# Evaluating Modern Defenses Against Control Flow Hijacking

by

Ulziibayar Otgonbaatar

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015



Author .....
Department of Electrical Engineering and Computer Science
August 18, 2015

Certified by .....
Hamed Okhravi
Technical Staff, Cyber Analytics and Decision Systems, MIT Lincoln Laboratory
Thesis Supervisor

Certified by .....
Howard Shrobe
Principal Research Scientist and Director Security, CSAIL
Thesis Supervisor

Accepted by .....
Albert R. Meyer
Chairman, Department Committee on Graduate Theses

**Evaluating Modern Defenses Against Control Flow Hijacking**

by

Ulziibayar Otgonbaatar

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

## Abstract

Memory corruption attacks continue to be a major vector of attack for compromising modern systems. Strong defenses such as complete memory safety for legacy languages (C/C++) incur a large overhead, while weaker and practical defenses such as Code Pointer Integrity (CPI) and Control Flow Integrity (CFI) have their weaknesses. In this thesis, I present attacks that expose the fundamental weaknesses of CPI and CFI.

CPI promises to balance security and performance by focusing memory safety on code pointers thus preventing most control-hijacking attacks while maintaining low overhead. CPI protects access to code pointers by storing them in a safe region that is isolated by hardward enforcement on 0x86-32 architecture and by information-hiding on 0x86-64 and ARM architectures. We show that when CPI relies on information hiding, CPI's safe region can be leaked and thus rendering it ineffective against malicious exploits.

CFI works by assigning tags to indirect branch targets statically and checking them at runtime. Coarse-grained enforcements of CFI that use a small number of tags to improve the performance overhead have been shown to be ineffective. As a result, a number of recent efforts have focused on fine-grained enforcement of CFI as it was originally proposed. In this work, we show that even a fine-grained form of CFI with unlimited number of tags is ineffective in protecting against attacks. We show that many popular code bases such as Apache and Nginx use coding practices that create flexibility in their *intended* control flow graph (CFG) even when a strong static analyzer is used to construct the CFG. These flexibilities allow an attacker to gain control of the execution while strictly adhering to a fine-grained CFI.

Thesis Supervisor: Hamed Okhravi
Title: Technical Staff, Cyber Analytics and Decision Systems, MIT Lincoln Laboratory

Thesis Supervisor: Howard Shrobe
Title: Principal Research Scientist and Director Security, CSAIL

# Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Control Flow Hijacking

Memory corruption bugs in software written in low-level languages like C or C++ are one of the oldest problems in computer security. The lack of memory safety in these languages allows attackers to alter the program's behavior or take full control over it by hijacking its control flow. This problem has existed for more than 30 years and a vast number of potential solutions have been proposed, yet memory corruption attacks continue to pose a serious threat. Real world exploits show that all currently deployed protections can be defeated.

Recent code-injection attacks on iPhones [46, 14], have shown a level of sophistication that is able to bypass widely-deployed prevention mechanisms against code-injection.

Attacks that divert a program's control flow for malicious purposes are generally known as *control-hijacking attacks*. Figure 1-1 outlines the steps involved in control-hijacking attacks, we briefly review some of the defensive mechanisms that protects control hijacking at each individual steps.

A typical control hijacking attack starts by corrupting a pointer to an attacker-supplied malicious data, which we refer to as the *payload*.

Then the attack proceeds by modifying code pointers, which are objects that affect control flow of a program. These two steps could be accomplished by an overflow

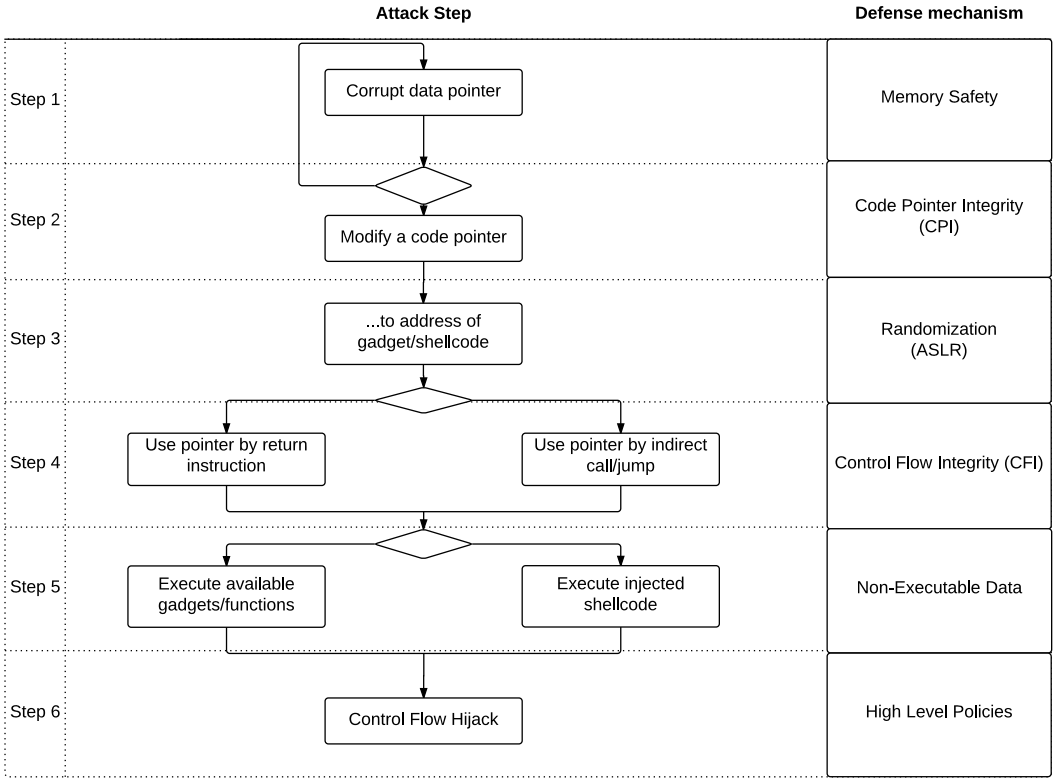


Figure 1-1: Steps in Control-Hijacking attack

exploit. In order to hijack control successfully, the attacker needs to know the correct target value ( i.e. the address of the payload) with which to overwrite the code pointer in step 2. That is represented in a separate step 3 in Figure 1-1. Assuming a code pointer (e.g. a function pointer) has been successfully corrupted in the first three steps, an attacker then proceeds by loading the corrupted pointer into the instruction pointer, which points to next instruction a program executes. This can be achieved by executing an indirect control-flow transfer instruction, like a return instruction, thus diverting the execution from the intended program control-flow. The final step of a control-flow hijack exploit is the execution of the payload, which is often something malicious such as stealing private information, or gaining high privilege on specific targets [33] .

Memory safety enforcement mechanisms [38, 19] ensure no data pointer is cor-

rupted, thus preventing the attack in step 1. The state of the art, Softbound [38] and its extension CETS [39] in fact guarantees complete memory safety at the cost of twice to four times performance slowdown. In practical applications, such a high performance overhead is intolerable.

Code Pointer Integrity (CPI) enforcement mechanism [31] along with Safestack[15] implementation validates the integrity of code pointers only, which incurs significantly lower cost (8.6% overhead) than complete memory protection. However, one of the main weakness of CPI is it's reliance on secrets which are kept in the same space as the process being protected. Information-hiding is shown to be vulnerable to side-channel attacks [52, 55, 58, 63]. Arguably, this problem has contributed to the weaknesses of many other defenses such as Address Space Layout Randomization (ASLR) [59]

ASLR attempts to thwart attacks by introducing entropy to memory addresses. In particular, in order to prevent the attacker from specifying the target location reliably, ASLR randomly arranges the key areas of address space positions. This stops attacks at step 3.

In its ideal form, Control Flow Integrity [4] is a promising enforcement mechanism against attacks that arbitrarily control and hijack a program's behavior in general. The CFI security policy dictates that any execution of a program must follow a path of a Control Flow Graph (CFG) determined ahead of time. If we can perfectly construct CFG for a given program before execution time, any malicious code that attempts to hijack the program's control could be detected as the control flow is not in the intended control flow graph. Hence, CFI prevents the attack at step 4 in Figure 1-1.

Perfectly constructing and maintaining CFG ahead of time firstly requires the source code of the program and secondly incurs very high performance cost due to runtime checks. Even with source code, constructing perfect CFG is impractically hard because there are programming constructs in the code that make analysis hard. These reasons have prevented CFI from being widely adopted as counter measurement against control flow hijacking attacks and much of current research efforts focus on making CFI fast and practical [66, 65, 37, 60].

As for protections at step 5, Not-Executable (NX) policies, such as W $\oplus$ X [42]

or DEP [36], protects the execution of injected payload.

In particular, NX marks memory pages either writable (W) or executable (X), hence the injected payload which generally resides in the data section of memory is not executable. Due to its low performance overhead and simplicity, NX policies remain to be one of the mostly deployed security feature in modern operating systems [42, 36].

However, even when Not-Executable policy is in place, the attacks that reuse the existing code in memory still remain possible such as return-oriented programming [13, 18], jump-oriented programming [10], and return-into-libc attacks [61].

### 1.1.1 Return-Oriented Programming

Return-Oriented programming (ROP) [13, 18], by reusing small gadgets, which are comparatively small sequences of instructions ending in return instruction, is able to bypass NX.

Generally, ROP is shown to be Turing-complete, meaning that an attacker can do arbitrary computation on target computer given the right set of gadgets.

It is broadly applicable on a number of architectures: Intel x86 [54], RISC [11], ARM [30].

Various mechanisms, kBouncer [43], ROPecker [16], ROPguard [21], that check the return addresses found in function epilogues can protect from ROP attacks. These techniques that leverage the Last Branch Recording (LBR) feature of modern hardware architectures to check for suspicious control flow transfers. These defenses are particularly interesting because they can be deployed on existing hardware, have nearly zero performance overhead, and do not require binary rewriting. However, because the number of states saved in LBR is finite, the security guarantees of these techniques is shown to be broken [13] [18].

In fact, attacks similar to ROP that leverages indirect jump instructions without needing return instruction [10], known as Jump Oriented Programming, have shown that the attack space is much broader than previously imagined.

## 1.2 Defenses Against Control Flow Hijacking

A variety of defensive mechanisms have been proposed to mitigate control-flow hijacking attacks. As previously mentioned, complete memory safety, code pointer integrity, and control flow integrity are promising defenses in theory. The practicality of these defenses relies on how a particular implementation balances security with the performance overhead.

### 1.2.1 Complete Memory Safety

Complete memory safety can defend against all control hijacking attacks by protecting all pointers.

As shown in Figure 1-1, the first step in control hijacking corrupts a pointer by first making it invalid and deferences the pointer. A pointer can become invalid by going out of the bounds of its pointed object or when the object gets deallocated. A pointer pointing to a deleted object is called a dangling pointer. Dereferencing an out-of-bounds pointer causes a so called spatial error, while dereferencing a dangling pointer causes a temporal error.

Memory safety methods [40, 29, 38, 5] that prevent spatial errors provides spatial memory safety, whereas the methods that prevents temporal errors provide temporal memory safety [53, 41, 39, 6].

Softbound with the CETS extensions [38, 39] enforces complete spatial and temporal pointer safety albeit at a significant cost (up to 4x slowdown, 116% on average) on the SPEC CPU benchmark [25].

On the other hand, experience has shown that low overhead mechanisms that trade off security guarantees for performance. For example, approximate [53] or partial [7] memory safety techniques eventually get bypassed [9, 56, 22, 13, 18].

Fortunately, hardware support can make complete memory safety practical. For instance, Intel memory protection extensions (MPX) [28] can facilitate better enforcement of memory safety checks. Secondly, the fat-pointer scheme shows that hardware-based approaches can enforce spatial memory safety at very low overhead

[32]. Tagged architectures and capability-based systems can also provide a possible direction for mitigating such attacks [62].

### 1.2.2 Code Pointer Integrity (CPI)

Unlike complete memory safety, CPI relies on protecting only the integrity of sensitive pointers (e.g. function pointer) to thwart control hijacking. Since sensitive pointers make up a subset of all pointers, CPI promises to provide strong security at a very reasonable performance cost.

CPI first over-approximately identifies all sensitive pointers via static analysis of the code. Then, it stores metata for checking validity of code pointers in a designated "safe region" of memory. The metadata includes the value of the pointer and its lower and upper bounds. Additionally, CPI adds instrumentation that propagates metadata when pointer operations occur.

CPI relies on secret region which is kept in the same space as the process being protected, to be isolated. It assumes it has to be leak proof and cannot be disclosed. However, we show that an attacker can disclose the location of the safe region using a timing side-channel attack. Once the location of a code pointer in the safe region is known, the metadata of the pointer is modified to allow the execution of a ROP chain.

We describe CPI in detail, offer evaluation of the design and implementation, and provide and attack that exposes the weaknesses of CPI in Chapter 2.

### 1.2.3 Control Flow Integrity (CFI)

Proposed originally by Abadi et al.[4], CFI detects control-hijacking attacks by enforcing a Control Flow Graph, which is a representation of all paths a program may take during its execution. It is a promising attempt to stop control-flow hijacking at step 4, thus protecting against ROP.

Any CFI based defense relies on the level of coverage of the CFG. If the CFG is perfectly constructed, then we can ensure only valid the control flows intended by

the program. However, perfectly constructing and maintaining CFG ahead of time requires the source code of the program, its hard and inaccurate, and incurs very high performance cost. These practical concerns demanded that we use a weaker version of CFI, which results in weaker security guarantees.

There are numerous practical CFI mechanisms, CCFIR [65], binCFI [66], and FCFI [60] and many more [37, 35].

Recent studies [18, 22] illustrate the fundamental weaknesses of these existing practical CFI techniques.

We describe CFI in detail, give an overview of attacks against CFI, and give a detailed description of how to attack fine grained CFI in Chapter 3.

# Chapter 2

# Code Pointer Integrity

## 2.1 Overview of Code Pointer Integrity

Code Pointer Integrity (CPI) policies aim to prevent the control hijacking attacks by guaranteeing the integrity of all sensitive pointers (e.g. function pointers) a program. Unlike the complete memory safety techniques which guarantees integrity of all pointers, CPI attempts to guarantee the integrity of only the sensitive pointers which is a subset of all pointers. Thus, if implemented right, it may be possible that CPI prevent all control hijacking attacks with a lower performance cost than Softbound+CETS [38, 39].

Introduced in 2014, Levee [31] is the first practical implementation of CPI. Given that Levee is the only implementation of CPI known to date, we use terms Levee and CPI interchangeably from here on.

In short, CPI works by

- identifying all the sensitive pointers (i.e. code pointers and pointers that may point to sensitive pointers). Note that this definition of sensitive pointers is recursive.
- instrumenting the program to protect all sensitive pointers in a "safe region" in memory, and
- isolating the safe region by preventing non-protected access.

The authors of Levee also present a weaker but more efficient version called Code Pointer Separation (CPS). CPS enforces safety for code pointers only, but not for pointers to code pointers.

Because CPI offers the best security guarantees, we do not discuss CPS in the rest of the chapter, instead focus on evaluating CPI.

This section describes the stages of CPI and states the summary of assumptions made by the authors during each stage.

### 2.1.1 Identifying Sensitive Pointers

CPI performs static analysis on the program to detect the set of sensitive pointers. Sensitive pointer types include pointers to functions, pointers to sensitive types, pointers to memory objects (i.e. `struct`-s or arrays) with members of sensitive types, or universal pointers (e.g. pointers to `void` or `char`). Defined this way, the notion of sensitivity of pointers is dynamic in that at runtime, a pointer may point to a benign integer value (non-sensitive) and may point to a function pointer (sensitive) at some other part of the execution.

This type-based static analysis provides over-approximate set of pointers, (i.e. it may include universal pointers that is never sensitive during runtime). The authors assume that over-approximate set of pointers does not affect the security guarantees of CPI.

After identifying the sensitive pointers, the analysis proceeds to find all the program instructions that operate on these pointers. These instructions are modified to create and propagate the metadata associated with the sensitive pointers in order to enforce integrity of the sensitive pointers. The metadata includes the value of the pointer and its lower and upper thresholds.

### 2.1.2 Instrumentations For CPI Enforcement Mechanism

The main goal of the instrumentation is to change the program to ensure the integrity the sensitive pointers. In simple terms, it is achieved by checking metadata of the

sensitive pointers on pointer dereference. The metadata for a sensitive pointer is created in designated safe region of memory and propagated at program run time.

Secondly, the instrumentation changes the code to ensure that only CPI intrinsic instructions can manipulate the safe region and that no pointer can directly reference the safe region. This is to prevent any code pointer from disclosing the location of the safe region.

### 2.1.3 Isolating Safe Region of Memory

The mechanism for isolating the safe region of memory from non-protected accesses is dependent on the computer architecture CPI is deployed on.

On x86-32 architecture, CPI uses hardware segment registers which are used to access specific regions of memory. By configuring all but one dedicated segment register inaccessible to the safe region, CPI guarantees safe region is accessed through only one segment register.

On architectures that do not support segmentation protection, such as x86-64 and ARM, CPI relies on the size and sparsity of the safe region ($2^{42}$ bytes), randomization, to isolate the contents of the safe region from disclosure. In particular, the safe region is placed at a randomly chosen address in specific area of memory.

In the design, the authors of CPI makes three fundamental assumptions to isolate the safe region. Firstly, they assume that the large parts of memory cannot leak. Secondly, they assume that the safe region is isolated because no address pointing to the safe region is ever stored in the rest of the memory. Thirdly, it's assumed that making guesses as to where the safe region is impractical because the failed guessing attempts would crash the program and is easily detectable.

## 2.2 Evaluation of Code Pointer Integrity

After thorough evaluation of CPI, we discovered numerous implementation and design flaws that can facilitate attack against CPI. This chapter focuses on the evaluating the design and implementation of CPI.

We found that the main weakness of CPI is it's reliance on secrets which are kept in the same space as the process being protected. Arguably, this problem has contributed to the weaknesses of many other defenses such as ASLR [55].

### 2.2.1 Weaknesses of CPI Design Assumptions

The assumptions made by the CPI authors about isolating the safe region are weak at best. In fact, we show that these assumptions are erroneous and can lead to control hijacking attack as we show in our next chapter.

#### Memory Disclosure

In CPI, it is implicitly assumed that large parts of memory cannot leak. Direct memory disclosure techniques may have some limitations. For example, they may be terminated by zero bytes or may be limited to areas adjacent to a buffer [58]. However, indirect leaks using dangling data pointers and timing or fault analysis attacks do not have these limitations and they can leak large parts of memory.

#### Memory Isolation

The assumption that the safe region cannot leak because there is no pointer to it is incorrect. As we show in our attacks in rest of the chapter, random searching of the mmap region can be used to leak the safe region without requiring an explicit pointer into that region.

#### Detecting Crashes

It also is assumed that leaking large parts of memory requires causing numerous crashes which can be detected using other mechanisms. This in fact is not correct. Although attacks such as Blind ROP [9] and brute force [55] do cause numerous crashes, it is also possible on current CPI implementations to avoid such crashes using side- channel attacks. The main reason for this is that in practice large number of pages are allocated and in fact, the entropy in the start address of a region is much

smaller than its size. This allows an attacker to land correctly inside allocated space which makes the attack non-crashing. In fact, CPI's implementation exacerbates this problem by allocating a very large mmap region to contain the safe region.

### 2.2.2 Weaknesses of CPI Implementation

The published implementation (simpletable) of CPI uses a fixed address for the table for all supported architectures, providing no protection in its default configuration. We assume this is due to the fact that the version of CPI we evaluated is still in "early preview". We kept this in mind throughout our evaluation, and focused primarily on fundamental problems with the use of information hiding in CPI. Having said that, we found that as currently implemented there was almost no focus on protecting the location of the safe region.

The two alternate implementations left in the source, hashtable and lookuptable, use mmap directly without a fixed address, which is an improvement but is of course relying on mmap for randomization. This provides no protection against an ASLR disclosure, which is within the threat model of the CPI paper. We further note that the safe stack implementation also allocates pages using mmap without a fixed address, thus making it similarly vulnerable to an ASLR disclosure. This vulnerability makes the safe stack weaker than the protection offered by a stack canary, as any ASLR disclosure will allow the safe stack location to be determined, whereas a stack canary needs a more targeted disclosure (although it can be bypassed in other ways).

In the default implementation (simpletable), the location of the table is stored in a static variable (`__llvm__cpi_table`) which is not zeroed after its value is moved into the segment register. Thus, it is trivially available to an attacker by reading a fixed offset in the data segment. In the two alternate implementations, the location of the table is not zeroed because it is never protected by storage in the segment registers at all. Instead it is stored as a local variable. Once again, this is trivially vulnerable to an attack who can read process memory, and once disclosed will immediately compromise the CPI guarantees. Note that zeroing memory or registers is often difficult to perform correctly in C in the presence of optimizing compilers [45].

We note that CPI's performance numbers rely on support for superpages (referred to as huge pages on Linux). In the configurations used for performance evaluation, ASLR was not enabled (FreeBSD does not currently have support for ASLR, and as of Linux kernel 3.13, the base for huge table allocations in mmap is not randomized, although a patch adding support has since been added). We note this to point out a difference between CPI performance tests and a real world environment, although we have no immediate reason to suspect a large performance penalty from ASLR being enabled.

It is unclear exactly how the published CPI implementation intends to use the segment registers on 32-bit systems. The simpletable implementation, which uses the `%gs` register, warns that it is not supported on x86, although it compiles. We note that using the segment registers may conflict in Linux with thread-local storage (TLS), which uses the `%gs` register on x86-32 and the `%fs` register on x86-64 [20]. As mentioned, the default implementation, simpletable, does not support 32-bit systems, and the other implementations do not use the segment registers at all, a flaw noted previously, so currently this flaw is not easily exposed. A quick search of 32-bit libc, however, found almost 3000 instructions using the `%gs` register. Presumably this could be fixed by using the `%fs` register on 32-bit systems; however, we note that this may cause compatibility issues with applications expecting the `%fs` register to be free, such as Wine (which is explicitly noted in the Linux kernel source) [3].

Additionally, the usage of the `%gs` and `%fs` segment registers might cause conflicts if CPI were applied to protect kernel-mode code, a stated goal of the CPI approach. The Linux and Windows kernels both have special usages for these registers.

## 2.3 Attacking Code Pointer Integrity

This chapter shows how an attacker can defeat CPI, on x86-64 architectures, assuming only control of the stack. Specifically, we show how to reveal the location of the safe region using a data-pointer overwrite without causing any crashes, which was assumed to be impossible by the CPI authors. We present experimental results that

demonstrate the ability to leak the safe region using a timing side-channel attack. Also, we present how to attack CPI protected version of the popular Nginx [48] web server without causing any crash. Our analysis also shows that an attack can be completed without any crashes in about 98 hours for the most performant and complete implementation of CPI. If we tolerate a small number of crashes, our analysis shows that in Ubuntu Linux with ASLR, it takes 6 seconds to bypass CPI with 13 crashes.

The details on the timing side-channel attack, how to leak and locate the safe region, both with or without crashes, and how to launch a payload, is presented in the rest of the chapter.

### 2.3.1 Launch Timing Side-channel

A data-pointer overwrite vulnerability is used to control a data pointer that is subsequently used to affect control flow (e.g., number of loop iterations) is used to reveal the contents of the pointer under control (i.e., byte values). The data pointer can be overwritten to point to a return address on the stack, revealing where code is located, or a location in the code segment, revealing what code is located there.

Consider the code sequence below. If `ptr` can be overwritten by an attacker to point to a location in memory, the execution time of the `while` loop will be correlated with the byte value to which `ptr` is pointing. For example, if `ptr` is stored on the stack, a simple buffer overflow can corrupt its value to point to an arbitrary location in memory. This delay is small (on the order of nanoseconds); however, by making numerous queries over the network and keeping the fastest samples (cumulative delay analysis), an attacker can get an accurate estimate of the byte values [52, 17].

```
i = 0;
while (i < ptr->value) i++;
```

Figure 2-1: Data-pointer based side-channel example

In particular, we patch Nginx to introduce a stack buffer overflow vulnerability

allowing the user to gain control of a parameter used as the upper loop bound in the Nginx logging system. The vulnerability enables an attacker to place arbitrary values on the stack. Using the vulnerability, we modify a data pointer in the Nginx logging module (`nginx_http_parse.c`) to point to a carefully chosen address.

```
for (i = 0; i < headers->nelts; i++)
```

Figure 2-2: Nginx data-pointer used for attack

The data pointer vulnerability enables control over the number of iterations executed in the loop. Using this data pointer vulnerability, we perform Round Trip Time (RTT) based analysis that enables us to reveal contents of memory.

### 2.3.2 Data Collection

We measure round-trip response times to our attack application in order to collect the timing samples. We create a mapping between the smallest cumulative delay slope and byte 0, and the largest slope and byte 255. We use these two mappings to interpolate cumulative delay slopes for all possible byte values (0-255). This enables us to identify a byte in arbitrary memory location with high accuracy, which in turn enables us to read the contents of memory.

In particular, assuming the highest RTT observed is when the loop controller was byte 255 and the lowest RTT is when loop controller was byte 0, we can establish linear interpolation to estimate a byte in memory.

Equation 2.1 shows the exact interpolation. In the equation, $c_i$ represents the delay sample RTT for a nonzero byte value, and $s$ represents the number of samples taken. Note that the cumulative delay is adjusted in reference to a baseline round trip time, so that we are only interpolating the time taken by all the loop iterations to get the byte. In our setup, we collected 10000 timing samples to establish baseline RTT.

$$byte = c \sum_{i=1}^{s} c_i \tag{2.1}$$

We then use the set of delay samples collected for byte 255 to calculate the constant $c$ as shown below.

$$c = \frac{255}{\sum_{i=1}^{s} c_i}$$

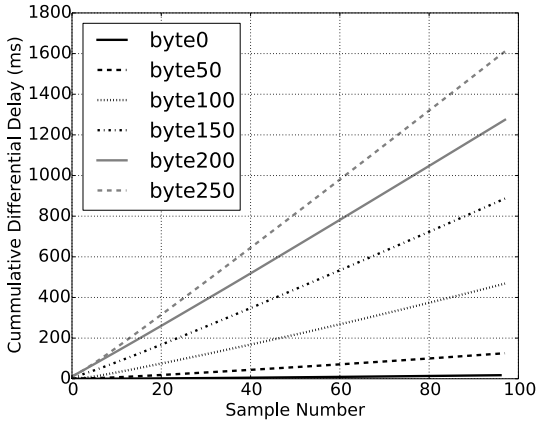


Figure 2-3: Timing Measurement for Nginx 1.6.2 over Wired LAN

The Figure 2-3 shows the slopes of the cumulative delay for different bytes as measured in our experimental setup. Using the equation 2.1, we can estimate a given byte in memory, with high accuracy. This enables us to reveal the content of memory byte-by-byte.

This particular timing side-channel attack shows violates the assumption made in CPI about memory disclosure.

### 2.3.3 Locate Safe Region Without Crashes

Using information about the possible location of the safe region with respect to the randomized location of mmap, we launch a search that starts at a reliably mapped location within the safe region and traverse the safe region until we discover a sequence of bytes that indicates the location of a known library (e.g., the base of libc). Under

the current implementation of CPI, discovering the base of libc allows us to trivially compute the base address of the safe region. Up to this point, the attack is completely transparent to CPI and may not cause any crash or detectable side effect.

The attack procedure is as follows.

1. Starting at always allocated address: Redirect a data pointer which is always allocated. (see Fig. 2-4).
2. Scan go back in memory by the size of libc.
3. Scan some bytes. If the bytes are all zero, goto step 2. Else, scan more bytes to decide where we are in libc by unique signature.
4. Launch payload once in libc.

Below, we describe each of the steps in locating the safe region in detail.

**Starting at Always Allocated Address**

For our attack without crashes, it is important that the addresses we disclose are allocated. Figure 2-4 illustrates the memory layout of a CPI-protected application on the x86-64 architecture. The stack is located at the top of the virtual address space and grows downwards (towards lower memory addresses) and it is followed by the stack gap. Following the stack gap is the base of the mmap region ($mmap\_base$), where shared libraries (e.g., libc) and other regions created by the mmap() system call reside. In systems protected by ASLR, the location of $mmap\_base$ is randomly selected to be between $max\_mmap\_base$ (located immediately after the stack gap) and $min\_mmap\_base$. $min\_mmap\_base$ is computed as:

$$min\_mmap\_base = max\_mmap\_base - aslr\_entropy * page\_size$$

where $aslr\_entropy$ is $2^{28}$ in 64-bit systems, and the $page\_size$ is specified as an operating system parameter (typically 4KB). The safe region is allocated directly

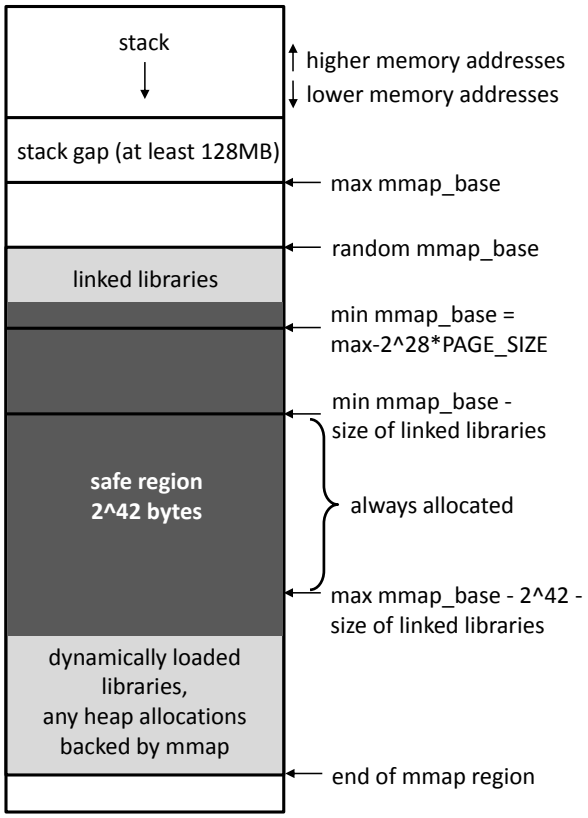


Figure 2-4: Safe Region Memory Layout.

after any linked libraries are loaded on $mmap\_base$ and is $2^{42}$ bytes. Immediately after the safe region lies the region in memory where any dynamically loaded libraries and any mmap-based heap allocations are made.

Hence, the address at $min\_mmap\_base$ is always allocated in CPI, when safe region is randomly allocated.

### Scanning Memory by Size of LIBC

The space of possible locations to search may require $aslr\_entropy * page\_size$ scans in the worst case. As the base address of mmap is page aligned, one obvious optimization is to scan addresses that are multiples of $page\_size$, thus greatly reducing the number of addresses that need to be scanned to:

$$(aslr\_entropy * page\_size)/page\_size$$

In fact, this attack can be made even more efficient. In the x86-64 architecture, CPI protects the safe region by allocating a large region ($2^{42}$ bytes) that is very sparsely populated with pointer metadata. As a result, the vast majority of bytes inside the safe region are zero bytes. This enables us to determine with high probability whether we are inside the safe region or a linked library by sampling bytes for zero/nonzero values (i.e., without requiring accurate byte estimation). Since we start in the safe region and libc is allocated before the safe region, if we go back in memory by the size of libc, we can avoid crashing the application. This is because any location inside the safe region has at least the size of libc allocated memory on top of it.

Hence, the number of memory pages we scan is as follows.

$$(aslr\_entropy * page\_size)/libc\_size \tag{2.2}$$

**Unique Signature in LIBC**

A key component of our attack is the ability to quickly determine whether a given page lies inside the safe region or inside the linked libraries by sampling the page for zero bytes. Even if we hit a nonzero address inside the safe region, which will trigger the search for a known signature within libc, the nearby bytes we scan will not yield a valid libc signature and we can identify the false positive. In our tests, every byte read from the high address space of the safe region yielded zero. In other words, we observed no false positives.

One problematic scenario occurs if we sample zero bytes values while inside libc. In this case, if we mistakenly interpret this address as part of the safe region, we will skip over libc and the attack will fail. We can mitigate this probability by choosing the byte offset per page we scan intelligently. Because we know the memory layout of libc in advance, we can identify page offsets that have a large proportion of nonzero bytes, so if we choose a random page of libc and read the byte at that offset, we will likely read a nonzero value.

In our experiments, page offset 4048 yielded the highest proportion of non-zero

values, with 414 out of the 443 pages of libc having a nonzero byte at that offset. This would give our strategy an error rate of $1 - 414/443 = 6.5\%$. We note that we can reduce this number to 0 by scanning two bytes per page instead at offsets of our choice. In our experiments, if we scan the bytes at offsets 1272 and 1672 in any page of libc, one of these values is guaranteed to be nonzero. This reduces our false positive rate at the cost of a factor of 2 in speed. In our experiments, we found that scanning 5 extra bytes in addition to the two signature bytes can yield 100% accuracy using 30 samples per byte and considering the error in byte estimation.

This non-crashing scan strategy is depicted on the left side of Fig. 2-5.

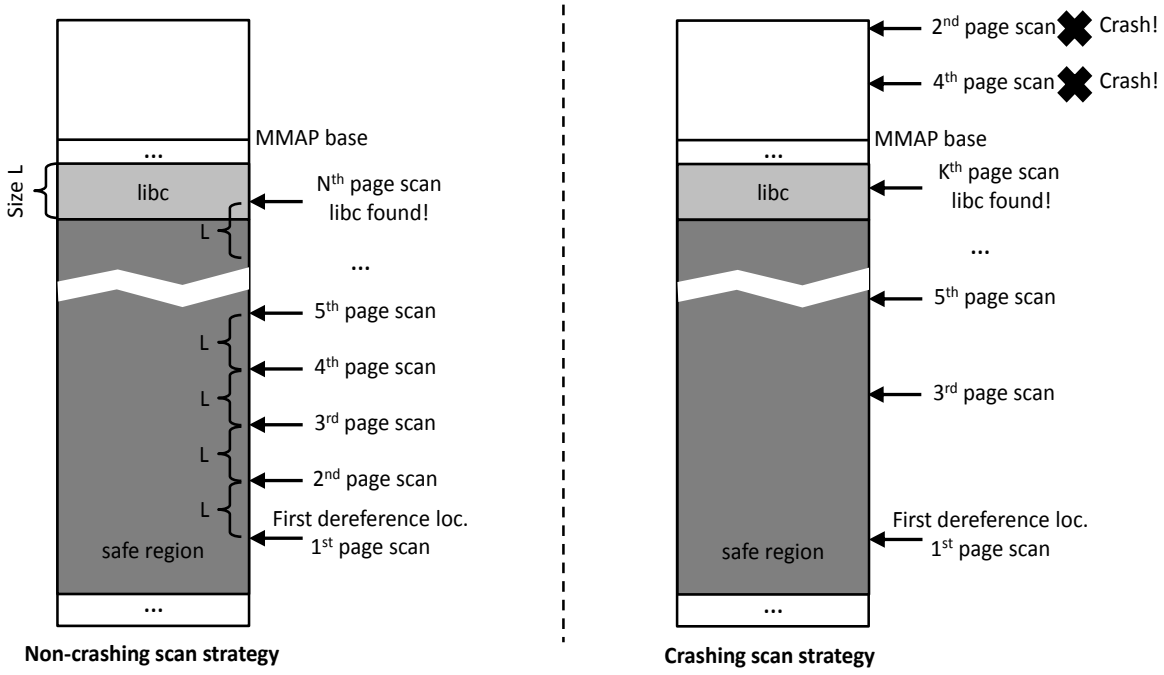


Figure 2-5: Non-Crashing and Crashing Scan Strategies.

### Number of Scans

The number of scans in the non-crashing scan is found in Equation 2.2. In our experiments, *libc_size* is approximately $2^{21}$. In other words, the estimated number of memory page scans is: $2^{28} * 2^{12}/2^{21} = 2^{19}$.

For each page we scan, we would like to sample as few bytes, as few times as possible.

Table 2.1 summarizes the number of false positives, i.e. the number of pages we estimate as nonzero, which are in fact 0. The number of data samples and estimation samples, and their respective fastest percentile used for calculation all affect the accuracy. Scanning 5 extra bytes (in addition to the two signature bytes for a page) and sampling 30 times per bytes yields an accuracy of 100% in our setup.

As a result, we scan $30 * 7 = 210$ bytes per size of libc to decide whether we are in libc or the safe region. In total, to locate libc, the attack requires $(2 + 5) * 2^{19} * 30 = 7 * 2^{19} * 30 = 110,100,480$ samples on average, which takes about 97 hours with our attack setup.

| **# Data samples (%-tile used)** | **# Estimation samples (%-tile used)** | **False positive ratio** |
|---|---|---|
| 1,000 (10%) | 1,000 (10%) | 0% |
| 10,000 (1%) | 1,000 (10%) | 0% |
| 1,000 (10%) | 100 (10%) | 0% |
| 10,000 (1%) | 100 (10%) | 0% |
| 1,000 (10%) | 50 (20%) | 0% |
| 10,000 (1%) | 50 (20%) | 3% |
| 1,000 (10%) | 30 (33%) | 2% |
| 10,000 (1%) | 30 (33%) | 0% |
| 1,000 (10%) | 20 (50%) | 5% |
| 10,000 (1%) | 20 (50%) | 13% |
| 1,000 (10%) | 10 (100%) | 91% |
| 10,000 (1%) | 10 (100%) | 92% |
| 1,000 (10%) | 5 (100%) | 68% |
| 10,000 (1%) | 5 (100%) | 86% |
| 1,000 (10%) | 1 (100%) | 54% |
| 10,000 (1%) | 1 (100%) | 52% |

Table 2.1: Error ratio in estimation of 100 zero pages using 7 bytes

Despite the high accuracy, we have to account for errors in estimation. For this, we have developed a fuzzy $n-$gram matching algorithm that, given a sequence of noisy bytes, tells us the libc offset at which those bytes are located by comparing the estimated bytes with a local copy of libc. In determining zero and nonzero pages, we only collect 30 samples per byte as we do not need very accurate measurements. After

landing in a nonzero page in libc, we do need more accurate measurements to identify our likely location. Our measurements show that $10,000$ samples are necessary to estimate each byte to within 20. We also determine that reading 70 bytes starting at page offset 3333 reliably is enough for the fuzzy $n-$gram matching algorithm to determine where exactly we are in libc. This offset was computed by looking at all contiguous byte sequences for every page of libc and choosing the one which required the fewest bytes to guarantee a unique match. This orientation inside libc incurs additional $70 * 10,000 = 700,000$ requests, which adds another hour to the total time of the attack for a total of 98 hours.

### 2.3.4 Locate Safe Region With Crashes

We can further reduce the number of memory scans if we are willing to tolerate crashes due to dereferencing an address not mapped to a readable page. Because the pages above $mmap\_base$ are not mapped, dereferencing an address above $mmap\_base$ may crash the application. If the application restarts after a crash without re-randomizing its address space, then we can use this information to perform a search with the goal of finding an address $x$ such that $x$ can be dereferenced safely but $x+libc\_size$ causes a crash. This implies that $x$ lies inside the linked library region, thus if we subtract the size of all linked libraries from $x$, we will obtain an address in the safe region that is near libc and can reduce to the case above. Note that it is not guaranteed that $x$ is located at the top of the linked library region: within this region there are pages which are not allocated and there are also pages which do not have read permissions which would cause crashes if dereferenced.

To find such an address $x$, the binary search proceeds as follows: if we crash, our guessed address was too high, otherwise our guess was too low. On right hand side of Figure 2-5, one can see the scanning strategy with crashes. Put another way, we maintain the invariant that the high address in our range will cause a crash while the lower address is safe, and we terminate when the difference reaches the threshold of $libc\_size$. This approach would only require at most $\log_2 2^{19} = 19$ reads and will crash at most 19 times (9.5 times on average).

### 2.3.5 Launching Payload

After finding the libc and hence the safe region, we use the same data pointer overwrite to change the `read_handler` entry of the safe region. We then modify the base and bound of the code pointer to hold the location of the system call (`sysenter`). Since we can control what system call `sysenter` invokes by setting the proper values in the registers, finding `sysenter` allows us to implement a variety of practical payloads. After this, the attack can proceed simply by redirecting the code pointer to the start of a ROP chain that uses the system call. CPI does not prevent the redirection because its entry for the code pointer is already maliciously modified to accept the ROP chain.

# Chapter 3

# Control Flow Integrity

## 3.1 Coarse and Fine Grained Control Flow Integrity

CFI thwarts control-hijacking attacks by ensuring that the control flow remains within the control flow graph intended by the programmer. Every instruction that is the target of a legitimate control-flow transfer is assigned a unique ID, and checks are inserted before control-flow instructions to ensure that only valid targets are allowed. There are two types of control-flow transfers: direct and indirect. Direct transfers have a fixed target and they do not require any enforcement checks. However, indirect transfers, like function calls and returns, and indirect jumps, take a dynamic target address as argument. As the target address could be controlled by an attacker due to a vulnerability, CFI checks to ensure that its ID matches the list of known and allowable target IDs of the instruction. Every control-flow is permitted as long as the CFG allows it.

Consequently, the quality of protection from CFI is dependent on the precision and coverage of the CFG that is generated. Since perfect CFG generation is an intractable in many setups [47], recent CFI solutions have used relaxed rules when enforcing CFI. In particular, the approach the original CFI [4] proposal as well as works like CCFIR [65] and binCFI[66], take is to instrument binary code with a handful of ID-s associated with the type of control flow transfer. We refer to these relaxed and provably weaker [22] form of CFI as *coarse-grained CFI*.

Original CFI and binCFI suggest an implementation using only two ID-s: one for function returns and jumps, and another one for function calls. CCFIR supports three IDs–one for function calls and indirect jumps, one for return addresses in normal functions, and a specialized one for return addresses in a set of security-sensitive functions. Among the three CFI tools, binCFI requires the least information about the binary being protected, and CCFIR has the strictest rules.

When use a unique ID for each control flow transfer, thus resulting in an unlimited number of tags (or ID-s), we refer to it a *fine-grained CFI*. A number of recent fine-grained CFI techniques have been proposed in the literature. Forward-edge CFI [60] enforces a fine-grained CFI on forward-edge control transfers (i.e. indirect calls, but not returns). Cryptographically enforced CFI [35] enforces another form of fine-grained CFI by adding message authentication code (MAC) to control flow elements which prevents the usage of unintended control transfers in the CFG. Opaque CFI (OCFI) [37] enforces a fine-grained CFI by transforming the problem of branch target check to bounds checking (possible base and bound of allowed control transfers). Moreover, it prevents attacks on unintended CFG edges by applying code randomization. The authors of OCFI mention that it achieves resilience against information leakage (a.k.a. memory disclosure) attacks [58, 52] because the attacker can only learn about *intended* edges in such attacks, and not the *unintended* ones which were used in previous attacks against coarse-grained CFI [22]. The attack we describe in section 3.4 shows that just the intended edges are enough for a successful attack that results in remote code execution.

## 3.2 Attacks on Control Flow Integrity

Practical CFI restricts control-flow transfers based on a finite, static CFG. As a result, it cannot guarantee that a function call returns to the call site responsible for the most recent invocation of the function. In addition, Gotkas et al. [22] and Lucas et al. [18] shows that weak CFI allows broader sets of control transfer that the incomplete CFG does not capture, therefore, it is not effective. Using these sets of control transfer

instructions, they are able to mount ROP attack.

Control Flow Bending (CFB) [12] also demonstrates attacks against fine-grained CFI. To perform their proof-of-concept attacks, Control Flow Bending introduces the notion of printf-oriented programming, a form of ACICS gadgets, that can be used to perform Turing-complete computation. CFB assumes a fully-precise CFG, which we show is undecidable. CFB relies on manual analysis for attack construction and is only able to achieve remote code execution in one of their six benchmarks. Moreover, printf-oriented programming is only applicable to older versions glibc. In the newer versions, the `%n` protection prevents the printf-oriented programming attack

In contrast, Control Jujutsu in Section 3.4 introduces a framework (policies and tools) that enable automatic attack construction. CFB and Control Jujutsu demonstrate attacks against fine-grained CFI are possible in theory and in practice.

Counterfeit Object Oriented-Programming (COOP) [51] is another recent attack on modern CFI defenses. COOP focuses exclusively on C++ by showing that protecting v-table pointers in large C++ programs is insufficient. Their work focuses on showing certain design patterns that are common in sufficiently large or complex applications and are not accounted for in the design of CFI defenses. It may be possible to extend the COOP approach to C programs, but we leave this exploration to future work.

## 3.3 Building Control Flow Graphs

As previously mentioned, the security of fine-grained CFI techniques is contingent on the ability to construct CFGs that accurately capture the intended control transfers permitted by the application.

For C/C++ applications, even with access to source code, this assumption is tenuous at best. In theory, the construction of an accurate CFG requires the use of a precise (sound and complete) pointer analysis. Unfortunately, sound and complete points-to analysis is undecidable [47]. In practice, pointer analysis can be made practical by either adopting unsound techniques or reducing precision (incomplete).

Unsound techniques may report fewer connections (tags), which can result in false positives when used in CFI. Given that false positives can interfere with the core program functionality, researchers have focused on building sound but *incomplete* pointer analysis algorithms [8, 34, 64, 50, 44, 24, 23, 27, 57] that conservatively report more connections. For example, two pointers *may* alias and an indirect call site *may* call a function. The hope is that such imprecision could be controlled and that the analysis could be accurate enough so that the generated CFG still does not contain malicious connections.

Another important design decision for pointer analysis algorithms is scalability [26]. Standard pointer analysis algorithms for C programs have three important knobs that control the trade-offs between accuracy and scalability: context-sensitivity, field sensitivity, and flow sensitivity.

### 3.3.1 Scalable Static Pointer Analysis

**Context Sensitivity:** A context-sensitive analysis [64, 34, 57] is able to distinguish between different invocations of a function at different call sites. It tracks local variables, arguments, and return values of different function invocations, at different call sites separately. A context-insensitive analysis, in contrast, does not distinguish between different invocations of a function, i.e., analysis results for local variables, arguments, and the return values from different invocations of the function are merged together.

Unfortunately, context-sensitive pointer analysis is expensive for large real-world applications. Full context-sensitive analysis is also undecidable for programs that contain recursions [49]. Standard clone-based context-sensitive pointer analysis [64] duplicates each function in a program multiple times to distinguish different invocations of the function. This unfortunately will increase the size of the analyzed program exponentially.

**Field Sensitivity:** A field-sensitive analysis [44, 34] is able to distinguish different fields of a struct in C programs, while a field-insensitive analysis treats the whole

struct as a single abstract variable. Modifications to different fields are transformed into weak updates to the same abstract variable, where the analysis conservatively assumes that each of the modifications *may* change the value of the abstract variable.

Field-sensitive pointer analysis is hard for C programs due to the lack of type-safety. Pointer casts are ubiquitous, and unavoidable for low-level operations such as `memcpy()`. Field-sensitive analysis algorithms [44, 34] typically have a set of hand-coded rules to handle common code patterns of pointer casts. When such rules fail for a cast of a struct pointer, the analysis has to conservatively merge all fields associated with the struct pointer into a single abstract variable and downgrade into a field-insensitive analysis for the particular struct pointer.

**Flow Sensitivity:** A flow-sensitive analysis considers the execution order of the statements in a function [27, 50, 23], while a flow-insensitive analysis conservatively assumes that the statements inside a function may execute in arbitrary order. Flow sensitivity typically improves pointer-analysis accuracy but when combined with context-sensitive analysis it can lead to scalability issues. To the best of our effort, we are unable to find any publicly available context-sensitive flow-sensitive pointer analysis that can scale to server applications such as Apache HTTPD.

## 3.4 Control Jujutsu: Attack on Fine-Grained CFI

We present a novel attack, *Control Jujutsu* that exploits the incompleteness of pointer analysis, when combined with common software engineering practices, to enable an attacker to execute arbitrary malicious code even when fine-grained CFI is enforced. The attack uses a new "gadget" class that we call Argument Corruptible Indirect Call Site (ACICS).

### 3.4.1 Argument-Corruptable Indirect Call Site Gadgets

ACICS gadgets are pairs of Indirect Call Sites (ICS) and target functions that enable Remote Code Execution (RCE) while respecting a CFG enforced using fine-grained CFI. Specifically, ACICS gadgets 1) enable argument corruption of indirect call sites

(data corruption) that in conjunction with the corruption of a forward edge pointer 2) can direct execution to a target function that when executed can exercise remote code execution (e.g., system calls). We show that for modern, well engineered applications (Apache and Nginx), ACICS gadgets are readily available as part of the intended control transfer.

### ACICS Requirements and Discovery

Control Jujutsu begins with a search for suitable ICS sites for the ACICS gadget. Control Jujutsu identifies the following requirements for ICS locations and target:

1. The forward edge pointer and its argument(s) should reside on the heap or a global variable to facilitate attacks from multiple data flows.

2. The arguments at the ICS can be altered without crashing the program (before reaching a target function).

3. The ICS should be reachable from external input (e.g., a network request).

4. The target site should reside in functions that exercise behavior equivalent to a RCE (e.g., *system* or *exec* calls).

We automate the discovery of ACICS gadgets using the ACICS Discovery Tool (ADT). To help discover candidate ICS/target function pairs (ACICS gadgets), ADT dynamically instruments applications using the GDB 7.0+ reverse debugging framework. For each candidate ACICS gadget, ADT runs a backward data-flow analysis that discovers the location of the ICS function pointer (and its arguments) in memory. Once a candidate pair is identified, ADT automatically corrupts the forward edge pointer and its arguments to verify that remote code execution can be achieved. Below, we describe ADT's approach in detail.

As input, ADT takes a target program, a list of candidate indirect call sites (ICS), sample inputs that exercise the desired program functionality (and the list of ICS), and the address of a candidate target function inside the target program. For each ICS location, ADT performs the following steps:

1. **Reach ICS:** ADT instruments program execution, using the GDB framework, with the ability to perform reverse execution analysis once program execution reaches a candidate ICS location.

2. **Backward Dataflow Analysis:** Once execution reaches the ICS location, ADT performs a backward reaching-definition dataflow analysis from the registers containing the target function address and its arguments to the memory locations that hold their values.

3. **Determine Last Write IP:** Next, ADT needs to identify program locations that can be used to corrupt the ICS function pointer and its values. To do this, ADT restarts the debugger and instruments the memory addresses, identified in the previous step, to record the code locations (i.e., the instruction pointer) that perform memory writes to these locations.

4. **Corrupt Function Pointers and Arguments:** At this point, ADT is able to restart the debugger and halt the program at the ideal point identified in the previous step. Then ADT redirects the ICS function pointer and its arguments to the target function. Additionally, by tracking every statement executed until the target ICS is reached, a lower bound of the liveness of the ACICS can be reported.

   The liveness of an ACICS allows us to reason about its exploitability; if the liveness persists across the program lifecycle, the ICS can be attacked by almost any memory read/write vulnerability, regardless of where it occurs temporarily. On the other hand, an ACICS whose liveness is contained in a single function is significantly less exploitable.

5. **ACICS validation:** Finally, ADT validates the ACICS gadget by verifying that the target function is reached, the argument values match the values in the corruption step and ultimately verifying that the target function can exercise functionality equivalent to remote code execution (e.g., create a file, launch a process, etc.).

### 3.4.2 ACICS in Apache HTTPD

#### Suitable Indirect Call Sites

Using these requirements and the ADT tool, we evalutaed the unoptimized Apache binary and found that the server contains 172 indirect call sites (ICS). We limit our evaluation to the core binary and omit reporting potential ICS target in other Apache modules, such as the Apache Portable Runtime (APR) and APR-util libraries. From these 172 sites, we want to find a subset of sites 1) which are exercised when the program processes a request and 2) whose forward edge pointer and arguments can be successfully corrupted by our ADT tool without crashing the program.

We run our ADT tool described in Section 3.4.1 on each of the 172 sites. We use a test script program that sends simple HTTP GET requests to drive our experiments. There are 51 sites exercised in our experiments. The remaining 121 sites do not satisfy our requirement, because they are either inside specific modules that are not enabled by default or depend on specific functionalities that a simple HTTP GET request does not exercise.

Table 3.1 presents the classification results of ICS exercised during different execution stages of Apache. In order to detect whether an ICS is exercised during the HTTP GET request life cycle or the startup, we vary when the test script is called in our tool. Our results show that there are 20 sites exercised during an HTTP GET request life cycle and 45 sites exercised during startup. Note that some of sites exercised during startup are also exercised by an HTTP GET request .

We use our ADT tool to detect the location of the forward edge pointer and arguments of each of the exercised 51 ICS and to corrupt these values. Table 3.2 presents our experimental results. Of the 51 ICS that are exercised dynamically in our experiments, our tool successfully corrupt forward edge pointers for 34 ICS. For 3 ICS our tool successfully corrupted both the forward edge pointers and the arguments.

| | |
|---|---|
| Total ICS | 172 |
| Exercised in HTTP GET request | 20 |
| Exercised during startup | 45 |
| Unexercised | 121 |

Table 3.1: Indirect Call Sites Dynamic Analysis

| | |
|---|---|
| Number of ICS dynamically encountered | 51 |
| Detected forward edge pointer on the heap/global | 34 |
| Automatically corrupted forward edges | 34 |
| Automatically corrupted forward edges + arguments | 3 |

Table 3.2: Automatic Corruption Analysis

### Target Functions

We run a script that searches the Apache source code for system calls that we can use to trigger behaviors equivalent to RCE such as `exec()` and `system()`. For each function in Apache, the script measures the distance between the function and a function that contains such system calls.

Table 3.3 presents the results. The farther away a target function is in the CallGraph, the harder it generally is to use it in the payload. At the same time, more viable functions become available. A related work has found similar results for the Windows platform [22]. Our example Apache exploit in the next section uses `piped_log_spawn()`, which is two calls away from the system call.

| Direct calls to system calls | 1 call away | 2 calls away |
|---|---|---|
| 4 | 13 | 31 |

Table 3.3: Target Functions Count Based on CallGraph distance

### Proof of Concept Attack

Here we present a detailed example exploit based on the selected ICS seen in Figure 3-1. Lines 1-5 use a macro defined in the Apache Portable Runtime (APR) library to define the function `ap_run_dirwalk_stat()`. Lines 7-30 present the simplified code snippet of `ap_run_dirwalk_stat()` after macro expansion. The actual ICS itself occurs at line 23, which invokes the function pointer `pHook[n].pFunc`.

```
AP_IMPLEMENT_HOOK_RUN_FIRST(apr_status_t,dirwalk_stat,
  (apr_finfo_t *finfo,
   request_rec *r,
   apr_int32_t wanted),
  (finfo, r, wanted), AP_DECLINED)

apr_status_t ap_run_dirwalk_stat(
 apr_finfo_t *finfo, request_rec *r,
 apr_int32_t wanted) {
  ap_LINK_dirwalk_stat_t *pHook;
  int n;
  apr_status_t rv = AP_DECLINED;
  ...
  //check the corresponding field of the global _hooks
  if (_hooks.link_dirwalk_stat) {
    pHook = (ap_run_dirwalk_stat_t *)
             _hooks.link_dirwalk_stat->elts;
   //invoke registered functions in the array one by
   //one until a function returns a non-decline value.
    for(n=0; n < _hooks.link_dirwalk_stat->nelts;++n){
      ...
      // our seelcted ICS
      rv = pHook[n].pFunc(finfo, r, wanted);
      ...
      if (rv != AP_DECLINED) break;
    }
  }
  ...
  return rv;
}
```

Figure 3-1: APR hook macro in server/request.c:97 defining `ap_run_dirwalk_stat()` in Apache HTTPD and the simplified code snippet of `ap_run_dirwalk_stat()`

```
if (r->finfo.filetype == APR_NOFILE ||
    r->finfo.filetype == APR_LNK) {
  rv = ap_run_dirwalk_stat(&r->finfo,
                           r,
                           APR_FINFO_MIN);
```

Figure 3-2: *dirwalk_ stat* called in server/request.c:616 in Apache HTTPD

Figure 3-2 presents the specific `ap_run_dirwalk_stat()` call we use in our exploit.

Apache HTTPD uses a design pattern that facilitates modularity and extensibility. It enables Apache module developers to register multiple implementation function hooks to extend core Apache functionality. `ap_run_dirwalk_stat()` is a wrapper function that iteratively calls each registered implementation function for the *dirwalk* functionality until an implementation function returns a value other than `AP_DECLINED`.

As for the target, the `piped_log_spawn` function meets and exceeds all of our requirements. Apache allows a configuration file to redirect the Apache logs to a

```
1  /* Spawn the piped logger process pl->program. */
2  static apr_status_t piped_log_spawn(piped_log *pl)
3  {
4    apr_procattr_t *procattr;
5    apr_proc_t *procnew = NULL;
6    apr_status_t status;
7
8    ...
9    char **args;
10   apr_tokenize_to_argv(pl->program, &args, pl->p);
11   procnew = apr_pcalloc(pl->p, sizeof(apr_proc_t));
12   status = apr_proc_create(procnew,
13                     args[0],
14                     (const char * const *) args,
15                     NULL, procattr, pl->p);
16   ...
17 }
```

Figure 3-3: Target function *piped_log_spawn* in Apache HTTPD

pipe rather than a file; this is commonly used by system administrators to allow transparent scheduled log rotation. This functionality involves Apache reading its configuration file, launching the program listed in the configuration file along with given arguments, and then connecting the program's standard input to Apache's log output.

Figure 3-3 presents a simplified version of the example target function, `piped_log_spawn`. This target function accepts a pointer to the `piped_log` structure as an argument. `piped_log_spawn` invokes an external process found in the `char *program` field of the `piped_log` structure.

### 3.4.3 ACICS in Nginx

#### Suitable Indirect Call Sites

Our analysis on the unoptimized Nginx binary shows that there are 314 ICS in Nginx. We run our ADT tool on each of the 314 ICS in a way similar to our Apache experiments. Table 3.4 presents the classification results of ICS based on different execution stages and Table 3.5 presents the corruption experiment results.

Our results show that there are 36 ICS exercised during our Nginx experiments and 27 of these ICS are exercised during an HTTP GET request lifecycle after Nginx startup. Of the 36 exercised ICS, our ADT tool successfully corrupted the forward

| | |
|---|---|
| Total ICS | 314 |
| Exercised in HTTP GET request | 27 |
| Exercised during startup | 18 |
| Unexercised | 278 |

Table 3.4: Indirect Call Sites Dynamic Analysis

| | |
|---|---|
| Number of ICS dyanmically encountered | 36 |
| Detected forward edge pointer on the heap/global | 7 |
| Automatically corrupted forward edges | 7 |
| Automatically corrupted forward edges + arguments | 4 |

Table 3.5: Automatic Corruption Analysis

edge pointers and arguments for 4 ICS.

We found that the ICS at core/ngx_output_chain.c:74 in `ngx_output_chain()` is an ideal candidate ICS for our attack. Figure 3-4 presents the simplified code snippet of `ngx_output_chain()`. The ICS is at line 27 in Figure 3-4. The function implements the filter chaining mechanism that is inherent to Nginx's modular design because it gives an easy way to manipulate the output of various handlers run on the request object to generate a response.

In this function, the function pointer `ctx->output_filter` and arguments `ctx->filter_ctx` are all derived from `ctx` which is a `ngx_output_chain_ctx` struct pointer. This `ctx` a global object lives on the heap, so that our tool successfully corrupts all of these values.

Secondly, the argument `ctx->filter_ctx` is a void pointer that is written only once during the request life cycle, whereas argument `in` is a pointer to the head of a linked list of filters that are applied to request responses. This linked list is modified in every module that implements a filter. However with manual dataflow analysis, it is possible to modify this linked list so that the checks at lines 18, 19, and 20 of Figure 3-4 pass and we reach the execution of the ICS before any crash happens. Thirdly, as all response body filters are called before the response is returned to the user, we were able to remotely exercise this ICS during the request life cycle.

```
ngx_int_t
ngx_output_chain(ngx_output_chain_ctx_t *ctx,
                 ngx_chain_t *in)
{
  ...

  if (ctx->in == NULL && ctx->busy == NULL)
  {
    /*
     * the short path for the case when the ctx->in
     * and ctx->busy chains are empty, the incoming
     * chain is empty too or has the single buf
     * that does not require the copy
     */

    if (in == NULL) {
      return ctx->output_filter(ctx->filter_ctx, in);
    }

    if (in->next == NULL
#if (NGX_SENDFILE_LIMIT)
        && !(in->buf->in_file && in->buf->file_last
             > NGX_SENDFILE_LIMIT)
#endif
            && ngx_output_chain_as_is(ctx, in->buf))
    {
      return ctx->output_filter(ctx->filter_ctx, in);
    }
  }
  ...
}
```

Figure 3-4: ACICS for Nginx found in *ngx_output_chain* function

### Target Functions

We use a script to search Nginx source code for system calls with RCE capability. Table 3.6 shows the number of potential targets based on the distance in the call graph. We found that the function `ngx_execute_proc()` (shown in Figure 3-5) is an ideal target function for our proof-of-concept attack, because it executes a `execve()` call with passed-in arguments and it has a small arity of 2, which facilitates the type punning.

| Direct calls to system calls | 1 call away | 2 calls away |
|---|---|---|
| 1 | 2 | 3 |

Table 3.6: Target Functions Count Based on CallGraph distance

```
1  static void
2  ngx_execute_proc(ngx_cycle_t *cycle, void *data)
3  {
4    ngx_exec_ctx_t  *ctx = data;
5
6    if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
7      ngx_log_error(...);
8    }
9    exit(1);
10 }
```

Figure 3-5: Nginx Target Function that calls *execve*

```
1  void ap_hook_dirwalk_stat(ap_HOOK_dirwalk_stat_t *pf,
2                                                ...) {
3    ap_LINK_dirwalk_stat_t *pHook;
4    //check the corresponding field of the global _hooks
5    if (!_hooks.link_dirwalk_stat)
6      _hooks.link_dirwalk_stat = apr_array_make(...);
7    // store the function pointer pf into the array
8    pHook = apr_array_push(_hooks.link_dirwalk_stat);
9    pHook->pFunc = pf;
10   ...
11 }
```

Figure 3-6: The code snippet for `ap_hook_dirwalk_stat()` in Apache HTTPD

### Proof-of-concept Attack

Hence, we identified the ACICS gadget pair for our attack which is composed of the ICS at core/ngx_output_chain.c:74 in `ngx_output_chain()` (see line 27 in Figure 3-4) and the target function `ngx_execute_proc()` (see Figure 3-5).

We then perform the attack as follows. We corrupt `ctx->output_filter` to point to the target function `ngx_execute_proc()` and we corrupt the memory region that `in` points to so that when the memory region is viewed as a `ngx_exec_ctx_t` struct in `ngx_execute_proc()`, it will trigger RCE at line 6 in Figure 3-5. We successfully achieved RCE with our attack.

## 3.4.4 Challenges for Stopping Control Jujutsu

As previously mentioned, the construction of a precise CFG requires a pointer analysis to determine the set of functions to which the pointer at each indirect call site can point. For example, CFG has to capture where ICS on line 23 in the Apache example shown in Figure 3-1 can point.

Figure 3-6 presents a simplified version of `ap_hook_dirwalk_stat()`, which reg-

isters implementation functions that `ap_run_dirwalk_stat()` (shown in Figure 3-1) can later invoke for the functionality of *dirwalk_ stat*. The intended behavior of the ICS shown at line 23 in Figure 3-1 is to only call implementation functions registered via `ap_hook_dirwalk_stat()` in Figure 3-6.

The example code in Figure 3-1 and Figure 3-6 highlights the following challenges for the static analysis:

- **Global Struct:** The analysis has to distinguish between different fields in global variables. Hence, field-sensitivity is important to stop Control Jujutsu using CFI. `_hooks` in Figure 3-1 and Figure 3-6 is a global struct variable in Apache HTTPD. Each field of `_hooks` contains an array of function pointers to registered implementation functions for a corresponding functionality. For example, the `link_dirwalk_stat` field contains function pointers to implementation functions of the functionality *dirwalk_ stat*.

- **Customized Container API:** The analysis has to capture inter-procedural data flows via customized container APIs. Hence, context-sensitivity is important to stop Control Jujutsu using CFI. The code in Figure 3-1 and Figure 3-6 uses customized array APIs `apr_array_push()` and `apr_array_make()` to store and manipulate function pointers.

- **Macro Generated Code:** The code shown in Figure 3-1 and Figure 3-6 is generated from macro templates found in Apache Portable Runtime library. For example, for a functionality *malicious*, there are pairs of functions `ap_hook_malicious()` and `ap_run_malicious()` that are structurally similar to the code shown in Figure 3-1 and Figure 3-6. This imposes a significant additional precision requirement on the static analysis, as it needs to consider a (potentially) large number of similar functions that can manipulate the data structures inside `_hooks`.

### Data Structure Algorithm (DSA)

As discussed above, the combination of context sensitivity and field sensitivity is critical for generating a precise CFG that can stop the attack described in Section 3.4.2. We next present the results of using the DSA algorithm [34] to generate a CFG for Apache HTTPD. We chose the DSA algorithm because, to the best of our knowledge, it is the only analysis that 1) is context-sensitive and field-sensitive, 2) can scale to server applications like Apache HTTPD and Nginx, and 3) is publicly available.

The DSA algorithm is available as a submodule of the LLVM project [2] and is well maintained by the LLVM developers. It works with programs in LLVM intermediate representation (IR) generated by the LLVM Clang compiler [1].

Unfortunately, the DSA algorithm produces a CFG that cannot stop the attack in Section 3.4.2. Specifically, the CFG specifies that the indirect call at line 26 in Figure 3-6 may call to the function `piped_log_spawn()`. We inspected the debug log and the intermediate pointer analysis results of the DSA algorithm. We found that although as a context-sensitive and flow-sensitive analysis the DSA algorithm should theoretically be able to produce a precise CFG to stop the attack, the algorithm in practice loses context sensitivity and flow sensitivity because of convoluted C idioms and design patterns in Apache HTTPD and the APR library. As a result, it produces an imprecise CFG. Fine-grained CFI systems that disallow the calling of functions whose address is not taken can prevent the proposed attack through `piped_log_spawn()`. The attack can succeed, however, by targeting `piped_log_spawn()` indirectly through functions such as `ap_open_piped_log_ex()`, whose address is directly taken by the application. We describe some of the sources of imprecision in more detail below.

- **Struct Pointer Casts:** We found that struct pointer-cast operations in Apache HTTPD cause the DSA algorithm to lose field sensitivity on pointer operations. Pointer casts are heavily used at the interface boundaries of Apache components. There are in total 1027 struct pointer conversion instructions in the generated

bitcode file of Apache HTTPD.

For example, pointers are cast from `void*` to `apr_LINK_dirwalk_stat_t *` at line 8 in Figure 3-6 when using the array container API `apr_array_push()`. Apache HTTPD also uses its own set of pool memory management APIs and similar pointer casts happen when a heap object crosses the memory management APIs. When the DSA algorithm detects that a memory object is not accessed in a way that matches the assumed field layout of the object, the algorithm conservatively merges all fields into a single abstract variable and loses field sensitivity on the object.

- **Integer to Pointer Conversion:** Our analysis indicates that the Clang compiler generates an integer to pointer conversion instruction (`inttoptr`) in the bitcode file for the APR library function `apr_atomic_casptr()`, which implements an atomic pointer compare-and-swap operation.

  For such `inttoptr` instructions, the DSA algorithm has to conservatively assume that the resulting pointer may alias to any pointers and heap objects that are accessible at the enclosing context. Although such instructions are rare (`apr_atomic_casptr()` is called three times in the Apache HTTPD source code), they act as sink hubs that spread imprecision due to this over-conservative aliasing assumption.

- **Cascading Imprecision:** The struct pointer casts and integer to pointer conversions are the root sources of the imprecision. One consequence of the imprecision is that the DSA algorithm may generate artificial forward edges (calls) for indirect call sites.

  Although initially such artificial forward edges may not directly correspond to attack gadgets in the Apache HTTPD, they introduce artificial recursions to the call graph. Because maintaining context sensitivity for recursions is undecidable, the DSA algorithm has to conservatively give up context sensitivity for

the function calls between functions inside a recursive cycle (even they are artificially recursive due to the analysis imprecision). This loss of context sensitivity further introduces imprecision in field sensitivity because of type mismatch via unrealizable information propagation paths.

In our Apache HTTPD example, this cascading effect continues until the DSA algorithm reaches an (imprecise) fix-point on the analysis results. As a result, 51.3% of the abstract struct objects the DSA algorithm tracks are merged into single abstract variables (i.e., the loss of field sensitivity); we observed a phenomenal artificial recursion cycle that contains 110 functions (i.e., due to the loss of context sensitivity). Some of this imprecision may be attributed to changes in LLVM IR metadata since version 1.9. Previous versions relied on type annotations that used to persist from the llvm-gcc front-end into the LLVM IR metadata that are no longer available. LLVM DSA prior to version 1.9 used a set of type-based heuristics to improve the accuracy of the analysis. Aggressive use of type-based heuristics is unsound and could introduce false negatives (opening up another possible set of attacks).

### CFG Construction using DSA

We next evaluate the precision of CFG construction using the DSA algorithm on four popular server applications: Apache HTTPD, Nginx, vsftpd, and BIND. Specifically, we evaluate the loss of context sensitivity by measuring the maximum size of strongly connected components and the loss of field sensitivity by measuring the number of merged objects. We performed all of our experiments on an Intel 2.3GHz machine running Ubuntu 14.04.

| Program | LoC | LLVM IR | Max. SCC Size | Merged% | Time |
|---|---|---|---|---|---|
| HTTPD | 272K | 318K | 110 | 51.3% | 14s |
| Nginx | 123K | 358K | 38 | 10.8% | 10s |
| vsftpd | 16K | 24K | 255 | 70.5% | 1s |
| BIND | 462K | 1167K | 1023 | 41.2% | 14m52s |

Table 3.7: DSA analysis statistics

Table 3.7 summarizes the results. The first column presents the application name. The second and third columns represent the source code line count and LLVM IR

count respectively. The application size ranges from 17K LoC for vsftpd to approximately 460K LoC for BIND.

The fourth column presents the number of functions in the largest (potentially artificial) recursion cycle DSA algorithm found for each application. High numbers translate to high loss of context sensitivity. The fifth column presents the percentage of the abstract struct objects that the DSA algorithm tracks which the DSA algorithm merges conservatively. High percentage numbers indicate high loss of field sensitivity.

Together, columns four and five show that the DSA algorithm is unable to produce satisfactory results on any of the four applications due to the loss of field sensitivity and context sensitivity. DSA loses field sensitivity on up to 70.5% of tracked struct objects and detects artificial recursion groups that contain up to 1023 functions.

Note that even for Nginx, where the relative loss is small, the generated CFG is unable to stop the ASICS gadgets index Section 3.4.3. The CFG allows the ICS found in core/ngx_output_chain.c:74 (line 27 in Figure 3-4) to call the target function `ngx_execute_proc` shown in Figure 3-5 due to the pointer analysis imprecision.

The sixth column presents the running time of the DSA algorithm on each application. Our results show that the running time of the DSA algorithm grows non-linearly to the amount of analyzed code. For BIND, the algorithm needs more than 14 minutes to finish. This result highlights the difficult trade-offs between the accuracy and the scalability in pointer analysis algorithms.

# Chapter 4

# Conclusion

We present an attack on the recently proposed CPI technique as well as an attack that is able to bypass fine-grained enforcement of CFI.

For CPI, we show that the use of information hiding to protect the safe region is problematic and can be used to violate the security of CPI. Specifically, we show how a data pointer overwrite attack can be used to launch a timing side channel attack that discloses the location of the safe region on x86-64. We evaluate the attack using a proof-of-concept exploit on a version of the Nginx web server that is protected with CPI, ASLR and DEP. We show that the most performant and complete implementation of CPI (simpletable) can be bypassed in 98 hours without crashes, and 6 seconds if a small number of crashes (13) can be tolerated. We also evaluate the work factor required to bypass other implementations of CPI including a number of possible fixes to the initial implementation. We show that information hiding is a weak paradigm that often leads to vulnerable defenses.

For CFI, Control Jujutsu is an attack that exploits the imprecision of scalable pointer analysis to bypass fine-grained enforcement of CFI. The attack uses a new "gadget" class, Argument Corruptible Indirect Call Site (ACICS), that can hijack control flow to achieve remote code execution while still respecting control flow graphs generated using context- and field-sensitive pointer analysis.

We show that preventing Control Jujutsu by using more precise pointer analysis algorithms is difficult for real-world applications. In detail, we show that code design

patterns for standard software engineering practices such as extensibility, maintainability, and modularity make precise CFG construction difficult.

Our results provide additional evidence that techniques that trade off memory safety (security) for performance are vulnerable to motivated attackers. This highlights the need for fundamental memory protection techniques such as complete memory safety and indicates that the true cost of memory protection is higher than what is typically perceived.

# Bibliography


[1] Clang. http://clang.llvm.org/.

[2] The LLVM Compiler Infrastructure. http://llvm.org/.

[3] Linux cross reference, 2014.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proc. of ACM CCS*, 2005.

[5] Manuel Costa Miguel Castro Akritidis, Periklis and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors.

[6] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[7] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[8] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[10] T. Bletsch, X. Jiang, V.W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proc. of ACM CCS*, 2011.

[11] Ryan Roemer Hovav Shacham Buchanan, Erik and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *In Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[12] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[13] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[14] Vincenzo Iozzo Charlie Miller. Fun and games with mac os x and iphone payloads. *BlackHat Europe*, 2009.

[15] Hai Jin Deqing Zou Bing Bing Zhou Zhenkai Liang Weide Zheng Chen, Gang and Xuanhua Shi. Safestack: automatically patching stack-based buffer overflow vulnerabilities. 2013.

[16] Zongwei Zhou Yu Miao Xuhua Ding Cheng, Yueqiang and Huijie DENG. Ropecker: A generic and practical approach for defending against rop attack. 2014.

[17] Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):17, 2009.

[18] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[19] Colin Blundell Milo MK Martin Devietti, Joe and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language.

[20] Ulrich Drepper. Elf handling for thread-local storage, 2013.

[21] Ivan Fratric. Ropguard: Runtime prevention of return-oriented programming attacks. 2012.

[22] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[23] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proc. of POPL*, 2009.

[24] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proc. of PLDI*, 1998.

[25] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[26] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of PASTE*, 2001.

[27] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 1999.

[28] intel. Introduction to intel memory protection extensions, 2013.

[29] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[30] Tim Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-Universitat Bochum, 2010.

[31] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. 2014.

[32] Albert Kwon, Udit Dhawan, Jonathan Smith, Thomas Knight, and Andre Dehon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732. ACM, 2013.

[33] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. 2011.

[34] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of PLDI*, 2007.

[35] Ali Jose Mashtizadeh, Andrea Bittau, David Mazieres, and Dan Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[36] Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. Online, September 2006.

[37] Vishwath Mohan, Per Larsen, Stefan Brunthaler, K Hamlen, and Michael Franz. Opaque control-flow integrity. In *Proc. of NDSS*, 2015.

[38] Jianzhou Zhao Milo MK Martin Nagarakatte, Santosh and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c.

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, 2010.

[40] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices*, 37(1):128–139, 2002.

[41] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[42] OpenBSD. Openbsd 3.3, 2003.

[43] Vasilis Pappas. kbouncer: Efficient and transparent rop mitigation. 2012.

[44] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[45] Colin Percival. How to zero a buffer, September 2014.

[46] Vincenzo Iozzo Ralph-Philipp Weinmann. Ralph-philipp weinmann & vincenzo iozzo own the iphone at pwn2own.

[47] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[48] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[49] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, January 2000.

[50] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proc. of PLDI*, 1999.

[51] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming. In *Proc. of IEEE S&P*, 2015.

[52] Jeff Seibert, Hamed Okhravi, and Eric Soderstrom. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proc. of ACM CCS*, 2014.

[53] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[54] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc.of ACM CCS*, 2007.

[55] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proc. of ACM CCS*, pages 298–307, 2004.

[56] 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 *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[57] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proc. of PLDI*.

[58] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proc.of EuroSec '09*, 2009.

[59] The PaX Team. Address space layout randomization. March 2003. http://pax.grsecurity.net/docs/aslr.txt.

[60] Tom Roeder Peter Collingbourne Stephen Checkoway Úlfar Erlingsson Luis Lozano Tice, Caroline and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. 2014.

[61] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proc. of RAID'11*, 2011.

[62] Robert NM Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, Munraj Vadera, and Khilan Gudka. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*, 2015.

[63] Yoav Weiss and Elena Gabriela Barrantes. Known/chosen key attacks against software instruction set randomization. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 349–360. IEEE, 2006.

[64] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of PLDI*, 2004.

[65] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proc. of IEEE S&P*, 2013.

[66] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security*, pages 337–352, 2013.