos, H.; Giuffrida, C. Practical Context-Sensitive CFI. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 927–940. [CrossRef]
74. Andrea Biondo, M.C.; Lain, D. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2018, SanDiego, CA, USA, 18–21 February 2018. [CrossRef]
75. Van der Veen, V.; Andriesse, D.; Stamatogiannakis, M.; Chen, X.; Bos, H.; Giuffrdia, C. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 1675–1689. [CrossRef]
76. Zhang, J.; Qi, B.; Qu, G. HCIC: Hardware-assisted Control-flow Integrity Checking. *arXiv* **2018**, arXiv:1801.07397 .
77. Li, J.; Chen, L.; Xu, Q.; Tian, L.; Shi, G.; Chen, K.; Meng, D. Zipper Stack: Shadow Stacks Without Shadow. *arXiv* **2019**, arXiv:1902.00888 .
78. Wang, X.; Huang, F.; Chen, H. DTrace: Fine-grained and efficient data integrity checking with hardware instruction tracing. *Cybersecurity* **2019**, *2*, 1. [CrossRef]
79. Gu, Y.; Zhao, Q.; Zhang, Y.; Lin, Z. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In Proceedings of the CODASPY 2017, Scottsdale, AZ, USA, 22–24 March 2017.
80. Zhang, J.; Chen, W.; Niu, Y. DeepCheck: A Non-intrusive Control-flow Integrity Checking based on Deep Learning. *arXiv* **2019**, arXiv:1905.01858 .
81. Liebchen, C. Advancing Memory-Corruption Attacks and Defenses. Ph.D. Thesis, Technische Universität, Berlin, Germany, 2018.
82. Wang, C. Advanced Code Reuse Attacks against Modern Defences. Ph.D. Thesis, Nanyang Technological University Library, Singapore, 2019.
83. Christoulakis, N.; Christou, G.; Athanasopoulos, E.; Ioannidis, S. HCFI: Hardware-enforced Control-Flow Integrity. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, 9–11 March 2016; pp. 38–49. [CrossRef]
84. Power of Community. Windows 10 Control Flow Guard Internals. 2014. Available online: http://www.powerofcommunity.net/poc2014/mj0011.pdf (accessed on 15 January 2018).
85. Crane, S.J.; Volckaert, S.; Schuster, F.; Liebchen, C.; Larsen, P.; Davi, L.; Sadeghi, A.R.; Holz, T.; De Sutter, B.; Franz, M. It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 243–255.[CrossRef]
86. Zhang, M.; Qiao, R.; Hasabnis, N.; Sekar, R. A Platform for Secure Static Binary Instrumentation. *SIGPLAN Not.* **2014**, *49*, 129–140. [CrossRef]
87. Dang, T.H.; Maniatis, P.; Wagner, D. The Performance Cost of Shadow Stacks and Stack Canaries. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore, 14–17 April 2015; pp. 555–566. [CrossRef]