Analysis of Defenses Against Code Reuse Attacks on Modern and New Architectures

by

Isaac Noah Evans

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author . . . .
Department of Electrical Engineering and Computer Science
September 2015

Certified by. . . . .
Dr. Hamed Okhravi
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

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

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Analysis of Defenses Against Code Reuse Attacks on Modern and New Architectures

by

Isaac Noah Evans

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

Abstract

Today, the most common avenue for exploitation of computer systems is a control-flow attack in which the attacker gains direct or indirect control of the instruction pointer. In order to gain remote code execution, attackers then exploit legitimate fragments of code in the executable via techniques such as return-oriented-programming or virtual table overwrites.

This project aims to answer fundamental questions about the efficacy of control-flow-integrity (CFI), a defensive technique which attempts to prevent such attacks by ensuring that every control flow transfer corresponds to the original intent of the program author. Although this problem is in general undecidable, most programs running on modern operating systems adhere to standard conventions which allow inferences from static analysis to set a specification for allowable runtime behavior.

1. By examining extremely large, complex real-world programs such as web browsers, this project will characterize the fundamental limits of CFI techniques. We find that it is possible for a program in which CFI is perfectly enforced to be exploited via a novel control flow attacks.

2. We examine the potential for hardware support for CFI and other techniques via generalized tagged architectures, and explore the tradeoff between the compatibility, performance, and security guarantees of hardware-assisted policies on tagged architectures.

Thesis Supervisor: Dr. Hamed Okhravi
Title: Technical Staff, MIT Lincoln Laboratory

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

Acknowledgments

I would like to thank my advisors, Dr. Howard Shrobe and Dr. Hamed Okhravi, for their encouragement, insights, and thoughtful commentary and criticism. Stelios Sidiroglou-Douskos, at CSAIL, also became an unofficial advisor and a great mentor on all things academic. I was very fortunate to be able to work with such talented mentors.

I also thoroughly enjoyed working with my teammates Ulziibayar Otgonbaatar and Tiffany Tang on the software side and Julián Gonzalez and Sam Fingeret on the hardware side. This work would not have been nearly as interesting or fun without them.

And how could I not thank my wonderful parents: my father who challenged me, encouraged me, and inspired my dreams and ambitions, and my mother, who poured so much of her life into my education and always reminds me that "I can do all things through Christ who strengthens me."

Finally, I would not have even started this process without the encouragement of the lovely Katherine and her constant support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Modern computer systems were not architected with an emphasis on security.

At the dawn of the computing era, most of the focus of computer research was on speed and capabilities. As computers have grew more affordable, engineers added usability—computing for the masses. Now in an age that is thoroughly dominated by fast, capable, and easily usable computers, there is an increasing demand for security. A principal reason for this demand is the rise of the Internet. As a global network that connects billions of devices and users at speeds which render our notions of physical distance nonsensical, the Internet throws the problem of malicious actors into sharp relief.

There are many avenues for mischief on the Internet, but few so exciting and powerful as "remote code execution" (RCE). An attacker sends a specially crafted message to a target machine, and suddenly that machine begins running the attacker's program. One might hope that scenarios such as this are rare; but in fact there are often dozens to hundreds of flaws in software such as web browsers and operating systems [8, 3, 2] that lead to remote code execution.

Are the programmers writing these systems inept? Sometimes. The more serious problem, however, is that the programming languages and hardware they are writing for were designed for the customers needing maximal speed and features for their

programs. Unfortunately, this emphasis on close-to-the-metal performance means that languages like C and C++ have emerged as the choice for most large software projects. In these languages, the penalty for incredibly small mistakes—overflowing an integer, failing to properly check the length of an array, or being overly accepting of user input during sanitization—can often be remote code execution.

The fact that there exist entire catalogs of vulnerability types which exist in some programming languages but not others is indicative of the idea that this is not the programmer's problem. The penalty for failure should not be so high. Worse, the rates for discovery and exploitation of these vulnerabilities are increasing [32].

Indeed, higher-level languages, in particular those with memory safety, eliminate several entire categories of vulnerabilities. In fact some older programming languages such as LISP had many desirable security properties: memory safety, tagged pointer types, and integer overflow detection. There is a currently a great deal of interest in bringing these strong guarantees against entire classes of vulnerabilties to lower-level languages such as C and C++, which account for the vast majority of operating systems and performant applications.

1.2 Outline

This thesis evaluates techniques built around the idea of control-flow integrity (CFI). CFI attempts to define a policy for the correct and safe flow of the instruction pointer, derived from the previously determined program control-flow graph (CFG). The CFG may be defined by static source-code analysis, static binary inspection, or dynamic instrumentation.

Although CFI has been examined extensively in prior work, there has been an unstated assumption regarding the underlying effectiveness of the technique. Papers typically assume a threat model in which an attacker can read or write arbitrary memory and then develop a policy designed to detect such alterations and defend against them. A perfect policy is extremely costly to implement, so typically an approximation is used. For instance, simply verifying that all calls go to the begining

16

of function bodies.

We will demonstrate that even an ideal CFI technique—without any compromises for performance—cannot fully protect from control-flow attacks, assuming, as is typical in the literature, a powerful attacker who can read or write any location in memory. The ideal CFI technique employs a "perfect" control flow graph, which is defined as a the combination of an infinite shadow stack, an infinite number of available tags (where a function may only call other functions that have the correct target tag), and a state-of-the-art static analysis.

In Chapter 2, we give an broad overview of the state-of-the-art in defense against code reuse attacks, including CFI and other creative techniques.

In Chapter 3, we analyze a novel system called "Code Pointer Integrity" which aims to preserve the benefits of full memory safety at a fraction of the performance costs by protecting only the code pointers of a program.

In Chapter 4, we show that there exist pairs of call sites and target functions through which an attacker can gain remote code execution without deviating from the strict constraints imposed by a shadow stack and an unlimited number of function tags drawn from a best-possible static analysis.

In conjunction with the exploration of the control-flow integrity work, Chapter 5 details the development of compiler extensions to support a policies on tagged extension to the new RISC-V [9] architecture. These compiler extensions are designed to enforce CFI mechanisms and more generally prevent control-flow attacks through hardware-provided extensions making use of compiler-provided tags. Chapter 6 details the design and evaluation of a restrictive function-pointer policy policy running on the extended hardware.

Finally, Chapter 7 details avenues of future work, including improvements to the tools and algorithms we have presented for CFI analysis and hardware policy evaluation.

Chapter 2

Background

There is an extensive body of work examining methods to prevent control-flow attacks. To aid in evaluating the numerous creative techniques, we categorize them into several families of approaches.

2.1 Code Diversification

A code reuse attack is typically employed when the virtual memory manager is enforcing "write xor execute" permissions on memory pages. Since new code cannot be injected by directly writing to an executable page, an attacker uses small snippets of instructions in the original program to achieve remote code execution. The snippets of code that used in a code reuse attack are called "gadgets." [13]. When chained together appropriately, these gadgets create what is termed a "weird machine" where the instructions are gadgets—chains of instructions which were never intended to be used in this fashion.

Code diversification aims to remove the gadgets from the program completely, or effectively remove them by making it probabilistically impossible for the attacker to guess where they are. If implemented correctly, this approach results in no known fragments of aligned or unaligned executable code that can be used by the attacker to exploit the system. In-Place Code Randomization [36] accomplishes this by changing the order of instructions and replacing with equivalents while examining the assembly

listing to ensure that useful gadgets are never created. Binary Stirring [47] reorders the basic blocks of an program each time it is executed, randomizing their locations. Another approach is NOP insertion, used in projects such as the MultiCompiler [22]. An extensive summary of code diversification is provided in [28].

The advantage of code diversification is extremely low overhead (1% for Binary Stirring, for instance) and high compatibility—it can often be implemented via binary instrumentation, which does not require program source. However, these techniques have been shown to fail to defend against powerful new attack techniques [43, 10]. They do not protect against an attacker with the ability to read or write to an arbitrary memory location. They also cannot protect programs that perform Just-In-Time (JIT) compilation. Other weaknesses exist: side channels can leak the location of the randomized gadgets [41], and injected addresses will be translated and can be leaked.

2.2 Memory Safety

Full memory safety is an optimal solution for the most pessimistic attack model, since it eliminates the potential for unsafe memory reads and writes, thus perfectly protecting all arrays and variables. Full memory safety prevents a wide variety of non-temporal control flow attacks including ROP, stack smashing, virtual table corruption, and others. It can also be extended to protect against temporal attacks such as use-after-free. The overhead of this approach is an upper bound for any ROP-mitigation technique. SoftBound [34] is the de facto standard for memory safety; it achieves 67-250% overhead. Unfortunately, in practice the overhead for these techniques on large, real-world applications such as Mozilla Firefox is typically over 100%, as seen in the AddressSanitizer [42] project. This prohibitively high overhead prevents widespread adoption.

2.3 Protect the Stack

In principle, ensuring the integrity of stack values should be useful in protecting against control-flow attacks such as ROP if the program is coded using a higher-level language like C that conforms to the architecture's calling convention. However, protecting the stack poses problems for compatibility because it is not always true in compiled real-world executables that a callee function always returns to the address pushed by its caller. The ROPDefender paper [16] shows three cases where assumption fails: a called function may not return, a function might be invoked without an explicit call, and the C++ standard library exception handlers violate this concept. Although it is possible to overcome these compatibility concerns, a more serious issue is that stack protection is of limited utility since it does not protect against other control flow attacks (e.g. virtual table corruption, function pointers on the heap, etc.). Additionally, the overhead of building a shadow stack can be prohibitive. Examples of this approach include RopDefender [16], which uses Pin, a JIT-based dynamic binary instrumentation framework to detect ROP via shadow stacks. Defending Embedded Systems Against Control Flow Attacks [21] allocates separate, protected stack for return addresses on the AVR architecture.

2.4 Control Flow Integrity

The original CFI paper [6] represented the CFG as a directed graph, but only enforced that calls correspond to valid call locations and returns correspond to valid return locations. This was quickly shown to be ineffective. CCFIR [49] expands on this approach, using a springboard enforcement mechanism that achieves 3-8% overhead by adding a separate third class of functions: returns to libc or other libraries. This also is not enough to prevent ROP gadgets from being built.

Other approaches to CFI rely on cryptographic mechanisms: Cryptographically Enforced Control Flow Integrity (CCFI) [31] performs a single round of AES as an secure message authentication code on pointer values to prevent their being forged by

an attacker, storing the key in a rarely-used x86 XMM register. GFree [35] performs "encryption" on return addresses in function prologue and then "decryption" before they end. K-Bouncer [37] performs runtime checks for abnormal control flow transfers when about to execute "important" functions (e.g. syscalls). However, K-Bouncer and a number of similar techniques have been recently shown to be fundamentally flawed, as we discuss in Section 3.1. There is also a recent, highly performant and compatible solution for forward-edge only (jumps and calls but not returns) control-flow integrity via indirection, by the Google Chromium toolchain team [45].

This work contributes to the attacks on CFI techniques by defining an upper bound on how effective any CFI technique can be on real-world programs.

2.5 Hardware Extensions

An optimal defense to the control-flow attack must assume the most pessimistic threat model: an attacker who can read and write arbitrary memory (typically dataflow attacks are considered out of scope). Hardware support is an ideal way to achieve this level of protection, although potentially at the expense of compatibility.

There are two basic approaches to preventing control-flow attacks in hardware: pointer-based schemes and data tagging schemes. On the pointer-based side, Hard-Bound + CETS [17] is an extension of the software approach in SoftBounds to hardware. WatchdogLite [33] is an ISA extension that is designed to efficiently execute hinted pointer safety checks, achieving overheads around 30%. The Low-Fat pointer scheme [27] uses a space-efficient representation inspired by floating-point representations to achieve only 3% memory overhead for pointers. The 64-bit fat pointers have 46 of the bits available for addressing; the encoding includes information for distinct types such as integers, instructions, and pointers.

There are also many projects implementing data-tagging schemes. The PUMP project [18] is a fully generic scheme that allows generalized checks based on meta-data or tags associated with all addresses, but at the cost of high (20-40%) overhead; it is essentially a completely generic scheme that expresses restrictions symbolically

and could be used to emulate in hardware many of the previously-discussed CFI schemes. The CHERI system [48] is another hardware scheme with fine-grained memory protection provided by a cooperative compiler. It supports a number of additional features including sandboxing and has been able to run modified versions of FreeBSD (CheriBSD) with pointer integrity and bounds checking. The call-and-return security mechanisms evaluated thus far in CHERI are however somewhat primitive, only supporting matching calls to call sites and returns to return sites.

This work implements a tagged architecture hardware approach as an extension to the RISC-V [9] ISA, due to its simplicity of implementation, versatility (with many applications beyond security), and compatibility. The lowRISC project [12] aims to add tag support to the new, open RISC-V ISA. Many of their goals are similar to ours, and we enjoyed several discussions with the lowRISC team at Cambridge University and a group of engineers at BAE Systems who are similarly interested in exploring tagged additions to RISC-V.

Chapter 3

Effectiveness of Code Pointer

Integrity

3.1 Inadequacy of Current Defenses

As seen in Section 2.4, there has been an arms race between the developers of defensive CFI schemes and increasingly-powerful exploitation techniques which break the defenses. Recently, a series of papers have shown that the most recent generation of CFI-based defenses are seriously flawed. The Out of Control paper [23] showed that full code execution could be achieved even when an attacker was restricted to only legitimate call and return sites inside common applications.

3.2 State-of-the-Art Defenses: Code Pointer Integrity

3.2.1 Background

A recently published technique, Code Pointer Integrity [26], employs a restricted, optimized version of full-memory safety that aims to protect only code pointers, trading off some of the guarantees of full-memory safety in exchange for dramatically better performance. CPI claims a 10-20% performance penalty on average across the SPEC benchmarks. The insight behind CPI is that if data pointers and code pointers

can be correctly separately identified, memory safety protections can be applied only to code pointers. Of course, CPI is not providing all the benefits of memory safety; dataflow attacks, information disclosures, and other non control-flow attacks are still possible. The low-overhead performance results are the significant novel component of the CPI system.

3.2.2   CPI in Detail

CPI Static Analysis

CPI separates the code pointers from data pointers via an LLVM analysis pass on the original source code. CPI divines what is a code pointer (in CPI parlance a "sensitive pointer") as opposed to a data pointer purely by examining the corresponding C data type. Any pointer to a function is obviously a member of the sensitive set, as are any composite types such as arrays which contain a sensitive type or any pointer to a sensitive type. Due to the prevalence of storing code pointers in types such as `char *` or `void *` by C programmers, CPI also considers those universal pointers to be sensitive types.

The number of code pointers identified is much lower than the number of data pointers; in practice about 10% in the experience of the CPI authors. This provides the key benefit of CPI, since the resulting performance overhead is much lower than full memory safety. A weaker variant of CPI, "Code Pointer Safety" (CPS) protects only the sensitive types (as opposed to also protecting pointers to sensitive types); a trade-off in security guarantees in exchange for even better (average 2% overhead) performance.

CPI Instrumentation

After identifying sensitive pointers, CPI inserts runtime checks which store and retrieve sensitive code pointers from a "safe region." For CPS, only the pointer values are stored; for CPI a lower and upper bound are stored along with the value. At any sensitive (i.e. indirect) control-flow transfer point, if a sensitive type is used,

CPI runtime instructions load the sensitive pointer from the safe region. Control-flow hijacking is thus prevented as there is an assumption that an attacker cannot modify the safe region and influence the load used in the indirect call.

The CPI instrumentation also guarantees, in theory, that no other instructions can access the safe region. CPI uses hardware memory segmentation support to accomplish this isolation on supporting architectures such as x86-32. On other architectures, CPI uses information hiding via randomization to protect the safe region. In addition to randomly locating the safe region, CPI also guarantees that there is no direct pointer into the safe region. This is done by hiding the base address of the safe region in a register (`%gs` or `%fs`), instrumenting the code to ensure there are no other accesses to that register, and then replacing the in-program pointers with indices relative to the hidden base address.

3.3   Attacks on CPI

We present an attack on CPI showing that on modern (x86-64 and ARM) architectures, the security guarantee of the randomized safe region is much lower than claimed in the original CPI paper [19]. Mitigations to the CPI approach are possible using a different, non-randomization based safe region, but the performance impact of these mitigations is unknown.

We make use of both fault and timing side-channel analysis techniques, explained below, separately in our attack on CPI. Fault analysis is used for a crashing attack and a novel application of timing analysis for a quieter but more time-intensive non-crashing attack.

3.3.1   Side-Channel Memory Attacks

Side-channel memory attacks are a new, indirect form of information disclosure. The closest previous use to our work is [41]. There are two principle types of side-channel disclosure: fault analysis and timing analysis. Both of these techniques can disclose the contents of arbitrary memory locations.

Fault analysis works by observing distinct program effects caused by an abnormal input—for instance, a program crash. A domain-relevant example of fault analysis is the Blind ROP attack [10], which strategically uses crashes to leak information about program memory.

Timing analysis, on the other hand, creates a distinct input and then measures the time taken for the processing of that input. As an example of a timing analysis attack, consider the pseudocode snippet in Figure 3-1. Here a pointer is dereferenced at an offset and used as the upper bound for a loop which performs an increment operation. The runtime of this loop is directly proportional to the value at the dereferenced memory location.

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

Figure 3-1: Example of Data Pointer Timing Side Channel

Although it is slower than fault analysis, there are a number of advantages to using timing analysis for memory disclosure. It can read locations that are zero without a crash, unlike most traditional memory disclosures. Additionally, memory locations with contents equal to zero, which are statistically more common in many environments, can be read faster than other locations. We make use of this fact in our non-crashing CPI attack.

3.3.2   CPI: Timing Attack Example

To validate our idea that CPI could be attacked with timing channels, we built a proof of concept timing attack against a CPI-protected version of Nginx. We assume only the existence of a data pointer overwrite vulnerability that affects control flow.

Vulnerability

This data pointer overwrite can be accomplished by a heap or stack overflow, depending on where the original data pointer was located. For our purposes in the Nginx

analysis, we used a stack overflow similar to CVE-2013-2028 [4] to overwrite a data pointer.

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

Figure 3-2: Nginx Loop Pointer Overwritten in `nginx_http_parse`

We overflowed a data pointer in a core Nginx component that performs string processing. The value at the data pointer we control now becomes the upper bound of the string processing loop, meaning that the overall time to process a request has a dependency on an arbitrary memory location. Specifically, the loop we used is in the source code of `nginx_http_parse.c`, shown in Figure 3-2.

Importantly, our vulnerability exploits the fact that the value dereferenced by the pointer is a `char` type corresponding to a single byte, rather than an integer or other data type; this means that there are only 256 possible values that the remote timing attack needs to discriminate between. In practice, we are able to require even lower resolution due to our knowledge of the layout of the CPI safe region.

Timing Side Channel

Next for our proof of concept, we measured the round-trip time (RTT) of a simple HTTP request that exploited this vulnerability. Several dozen samples are collected for each memory location. We then build a model that creates linear function mapping round trip times to byte values between 0 and 255. Using this, we can read the contents of arbitrary bytes in memory with some degree of certainty, depending on the amount of jitter in the RTT of the network, the time we are willing to spend reading samples on each byte, and the overall bandwidth of the network and Nginx. We later found that our knowledge of the safe region's layout and structure was consistent enough that it is sufficient to simply divine zero from non-zero bytes; the algorithm presented uses this simplification.

The algorithm used to create this model mapping remote addresses to "zero" or "non-zero" from the timing channel is as follows:

- **Parameters**:

  - `nsamples` The number of samples to take for each remote address. Each sample is a measurement of the round-trip time (RTT) for a request that references a specific remote address.

  - `max_calibration_rounds` The maximum number of calibration rounds that will be performed before preceeding to the scan.

  - `positive_calibrations` The number of addresses corresponding to non-zero memory to scan during calibration.

  - `negative_calibrations` The number of addresses corresponding to zero memory to scan during calibration.

- **Calibration**:

  While the current calibration round number is less than `max_calibration_rounds`:

  1. Repeat a network request `nsamples` times and record the RTT for each address in a parameterized range of known addresses with non-zero contents.

  2. Repeat a network request `nsamples` times and record the RTT for each address in a parameterized range of known addresses with zero contents.

  3. Flatten the results of the previous steps into two contiguous range of samples, one corresponding to non-zero memory (`positives`) and the other to zero memory (`negatives`).

  4. For every percentile between 0 and 100, filter all out all data in `positives` and `negatives` below that percentile range. Then compute `hi`, the mean of the positive filtered scan results; also compute `lo`, the mean of the negative filtered scan results. Use the mean of these two values as the threshold to classify all of the measurements into positive (non-zero) and negative (zero) pages. If the number of true positives added to the number of true negatives is higher than that of any previous percentile, set the global best result to the `hi`, `lo` values used in this iteration.

5. If the number of false positives is 0 and the number of false negatives is also 0, terminate the calibration. Calibration will also terminate after `max_calibration_rounds`, as mentioned. The speed of calibration is dependent on network conditions, most importantly the jitter observed on the network.

- **Scanning**:

    We now continue at a much higher speed to scan the target addresses for the attack. For each address, we take `nsamples` measurements, filter by the best percentile found during calibration, and use the threshold compute by averaging the best `hi` and `lo` from calibration. If our analysis shows a positive result, we will re-perform a calibration sequence and scan again before reporting success and terminating the scan. This avoids spurious success reports caused by drift in the network conditions.

Timing Side Channel In Practice

The overall speed achievable by this algorithm is dependent on the time difference between the maximum and minimum time added to the remote request by the memory value at the attacker-controlled dereference. If the dereference is byte-specific, as in our case, this is the difference between in time between 0 and 255 loop iterations. This time delta forms what is effectively the signal-to-noise ratio of the timing side channel.

We performed an extensive analysis of the effect on this ratio on the viability of remotely using the side channel, because we initially had difficulty reproducing similar previous results such as [41]. To diagnose our issue, we reconstructed the exploit used in [41]. The loop used in the Apache web server takes about 16ms to run 255 times as seen in graphs presented in their paper. We directly instrumented Apache to find the time taken in one iteration of the loop, and found it to be 0.4ms. This is within experimental error the 16ms figure when multiplied by 255.

Next, we tested the loop time for the Nginx exploit we developed. We found that

our Nginx loop was only 1 to $3\mu s$ per iteration. This is notable because the loop employed in our attack had two orders of magnitude less signal than those previously tested in the literature.

Table 3.1: Samples required to discriminate zero from non-zero memory as a function of timing side-channel loop runtime in a wired LAN

Time for 255 Iterations	Samples Per Byte	False Positives	False Negatives
$16\mu s$	255 samples	0 / 200	0 / 10
$14\mu s$	337 samples	0 / 200	0 / 10
$12\mu s$	150 samples	20 / 200	0 / 10
$10\mu s$	505 samples	no data	no data
$8\mu s$	150 samples	no data	no data
$6\mu s$	1702 samples	5 / 200	0 / 10
$4\mu s$	757 samples	41 / 200	0 / 10
$2\mu s$	no convergence	n/a	n/a

We experimented on a quiet wired LAN to discover just how low the loop iteration could go before we were unable to reliably divine between zero and non-zero pages. Table 3.1 shows various values tested for the overall runtime of 255 iterations of the loop. The validation used a set of 200 zero pages for the negative tests and 10 pages in the C standard library (libc) for the positive test. Calibration ran before the validation set until 30 zero pages (`negative_calibrations`) and 10 libc pages (`positive_calibrations`) could be classified correctly. The $8\mu s$ case was clearly anomalous; we attribute this to a fortunate lack of drift in the network after the $10\mu s$ test. For the $2\mu s$ case, there was no convergence after 5743 samples and the test was terminated.

The closest other result discussing the performance of timing side channels as a function of the loop iteration value was a whitepaper by Matasano Security Research [15], which we also credit for some of the ideas used in our proceedural setup to ensure higher accuracy.

Before these experiments started, our setup procedure checked the following environmental conditions:

- The libc checksum is `7b6bbcea6627deace906d80edaefc631`, corresponding to `libc-2.19.so` on Ubuntu Linux.

- Frequency scaling on all CPU cores is disabled.

- The X windowing system is disabled—i.e. the attacker system is running without a graphical interface.

- Nginx logging is off to reduce noise from file system writes.

- The machine was booted with an isolated CPU, using the following argument to the GRUB bootloader:

```
GRUB_CMDLINE_LINUX_DEFAULT="maxcpus=2 isolcpus=1"
```

A further optimization suggested by [15] is to pin the attacker process to a single core; however this was not accomplished in our setup due to technical issues. Another suggestion was to set the ACPI settings to avoid lower c- and p-states [44]; this was not possible on our hardware.

In conclusion, in our experiments on a quiet LAN with an average RTT of 3.2ms, `max_calibration_rounds` could be safely set to 100 with `nsamples` set to 100, `positive_calibrations` equal to 10 and `negative_calibrations` equal to 30.

3.3.3   CPI: Non-Crashing Attack

In Figure 3-3, we see the layout of process memory on an x86-64 CPI-protected application. The stack, at the top of memory and growing downwards to lower addresses, is followed by the "stack gap," an unallocated region of memory. The next address which might be allocated is the maximum base address for the memory-mapped regions often referred to as the "mmap" region due to the corresponding Linux system call name. The linked libraries of an application will be mapped at a random page-aligned address somewhere between the minimum and maximum `mmap_base` addresses.

The distance between the max and min `mmap_base` is the amount of entropy provide by the operating system's ASLR implementation multiplied by the system page size. On x86-64 there are 2^{28} possible start addresses and a default page size
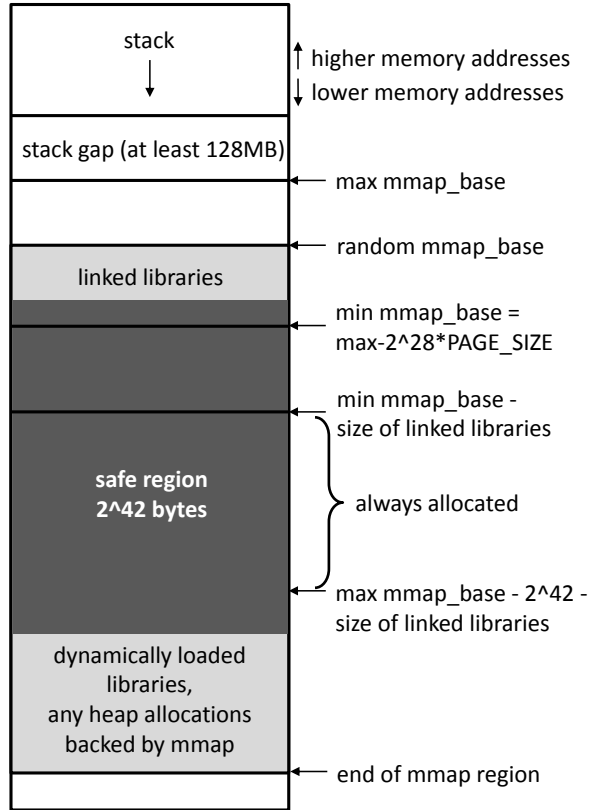
Figure 3-3: Safe Region Memory Layout

of 4KB. The safe region, with its default size of 2^{42} bytes, is larger than the entire space available for ASLR and thus will always produce a region directly after min `mmap_base` which is safe to de-allocate.

In other words, the CPI safe region in the default published implementation is so large it dwarfs the entropy available for ASLR on 64-bit systems. We exploit this fact to produce a non-crashing attack. Starting from the safe dereference location, we use a side-channel timing analysis to scan page by page until we reach the point where the C standard library (libc) lies next to the beginning of the next location which will be used by the linux `mmap` system call. This non-crashing scan approach is illustrated in Figure 3-4.

This attack works quite well, but takes a significant amount of time. Recognizing that ASLR does not change the order in which libraries are loaded, we can optimize this attack to increment the scan location by the size of the largest contiguous last-

Non-crashing scan strategy

Crashing scan strategy
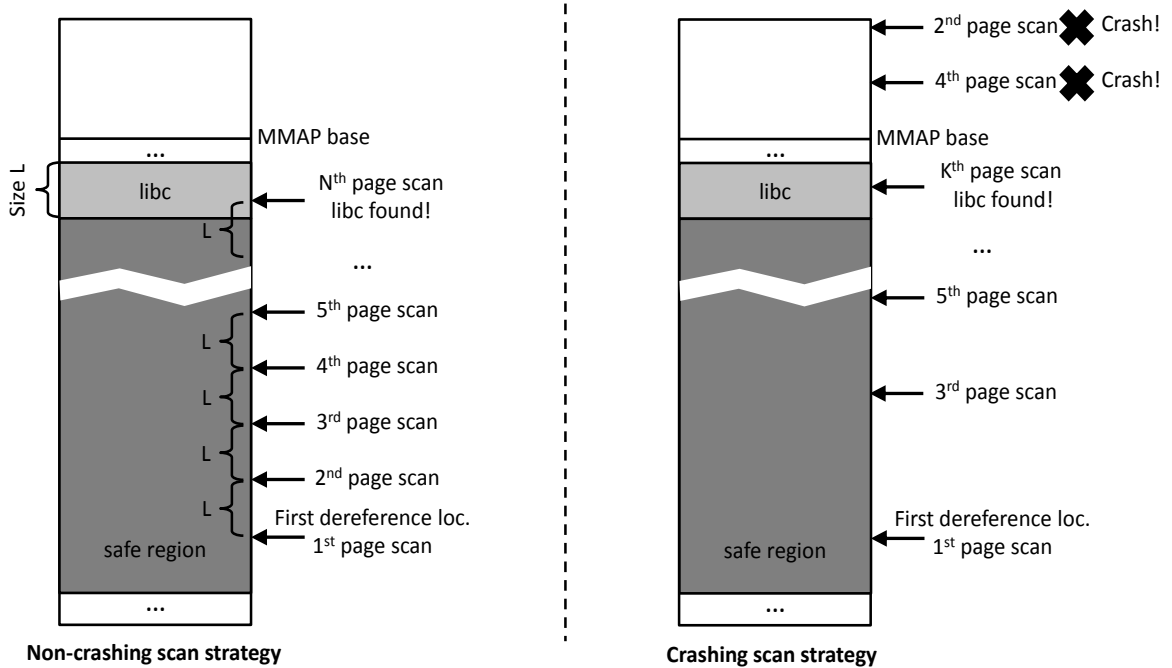
Figure 3-4: Non-Crashing and Crashing Scan Strategies

mapped libc segment (on Linux, the on-disk libc is not loaded in a continuous memory range; there is a large unmapped gap in the middle). With this optimization, we gain many orders of magnitude in speed.

The exact number of samples required to find the safe region is a function of the timing delta produced by the data pointer controlled for the timing side channel. In our tests only 30 samples per byte were required (recall that our side channel was $1 - 3\mu s$ *per* iteration). We scanned 7 bytes per region of libc—which in our experiments was enough to ensure 100% accuracy. The maximum number of libc-sized scans that might be needed from the known-safe dereference point to the base of the libc library in the `mmap` region is then $(aslr\_entropy * page\_size)/libc\_size$, which is $2^{28} * 2^{12}/2^{21} = 2^{19}$. The total of samples required for the attack is thus $7 * 2^{19} * 30 = 110100480$. At the previously-mentioned 3.2ms RTT on our test LAN, we can find the CPI safe region without crashes in 98 hours.

We mentioned earlier that it is sufficient to simply discriminate zero from non-zero bytes; now we can see why. As we scan towards the libc region, we are only concerned

with determining whether we are in a CPI page or a libc page. The CPI pages, due to the size of the safe region, will almost certainly be filled with zeroes. We pick offsets into libc that have byte values near the maximum value of 255 to maximize the difference between libc and CPI memory pages.

In fact, since the safe region is allocated contiguously, after all linked libraries are loaded, and the linked libraries are linked deterministically, the location of the safe region can be computed by discovering a known location in any of the linked libraries (e.g., the base of libc) and subtracting the size of the safe region (2^{42}) from the address of the linked library. A disclosure of any libc address or an address in another linked library trivially reveals the location of the safe region in the current CPI implementation. We consider this to be merely an implementation flaw in the current CPI release, however, and do not rely on it for our attack.

3.3.4   CPI: Crashing Attack

The released default implementation of CPI allocates a safe region with a size of 2^{42} bytes. Since the address space of modern 64-bit architectures is restricted to 48 bytes, a crashing attack can be easily constructed which guesses the location of the massive safe region and finds it with 2^5 crashes in expectation and 2^6 at most. Simply, this approach tries each of the possible slots for the safe region, waiting for the web server to re-spawn a worker process in the event of a crash. Once a slot has been accessed without a crash, we can employ the strategy from the non-crashing attack to quickly find the base offset of the safe region. Or we can continue the crashing attack to quickly find the edge of the safe region and unallocated memory.

However, the released version of CPI for x86-64 has 3 implementations:

1. **simpletable**: As mentioned, this is the default CPI implementation. In the simpletable, a massive table of 2^{42} is allocated. Every pointer in the target program is simply masked to strip the last three bits (since pointers on x86-64 systems must be 8-byte aligned) and then used directly as an index into the table. Performance overhead is minimal.

36

2. **hashtable**: In the hashtable implementation, a slightly smaller table is allocated and a every pointer is run through a hash function to produce the index into the hashtable. The hash function is a simple linear transformation and if the slot that the pointer hashes to is full, linear probing is used to find the next available slot. There is a fixed cap on the number of linear probes which will be performed.

3. **lookuptable**: The lookuptable allocates an initial directory table, and then dyanmically allocates subtable pages as needed. Thus every access will touch two pages in memory: the directory page for the address of the subpage, and then a subpage for actual value. This page directory system is similar to the lookups employed by a virtual memory manager.

We have already examined the simpletable implementation, so we next scrutinized the hashtable implementation. Due to its large size, it was vulnerable to the same crashing attack. Of course, the initial CPI implementation cannot be expected to be perfect. Thus we examined the minimal required size of the safe region by running CPI across the SPEC benchmark suite and decreasing the safe region size by powers of 2 until the benchmark failed. We made sure to remove the arbitrary maximum linear probing setting in the hashtable implementation.

The crashing attack is capable of exploiting each of these implementations; simpletable is the easiest and lookuptable is the most difficult.

3.4 CPI Exploitation

The two attacks shown have focused only on finding the base address of the safe region. However, once we have found the base address of the safe region, we can compute the address of any code pointer in the target application by masking with `cpi_addr_mask` (0x00ffffffffff8) and then multiplying by 4, the size of the table entry. From here it is trivial to modify a safe region pointer and build a ROP chain or target a library function leading to full remote code execution.

3.5   Countermeasures

We postulated a number of potential countermeasures to these attacks, which we also discussed with the CPI authors. The safe region of CPI can in theory be made secure from our attacks; the principle question whether this can be done without compromising CPI's benchmark performance.

1. **Runtime Rerandomization**: This attack currently relies on the fact that Nginx employs a master process which forks to create worker processes with the same memory layout. A modified architecture could cause the safe region to be reallocated in a new random location during each creation of a child process. Aside from the large memory overhead, this technique does not substantially improve the security guarantees of CPI against an attacker; the number of crashes required increases only slightly.

   Other more extreme rerandomization techniques could be used; for instance, the safe region location could be periodically shifted even without crashes. The performance impacts of such a design are unclear, and in general the probabalistic guarantees of a rerandomizing CPI are significantly weaker than those of full memory safety.

2. **Randomize Safe Region Location**: Instead of relying on ASLR randomization, which provides only 28 bits of entropy on x86-64 Linux systems, CPI could specify a fixed random address using the `mmap_fixed` argument for its `mmap` allocations backing the safe region. This makes the safe region non-contiguous with the rest of the linked libraries, eliminating the possibility of the "always safe" address that we make use of in our non-crashing attack. Unfortunately, there are a number of problems with this approach.

   - The `mmap` man page states that "the availability of a specific address range cannot be guaranteed, in general." The CPI authors might have to write platform-dependent code to ensure that the random region they have se-

38

lected is available and that they are not interfering with other ASLR techniques, both of which might weaken the security of this approach.

- If an attacker can cause heap allocations above a certain size, the heap allocations will be backed by `mmap`; if the attacker can leak these addresses, the address of the safe region can be inferred by noting where the heap allocations (presumably contiguous) skip a large range of virtual addresses, implying the presence of he safe region.

3. **Keyed Hash Function for Safe Region**: Another potential fix to CPI is to use the segment registers, which currently hide the base address of the safe region, to instead hide a key for a hash function into the safe region. The performance impact of this approach is unknown. Such an approach could still be vulnerable to attack as a fast hash function will not be cryptographically secure and an attacker can still read entries from the safe region, which may allow the eventual computation of the key being used by correlation with known safe region entries.

4. **Shrink Safe Region**: Our attacks has taken advantage of the safe region's massive size: does it really need to be that large, or was this just an oversight on the part of the CPI authors? What would the performance impacts of a smaller safe region be?

   We ran a series of tests using the C and C++ SPECint and SPECfp 2006 benchmarks [24] with a range of CPI hashtable sizes. These tests were run on an a machine with 4GB of DRAM running Ubuntu 14.04.1. We found that once the size of the hashtable implementation dropped to 2^{26}, two of the SPECint and two of the SPECfp benchmarks ran out of space. Note that we removed the linear probing limit arbitrarily imposed in the default CPI implementation.

5. **Software Fault Isolation**: The CPI paper references the use of Software Fault Isolation (SFI) for other architectures where information hiding is not feasible. There are also implementations for the now-obselete x86-32 architecture which

use hardware segmentation support to isolate the safe region. These are not vulnerable to our attack. However, unfortunately for CPI, Intel removed the hardware support for segmentation in the transition to x86-64, and it is also not available on ARM64 architectures.

Software fault isolation may indeed prevent our attack, however the performance overhead of such a technique is unknown and may be substantial. This would not be significant to an evaluation of CPI's security except that CPI's novel feature is its performance. If the performant scheme is not secure, the central contribution of the CPI technique is lost. By showing that the versions of CPI which the performance metrics are measured against can be bypassed, we place the burden of proof on the CPI authors to show a viable secure and performant implementation.

Chapter 4

Effectiveness of Ideal Control Flow Integrity

4.1 General Limits of Control Flow Integrity

4.1.1 Background

We next set out to characterize the limits of an "ideal" CFI scheme. Most modern CFI schemes are limited by practical considerations, as precise enforcement of CFI introduces substantial overhead [5, 7]. A CFI scheme with even only 50% overhead is typically considered unacceptable in industry, and thus many CFI schemes have compromised by providing a coarse-grained protection which has lower performance impact—but also lower security guarantees. However, these coarse-grained implementations have all been shown to have enough flexibility to allow an attacker to achieve Turing-complete computation leading to exploitation [23].

Given this fact, attention has turned to building "fine-grained" CFI schemes which can still meet the tight performance requirements needed for a generalized solution. Google's forward-edge CFI [45] enforces fine-grained policies on forward-edges, but ignores return edges. Cryptographically enforced CFI [31] uses cryptographic message authentication codes with a secret key hidden in a dedicated register to verify the authenticity of pointers and prevent transfers with injected addresses.

None of these techniques are perfect, but we decided to investigate a more fundamental question: taking the strongest CFI technique we could imagine—regardless of the performance costs–might there be ways to attack it?

We decided to examine an idealized, hypothetical CFI design to characterize the maximal security guarantees of CFI. Since we are unconcerned about the runtime performance of the CFI policy, we can allow a full shadow stack (to prevent backward-edge attacks) and an unlimited number of tags for function calls (to prevent forward-edge attacks). Although previous work exists such as Out of Control [23], which showed that gadgets enabling RCE can be found in a two-tag system created out of the pairings of all valid call and entry points, no work has yet examined such an idealized CFI design.

Assuming this theoretical idealized CFI design, we attempted to discover whether control-flow attacks were still possible. We found that they are possible, and developed a new attack that we termed *Control Jujutsu* [20], which exploits the incompleteness of scalable pointer analysis and the patterns used in software engineering best practices to enable an attacker to exploit sufficiently large programs and cause arbitrary malicious code execution—even in the presence of our ideal CFI design.

4.1.2   Sound and Complete Pointer Analysis

For simple programs without complex modularity or functional composition, the control flow graph can be statically and precisely determined. However, the real-world programs that CFI needs to work on are anything but simple.

The forward-edge protection of a CFI scheme is only as strong as the static analysis used to inform the model of the program CFG. Most CFI papers focus their attention on describing how to create CFI schemes that are efficient enough to be practically useful, assuming the static analysis will produce an accurate CFG that can be enforced on the program without (1) producing false positives and (2) without allowing any program exploitation.

To produce such a CFG, we would ideally use a sound and complete points-to program analysis algorithm. Unfortunately, it has been shown that sound and complete

points-to analysis is undecideable in general [38]. Thus to perform general points-to analysis, one must compromise either soundness or completeness. An incomplete analysis will produce extra edges in the CFG that might allow an attacker to slip through. An unsound analysis will under-approximate the true program CFG, which would break program functionality by producing false positives—clearly undesirable for a CFI technique. Thus for CFI schemes typically an incomplete but sound scheme is used. This works well for program analysis, but is insufficient to protect against an creative adversary attempting to exploit a targeted program. We show that state-of-the-art static analysis techniques leave enough flexibility in the CFG for an attacker to achieve arbitrary code execution. We use the best available such scheme, the state-of-the-art DSA pointer analysis algorithm [29], in our evaluation.

4.2   Attacking Ideal CFI

Specifically, we analyzed two large, popular open-source web servers (Apache httpd and Nginx) and find that the CFG constructed by static analysis is large enough that we can still achieve code exploitation.

To accomplish this attack, we find pairs of Indirect Call Sites and functions that provide Remote Code Execution (RCE)—existing program functionality designed to extend the program or make external commands which can be used as a basis for arbitrary remote code execution. We use these pairs as a kind of "gadget" and call them Argument Corruptible Indirect Call Site (ACICS). Note that in our case we show the more difficult task of *remote* code execution, which imposes an additional requirement that the exploit we show be triggerable by external input.

We use ACICS gadgets to (1) enable data (argument) corruption of an indirect call site that in conjunction with the corruption of a forward edge pointer (2) direct execution to a target function that when executed provides arbitrary code execution (e.g., system calls). We show that for modern, well-engineered applications, ACICS gadgets are readily available as part of the intended control transfer.

4.2.1 ACICS Discovery Tool

We developed a tool we termed the ACICS Discovery Tool (ADT) to find these ACICS and characterize their exploitability. ADT dynamically instruments applications using the GDB 7.0+ reverse debugging framework. It is implemented as a Python extension to GDB itself. The algorithm for the tool is as follows, also illustrated in Figure 4-1.

Figure 4-1: ACICS Discovery Tool
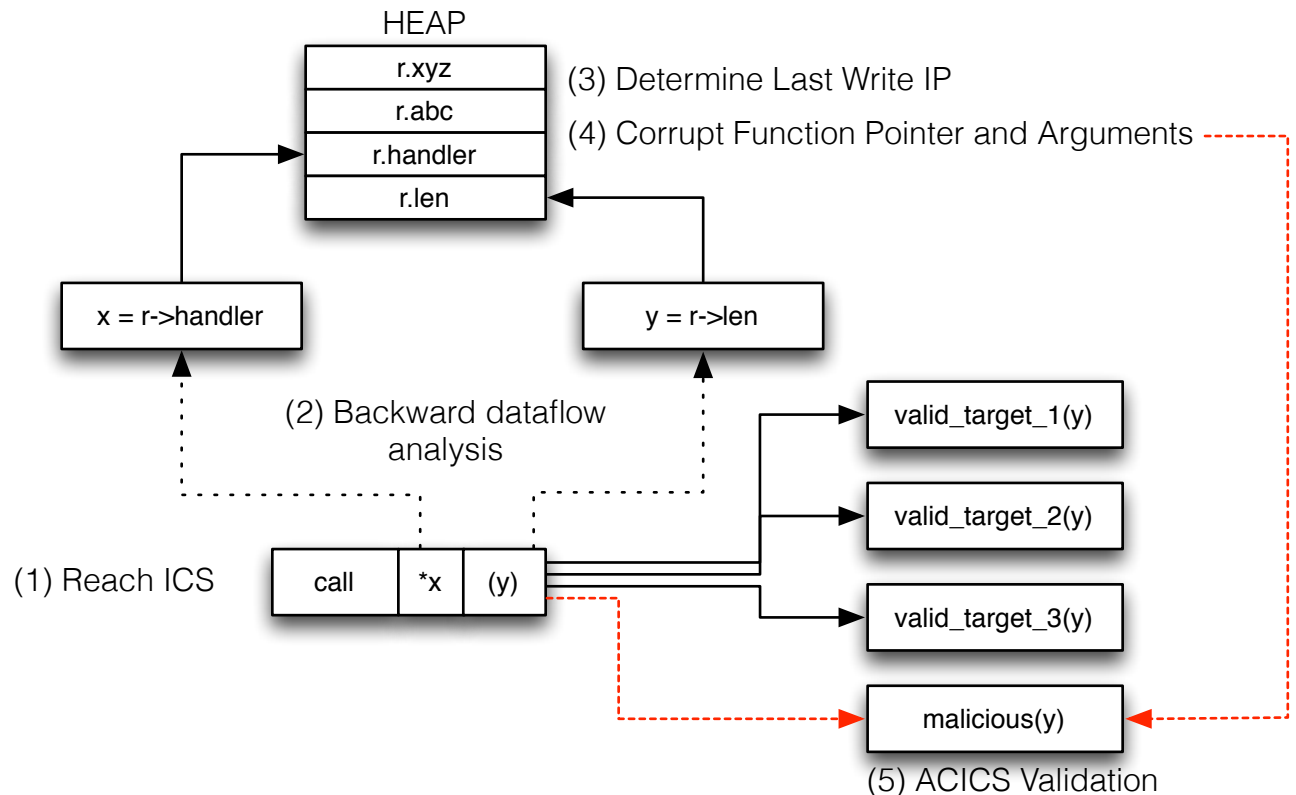
Algorithm For each ICS identified in the target program our ACICS detection tool performs the following procedure:

1. Read the ICS breakpoint given as input, `target_ics`

   (a) Set a breakpoint at `target_ics`

   (b) Set a special breakpoint at the entry to the function containing the `target_ics`. This is a "reverse execution" breakpoint provided by our plugin. When hit,

this breakpoint will enable GDB's process recording functionality, allowing us to run process execution in reverse if we hit the ICS breakpoint.

2. Run the program with the debugger:

   (a) Start the target program

   (b) Wait until the target program has finished starting, then run a suite of exercise programs.

   (c) Start a program that will interrupt GDB if after a fixed timeout is reached (useful for cases the ICS is not exercised)

3. This step corresponds to Figure 4-1-1. If the ICS breakpoint is not hit: report a failure to exercise the ICS and end evaluation of this ICS. If the ICS breakpoint is hit: perform the backward dataflow procedure (see Section 4.2.2) on the ICS target register and the x86-64 argument-passing registers `rdi, rsi, rdx, rcx, r8, r9`.

4. This step corresponds to Figure 4-1-2. For each pointer identified as the source of the backward dataflow procedure:

   (a) Find the memory region in the Linux process memory map (`/proc/pid/maps`) that the pointer corresponds to. Report the categorization of the pointer as stack, global, heap, or note an invalid pointer (typically implying a backward dataflow failure).

   (b) Reset the debugger and repeat Steps 1–2 after adding a special hardware watchpoint on the pointer and continue. This hardware watchpoint is provided by our plugin and will record the number of number of writes to the pointer from each instruction pointer. Once we reach the target ICS breakpoint, stop and report the last instruction pointer and number of writes. This step corresponds to Figure 4-1-3.

   (c) Reset the debugger and repeat Step 4b until a breakpoint on the last instruction pointer is observed. Continue until the number of hits on the

breakpoint is equal to the number of writes from Step 4b. Now we know that the pointer is live until it is used at `target_ics`, and this ICS is officially an ACICS. Therefore it is safe to corrupt the pointer. Additionally, if we are willing to track every statement executed between now and the `target_ics`, a lower bound of the liveness of the ACICS can be established.

Now we corrupt the pointer; if the pointer is the function pointer, we change it to a malicious target function; if it was an argument, we can set it any valid memory location containing a unique value that we will verify when we arrive at the malicious function. If our dataflow algorithm failed, sometimes the pointer will not be valid and the memory cannot be written; report a failure in this case. This step corresponds to Figure 4-1-4.

5. Set a breakpoint at the malicious function and let the program continue execution. If we arrive at the malicious function, we will note that control flow was succesfully redirected for this `target_ics`.

6. Testing if the arguments to the malicious function match the expected values set in Step 4c.

7. Finally, as a sanity check we ensure that the malicious function was executed by checking some external state (a remote webserver receeived a request from out hijacked application, a file was created, etc).

4.2.2   Reverse Dataflow

Figure 4-2 is a psuedocode representation of our simple backward dataflow algorithm. The goal of this algorithm, which operates within the debugger, is to find the memory address from which the forward edge pointer originated. For example, in Figure 4-1, the dataflow algorithm called on input x would produce the address of `r.handler`. This is done by iteratively stepping back in time (reverse debugging) and examining any instruction that modifies the register which originally held our function pointer. We assume that the instructions involved in the dataflow of the target function can

Input : The target ICS instruction *icsinst*.
Input : *Prev*, a function that returns the previous instruction (or NULL if not available) before a given instruction.
Output: The memory address that stores the call target or NULL if failed

1 **if** *icsinst is the form of "`call` REG[i]"* **then**
2     $r \longleftarrow i$
3 **else**
4   **return** NULL

5 *inst* $\longleftarrow$ *Prev(icsinst)*
6 **while** *inst* $\neq$ *NULL* **do**
7   **if** *inst modifies* REG[r] **then**
8   **if** *inst is the form of "*REG[r] = a * REG[i] + c*"* **then**
9           $r \longleftarrow i$
10   **if** *inst is the form of "*REG[r] = *(a * REG[i] + c)*"* **then**
11   **return** $a \times regv(i) + b$
12   *inst* $\longleftarrow$ *Prev(inst)*
13 **return** NULL

Figure 4-2: Backward dataflow analysis to identify the target address

be represented as the composition of linear functions and dereferences, and report a dataflow error if this does not hold. Once we have found a function which dereferences a memory location, we use a linear function modeling that instruction to compute the source address of the forward edge.

The actual GDB plugin code adds many additional checks, such as an assertion to ensure that the forward edge pointer value at the ICS matches the value observed at the computed source memory address which is the output of the backward dataflow procedure. The typical use case that our algorithm find is the lookup of a member element from a struct pointer; such as `x->y`; more levels of indirection such as `x->y->z` are not currently covered.

In general, however, a failure in the dataflow will be obvious by our algorithm failing to hit the target function in Step 5 of the overall algorithm.

4.2.3 Discussion

The number of viable ACICS reported by this tool is a lower bound for several reasons. First, our simplistic dataflow analysis will report that ICS do not have corruptable arguments in the case of multiple levels of indirection. Second, the ACICS detection tool works best when the execution environment is deterministic. It will not report incorrect results in the case of non-determinism; it will simply underreport the number

of ACICS. Third, the number of viable ACICS might be expanded by examining unintentional arguments—pointers left in registers from previous function calls which might become relevant again if a function pointer were changed to point at a higher function of higher arity.

4.2.4 ICS Instrumentation Tool

We developed another tool, also using the GDB debugging framework, capable of setting hardware watchopints on all indirect call sites and then recording the pointers observed as forward edges at those call sites as well as the addresses of all arguments passed in calls. The tool assumes a fixed upper bound on the number of arguments, as it would require inspection of the source code to determine the exact number of arguments being used—a measure which would also be invalid for functions with variadic arguments.

This tool could also have been implemented as an LLVM plugin, but it was more convenient to retrofit the GDB framework already being used for ACICS discovery to accomplish the task. We used this tool to survey the ICS in the program and understand the locations and functions of the most frequently used call sites.

4.3 In-CFG Attack Examples

We next used our tools to develop an attack against an idealized CFI system in the Apache HTTPD web server.

4.3.1 Threat Model

We assume a remote attacker with knowledge of the software versions present on the target machine and knowledge of a memory corruption vulnerability that allows the attacker to corrupt values on the stack, heap, or data segments. On the defensive side, we assume an "ideal" CFI scheme as discussed earlier with unlimited tags for the forward edge an infinite shadow stack to protect return edges. Additionally, we assume

the existence of the write xor execute and ASLR exploit mitigation mechanisms.

4.3.2 ACICS in Apache

We found 172 indirect calls sites in the unoptimized, dynamically-linked Apache httpd binary. We limit our evaluation to the core binary, omitting potential ICS sites and targets in other Apache modules such as the Apache Portable Runtime (APR) library and the APR-util library.

Our requirements on the 172 ICS to be Argument Corruptible Indirect Call Sites are as follows:

- The ICS is exercised by external input into the program (e.g. a network request in Apache's case)

- The forward edge pointer and its arguments at the ICS must be visible during a code path exercised by an internal program input. Practically, this means that the forward edge pointer and its arguments should be on the heap or static globals so that they are visible at any execution point in the program.

- After we corrupt the ICS arguments in a path triggered by external input, the program should not crash or overwrite the arguments before reaching our target function.

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

Table 4.1: 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 4.2: Automatic Corruption Analysis

Table 4.1 shows test suite coverage of the ICS sites. Table 4.2 shows the results of our automated ACICS discovery tool running on the HTTPD binary.

We examined the location of the 172 ICS sites inside the Apache source to look for commonality and patterns. 108 of the sites are related to the Apache library's "hook" system. This is Apache's generic mechanism allowing the registration of function pointers in structures for later callbacks. Every function pointer in the hook system is stored inside a global struct named `_hooks`, which is live across the lifespan of the Apache worker process—an ideal quality for our attack. Furthermore, most of the points calling function pointers in the hook structure have arguments which are themselves visible across the lifecycle of an Apache web request—for instance the ubiquitous `request_rec * r` argument.

After identifying viable ACICS sites, we construct an exploit as a proof of concept. We choose the ACICS at `ap_run_dirwalk_stat`, which meets every requirement we have and is exercised in every HTTP GET request. Having picked the ACICS where we can control the target and the arguments without crashing the executable, we now choose an appropriate target function to redirect control flow to.

4.3.3 Target Functions in Apache

An ideal target function has minimal requirements on its control flow before it makes a powerful system call, such as `system` or `exec`, using one or more of its arguments. The principled way to find such target functions would be a backwards reaching dataflow analysis from each potentially dangerous system call to the entry point of the function containing it to identify constraints, and then an analysis of the control flow graph to identify functions calling it. We performed a simplified, heuristic version of this analysis to measure the distance between a function and a function containing this class of system calls. The results of this analysis are presented in Table 4.3. Previous work found similar results for the Windows platform [23].

For our Apache exploit, we use the `piped_log_spawn` function, which is two steps away in the callgraph from an `exec` call.

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

Table 4.3: Target Functions Count Based on CallGraph distance

4.4   Defenses

4.4.1   Static Analysis within Type System

One principled defense to our attack is a strict runtime type-checking system. There are a number of practical difficulties with building such a tool–principally the deviations from the C standard that are common in real-world C programs. However, if we assumed such a tool had implemented correctly and were not concerned with breaking program functionality, would it necessarily stop *Control Jujutsu* attacks?

We developed another tool to analyze the function signatures observed between large bodies of code. This tool involved a compiler shim which recorded the compiler arguments passed for each compilation unit during the invocation of an overall project build. Thus preprocessor directives and other flags which might influence the interpretation of a source file were properly respected.

With this information captured, we invoked Clang 3.6.0's parser API in libclang to parse the C and C++ files of an application and extract any function signatures or indirect call signatures, passing the appropriate arguments captured in the previous step as mentioned. Table 4.4 shows the results when we ran this analysis on Apache HTTPD 2.4.12, APR 1.5.1, BIND 9.10.2, vsftpd 3.0.2, and Nginx 1.7.11. These results include linked headers from standard and application libraries as their functions must be included in the list of valid targets.

Number of unique...	HTTPD+APR	BIND	vsftpd	Nginx
...function names	9158	10125	1421	2344
...function signatures	5307	5729	730	1135
...indirect call signatures	117	135	5	3
...aliased signatures	81	27	5	2
...aliased functions	553	328	68	119

Table 4.4: Matching ICS & Function Signatures

In Table 4.4, "unique aliased signatures" means the number of signatures which exactly correspond to at least one indirect call and at least one function declaration. "Unique aliased functions" refers to the total number of functions—as opposed to function signatures—which might be targets of these indirect calls sites. If an ACICS existed for each aliased function signature, the aliased functions count is the number of functions the ACICS could legitimately target while respecting the constraints of the runtime type-checker. Obviously many of these are intentional targets, but the number of unintentional aliases cannot be reliably determined without programmer annotations.

4.4.2 Connection to Programming Language Constructs

One might hope there are a few outlying conventions or behaviors that programmers could be instruction to avoid in order to produce programs with CFGs that are immune to this attack. Unfortunately, the emphasis in software engineering best practices on modularity and abstraction is often exactly what enables our attack to succeed.

```
1   struct apr_bucket_type_t {
2     const char *name;
3     int num_func;
4     void (*destroy)(void *data);
5     ...
6   };
7
8   struct apr_bucket {
9     const apr_bucket_type_t *type;
10    apr_size_t length;
11    apr_off_t start;
12    void *data;
13    void (*free)(void *e);
14    ...
15  };
```

Figure 4-3: `bucket_brigade` declarations in APR-util

Consider for example the Apache bucket brigade mechanism. Figure 4-3 shows the definition of the bucket brigade structure, part of Apache's custom memory manager. Macros such as `bucket_brigade_destroy` in the Apache utility library are called extensively by the Apache HTTPD source. When we examine the definition of this

```
1  # define apr_bucket_destroy(e)
2  do {
3    (e)->type->destroy((e)->data);
4    (e)->free(e);
5  } while (0)
```

Figure 4-4: `bucket_brigade_destroy` macro definition in APR-util

```
1  result = xfr->stream->methods->next(xfr->stream);
```

Figure 4-5: Example call from BIND xfrout.c

macro, shown in Figure 4-4, we find that it is making use of the bucket brigade structure in an object-oriented fashion, calling methods in the struct and passing other members of the struct as arguments. Clearly, this pattern is vulnerable to exploitation. If an attacker corrupts a bucket brigade structure, the static analysis cannot determine the limited set of function pointers that should be allowed. The situation is analogous to a virtual table pointer overwrite in C++, but without the support of the C++ type system to determine the allowed function pointer entries.

We found this pattern of extending C was common in complex programs. For instance, BIND has a very similar structure which is used extensively and contains a mix of code and data pointers, which the points-to analysis has little hope of restricting. An example call from BIND showing the C++-like use of this structure is shown in Figure 4-5.

Fine-grained CFI schemes which can make strong guarantees about typical C programs will fail to account for cases such as this and other higher-level patterns implementing, without language support, object-oriented techniques.

Clearly the principled solution to this problem is programmer annotations, The duality of code and data in modern programs, and languages such as LISP, explicitly listing the allowed targets for every function pointer reference, make this a difficult problem for CFI schemes to solve for programs in general.

Chapter 5

Tagged Hardware Architecture

5.1   Background

The second component of this work is an evaluation of the minimal set of tag bits and policies needed to provide a practical defense against code reuse attacks on supporting hardware. Our overall goal is to determine whether a set of minimal control-flow policies could be defined which prevent code reuse attacks. Once such a set of policies have been identified, could they be efficiently implemented via hardware support and then perhaps be turned into hardware extensions to modern architectures?

Specifically, we evaluated several tag encodings and policies on a RISC-V architecture which we modify to provide tag support. In parallel, we developed compiler infrastructure for the C and C++ languages to support the initial tag mappings.

If we implement this tagged memory-safety system correctly, we will have dramatically increased the difficulty required for an attacker to exploit a vulnerability: the attack surface is limited to temporal memory vulnerabilities (use-after-free), bad casting vulnerabilities [30], data-flow or logic vulnerabilities, and the integrity of tag files. Of course, attacks that exploit flaws in the hardware-level, such as the Row-Hammer attack on physical memory, [25] are not addressed by our approach.

5.2 RISC-V Extensions

There are several options–MIPS, x86, ARM–when considering an architecture to base our tagged extensions on. We chose the new RISC-V ISA, a novel, modern, fully open-source competitor to architectures such as ARM64. Our group in parallel with the Cambridge lowRISC project [12] has added tagged extensions to the new RISC-V ISA.

The lowRISC project's focus has thus far been on defining the hardware of a tag-extended architecture; our focus has been on defining policies for that hardware. Since our interest is in the policies and their interaction with modern software applications, we began by modifying Spike, the "golden" reference emulator for the RISC-V architecture to explore the impact of our modifications,

We add a register file in which each register has been extended with bits to store the tag. The L1 and L2 caches of the RISC-V architecture implementation are similarly extended. A cache miss will, in addition to reaching out to main memory, perform in parallel a lookup in a specialized tag cache. If the tag is not in this tag cache, there is an additional performance penalty in the form of a second access to main memory to fetch the tag.

We extend the RISC-V registers to support an arbitrary-width tag. We then modify all of the instructions available in the CPU to support passing this tag. For simple instructions such as `mov`, this support is easy. For other instructions, such as binary and unary arithmetic operations, the proper flow of the tag is policy-defined and the implementation is left in a policy-dependent file via macros.

We then continue the development work by building tag-supporting versions of the GCC and LLVM compilers provided by the RISC-V project. We add several new instructions to these compilers:

- `settag`: This sets a given memory location to have a particular tag value. Our mappings between tag names and their integer values are implemented as C preprocessor macros.

- `tagenforce`: This instructions turns the enforcement of tag policy (specifically,

tag propagation and policy-generated traps) on or off.

```
1   #ifdef __riscv
2     // Set the tag on the register that mem is located.
3     // Additionally, store this register so that when it is
4     // reloaded (which is likely to happen immediately)
5     // the tag is correct. -24(s0) is what is generated
6     // on the store, so we match it unintelligently here.
7     if (is_fptr) {
8       __asm__ __volatile__ (
9         "settag %0, 2\n\t"
10        : "=r"(mem)
11        : "r"(mem));
12    } else {
13      __asm__ __volatile__ (
14        "settag %0, 1\n\t"
15        : "=r"(mem)
16        : "r"(mem));
17    }
```

Figure 5-1: `settag` macro in tag_extensions.h

We implement the addition of these new instructions in GCC and then use wrapping them in macros using inline assembly, as shown in the excerpt in displayed in Figure 5-1.

We modified the other instructions in several broad categories:

- **move instructions**: Move instructions always propogate the tag to the destination; it is policy-dependent whether they also strip the tag from the source register or memory location.

- **binary arithmetic instructions**: Policy-dependent.

- **unary arithmetic instructions**: Policy-dependent.

- **floating point instructions**: Floating point instructions do not propagate tag information.

Having developed underlying infrastructure based on RISC-V, we continued by developing tag policies to prevent code reuse attacks.

5.3 RISC-V Tag Policies

5.3.1 Basic Return Pointer Policy

We begin with a policy that adds a return pointer tag to the return address stored before a call instruction executes. RISC-V does not have a call primitive, instead using the `jalr` instruction, so this instruction is the only one that needs to be modified in order to effect this change.

In this policy, return pointer tags are copied on move, store, and load, but not arithmetic operations. Whenever a return instruction is encountered, the return register is checked for the return tag. If the tag is not present, a trap is generated.

This policy was easy to implement and generated relatively few exceptions. It prevents an attacker from using traditional methods to forge return addresses. The cases where this policy ran into difficulty were virtually identical to the cases where traditional CFI work enforcing shadow stacks has difficult (see Section 2.3.

First, signal handling code generates a return address which is used to jump to trampoline code; we added a blessing `settag` instruction in the signal handling code, which is a slight but reasonable increase in the trusted code of the process. Even with this exception, our tag system is still not vulnerable to sigreturn-oriented programming [11] attacks.

A second difficulty is C++ exceptions. Our team manually patched the GCC library binary, `libgcc_s.so.1`, to add code to bless the return addresses generated by the exception handler. This is also a slight increase in the trusted code base.

This return pointer policy is basic but still quite strong compared to many other code reuse defenses discussed previously. It prevents all current ROP attacks and mechanisms because an attacker can never inject an address into the exeuctable which will be interpreted as a return pointer. However, an attacker can "clone return addresses, which may be enough to achieve arbitrary code execution. For instance, an attacker with control of the arguments to `memcpy` can overwrite return addresses on the stack by copying existing return addresses from the stack or main memory. The viability of this attack to actually create working exploits depends on the stack depth

at attack time and whether a program is saving portions of the stack in main memory. Thus RCE attacks are unlikely, but perhaps a more restrictive policy can provide even stronger security guarantees.

5.3.2   Linear Return Pointer Policy

We next implemented a "linear" return pointer policy. In this policy, there is only one usable copy of return pointer in the process memory at any given time. Loading and storing a return address from and to a register adds the return-pointer tag to the destination, but strips it from the source. Just as in the basic return pointer policy, to be used in a return instruction, the return register must have the correct tag. In contrast to the basic return pointer policy, the linear policy requires that arithmetic or logic binary or unary operations remove the tag on the source register.

This policy is quite powerful, providing all the guarantees of the basic return pointer policy but also addressing in a principled way prevention of the replay attacks previously mentioned.

In addition to the exceptions of the basic return pointer policy, there are a number of new exceptions which must be carefully accounted for when implementing these policies:

1. One exception to the removal of the destination register's tag is non-usespace code such as the `fork` system call. Since `fork` is expected to create a new process, it is acceptable for the kernel to make a copy of the register with the tags preserved for the new process. In general we allow non-userspace code to ignore the linearity restriction of this policy.

2. The C standard library's error-handling system, the `setjmp` and `longjmp` functions, require careful handling. The `setjmp` function saves the current stack, including return addresses, into a buffer, which `longjmp` can then restore. However, the restore from `longjmp` cannot strip the tag from the buffer, as `longjmp` may be called again. A principled solution to solve this problem is not possible without modifications to the C standard library and refactoring of applications

using these two functions. However, we weaken our security guarantees slightly by building special versions of `setjmp` and `longjmp` which ignore the linearity policy and instead applies the basic return pointer policy. Our security guarantees are weakened somewhat in that if an attacker gains control of the instruction pointer while a `setjmp`-created buffer is live on the stack, the attacker may be able to control the arguments from `setjmp` and `longjmp` and construct a replay attack. However, it seems unlikely that most programs are vulnerable to this sort of attack in general, and the attack surface is greatly reduced.

5.3.3 Function Pointer Policy

We also designed a function pointer policy, which is described at length in Chapter 6.

5.4 RISC-V Policy Test Framework

We developed a powerful, full-featured test framework to evaluate the impact of our policies on test scripts, small programs, and full applications.

To setup our test environment, we first compiled a version of Linux that could run in the Spike emulator. We installed the busybox [46] userspace on top of the Linux kernel, a minimal userspace that provides a shell and some common GNU utilities with few extraneous utilities. We wrote a script, `run_in_spike_linux.sh`, which wrapped all of this functionality to act as an interface to our testing rig.

The testing environment is capable of characterizing the output of given program across various policies enforced inside the Spike emulator. The testing tool compiles the program, copies the program into the Spike emulator, and then boots the Spike emulator running our Linux busybox setup and runs it. The full output and return code are recorded and compared to the expected test output. A matrix, shown in Figure 5-2, is produced, which shows whether the program output matched the expected test output for each policy defined in the testing rig.

2f3f10d		
suite	**policies**	**results**
need_linux	SPIKE=spike ../linux/run_in_spike_linux.sh	0/3
need_linux	SPIKE=spike-no-return-copy pk ../linux/run_in_spike_linux.sh	0/3
single-file-tests	spike pk	13/16
single-file-tests	spike-no-return-copy pk	13/16

suite	**filename**	**policies**	**result**
need_linux	test/need_linux/bin/riscv/forktree	SPIKE=spike-no-return-copy pk ../linux/run_in_spike_linux.sh	fail
need_linux	test/need_linux/bin/riscv/forktree	SPIKE=spike ../linux/run_in_spike_linux.sh	fail
need_linux	test/need_linux/bin/riscv/tagon	SPIKE=spike-no-return-copy pk ../linux/run_in_spike_linux.sh	fail
need_linux	test/need_linux/bin/riscv/tagon	SPIKE=spike ../linux/run_in_spike_linux.sh	fail
need_linux	test/need_linux/bin/riscv/time	SPIKE=spike-no-return-copy pk ../linux/run_in_spike_linux.sh	fail
need_linux	test/need_linux/bin/riscv/time	SPIKE=spike ../linux/run_in_spike_linux.sh	fail
single-file-tests	test/single-file-tests/bin/riscv/good/bss	spike-no-return-copy pk	fail
single-file-tests	test/single-file-tests/bin/riscv/good/bss	spike pk	fail
single-file-tests	test/single-file-tests/bin/riscv/good/cat	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/cat	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/divzero	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/divzero	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/echo	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/echo	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/helloworld	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/helloworld	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/insertionsort	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/insertionsort	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/num	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/num	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/touch	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/good/touch	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/rop-defender-tests/setjmp	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/rop-defender-tests/setjmp	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/rop-defender-tests/setjmp2	spike-no-return-copy pk	ok
single-file-tests	test/single-file-tests/bin/riscv/rop-defender-tests/setjmp2	spike pk	ok
single-file-tests	test/single-file-tests/bin/riscv/rop-defender-tests/setjmp3	spike-no-return-copy pk	fail
single-file-tests	test/single-file-tests/bin/riscv/rop-defender-tests/setjmp3	spike pk	fail

Figure 5-2: RISCV Policy Evaluation Test Matrix Excerpt

This framework was key in identifying compatibility issues and then verifying that we had successfully fixed them through the use of targeted test-cases.

Chapter 6

Compiler Support

6.1 LLVM Design Decisions

To support the function pointer policy, we add passes to LLVM project that track the flow of any address-taken values of any function, instrumenting them so that the introduction of a constant corresponding to a function address will be immediately tagged as a code pointer. We add support to the hardware policy engine to enforce policies on these code pointers and instructions to create them. Our tag-supporting processor will pass on the function pointer tag to any any static, global, or local variable which the function pointer might be copied to.

This approach differs from other recent code-pointer protection techniques for multiple reasons. First, it is implemented in hardware. Second, it is seeded with a dataflow analysis produced by the compiler. The compiler knows the provenance of any constant in the source code; our analysis finds all constants with provenance including a function address or a block address, which is the address of a basic block used as a interproceedural jump. This is in contrast to schemes such as Code Pointer Integrity which must rely on the C type system to identify protected function pointers.

6.2 Compiler Support Component: Find All Function Addresses

Function and block addresses can be encoded the following locations:

1. Emitted as immediates of the program text, corresponding to values which will eventually be used as call or jump targets.

2. As values in the symbol table section of the ELF executable.

3. It is possible for function pointers to be emitted in the `rodata` or `data` segments of an ELF executable, although our test programs with the LLVM project's Clang compiler did not observe this in practice.

4. Dynamically generated by functions such as `dlopen` and `dlsym`, which convert names of functions in foreign modules to the addresses of those functions so they can be resolved and called at run-time.

Our strategy for handling these cases is as follows:

1. The first component of the compiler suport is an LLVM pass over every function in the module (a module is a compilation unit in LLVM parlance). For each function, we find the `LLVM::Users` of the function address. For each `User`, we append it to a list in the hash table with the key as the function that the use took place in. Once this table is complete, we go through every function key in the table. For every item in the list, we create a new instruction using the function pointer at the early possible point in the function. We mark this instruction as volatile so that it won't be optimized out, as it has no side effects. Immediately after this instruction we add a call to `settag`.

   An exception is made if the use is direct, e.g. a LLVM `_call_` instruction using a function pointer will translate into an direct call using an immediate, and we do not need to add a blessing `settag`.

One item of future work for this pass is properly handling inline asm code, which may be a problem in applications which use hand-coded assembly.

2. The second component of the compiler support has not been fully implemented; what we describe here is our design.

   For every entry generated in the procedure linkage table (PLT), we imagine adding a `settag` instruction. Every entry in the global offset table (GOT) should receieve a special function pointer tag which decays after one arithmetic operation; one arithmetic operation; i.e. after the GOT entry is added to a base address, the function pointer tag persists but is no longer permitted to propagate through arithmetic instructions.

   This could pose compatibility problems if a program expects an array of function pointers in the GOT and iterates over one, adding an different offsets directly to the pointer value multiple times.

   The system loader will then modified to support reading these symbols and marking them with a code pointer type. One implementation of the tag file could be to reuse the DWARF debugging symbols typically generated for a ELF executable, and rename symbols which are used as code pointers with a prefix indicating that they should receive that tag. The loader would then scan for symbols matching the agreed upon pattern, and as they are loaded into the `.bss` or `.static` segments use machine-specific instructions to give them the specified tag. An obvious disadvantage of this approach is that it requires promoting every static or global variable to an externally visible symbol.

6.3   Alternative Compiler Designs

We considered, and rejected, several alternatives to this design, which we describe here to contrast our approach with related work:

- **CPI-Style Static Analysis**: We might instrument all types in the program and automatically tag data that flows into variables with those tpyes. This is not

ideal however, as we know we only want to bless the value at first initialization. Any static or global variable (such as a `void *`) which the compiler gave the designation "code pointer" should be registered only once.

- **Replace Every Variable**: All symbols which are users of a function address or block address will be replaced with a renamed symbol designating them as a function pointer. We could have replaced every constant which is a code pointer with a pass that creates a new temporary, moves the constant into the temporary, and then blesses it with the function pointer tag via a `settag` instruction. Unfortunately, this does not work because in LLVM one can't replace a constant with a non-constant (this is because other constants might depend on the value of that constant, apparantely). Additionally, this would typically end up replacing the constant right before each use.

- **Create A Constants Table**: We could have created a table of constants somewhere in the program section. We would then have had to insert a stub before the program runs which blesses those constants. This would have less overhad than our current approach, but is less simple to implement in LLVM.

- **Add Blessing Flags To Instructions**: A simpler approach is, if supported by the underlying architecture, to simply add a bit to any instruction that contains an immediate derived from a code pointer type. The hardware policy can then be that if an instruction is executing and the code pointer tag bit is set on it, the result of the instruction should have the code pointer tag set. One complication is that in LLVM bitcode, a single statement may expand to multiple assembly lines. So for instance `.call (x, y, z)` where z is supposed to be a function pointer would incorrectly set x and y to have function pointer tags as well. This is undesirablly imprecise.

6.4   Function Pointer Policy

Once we have properly implemented function pointer passes which set the tags in an executable appropriately, we design a function pointer policy for our processor to enforce.

In the basic function pointer policy, function pointer tags are moved on store and load. They are propagated through addition and subtraction with non-function-pointer-tagged data, but not through other arithmetic or logical functions. This is to avoid the unfortunately common practice of pointer arithmetic often employed by C programmers; such violations of the C standard are almost idiomatic in many C codebases as extensively documented in [14].

Before an indirect call, the processor checks that the register being called through has the function pointer tag. If not, the hardware raises a trap.

We were able to verify that this worked successfully on small programs; however a full implementation needs the following features, which are left to future work:

- C library support for handling calls such as `dlopen` and `dlsym`. Presumably testing a large corpus of C programs would also reveal frameworks which have implemented similair functionality outside of the C standard library, which would require refactoring to avoid spurious traps.

- As previously mentioned, ensuring the proper marking of globals and PLT and GOT entries is critical to avoid spurious traps from library and interprocess calls.

6.5   Function Pointer Policy Evaluation

A function pointer policy preventing the attacker from directly injecting function pointers would immediately stop a range of wide range of attacks such as virtual table overwrites and use-after-free (asuming integration with the memory deallocator that strips tags from freed memory).

In theory, a strictly enforced function pointer policy—particularly in conjunction with a strict return pointer policy—might approach the security guarantees of full memory safety. However, as implemented our policy leaves some attack surface area, though we believe it is greatly reduced compared to the status quo. Attacks are still possible by mixing with data; e.g. replay attacks or attacks in which an attacker cannot forge a code pointer directly, but can add or subtract with injected data to produce a usefully malicious pointer. By evaluating the compatability effects of the policy in future work, we hope to further reduce this attack surface by producing a linearity function pointer policy or even stricter variations.

Chapter 7

Future Work and Conclusion

7.1 ACICS Improvements

Much interesting work remains in the field of automatically detecting ACICS sites and building viable exploits in an automated fashion. If fine-grained CFI schemes become the norm, identifying such sites will become an important part of future offensive security research in the same way that ROP compilers entered public consciousness as ASLR defenses grew in popularity.

Specifically, an ACICS detection tool would benefit from a true backwards-reaching analysis, as opposed to the heuristic approach shown in this work. This would be better served by using a binary instrumentation framework, such as PIN [40] for instance, which has better guarantees regarding lack of impact on the executing program.

Furthermore, it would be highly beneficial to integrate with the test suites of larger programs to experiment with viable inputs. We attempted to do this with Apache and their web server benchmarking tool, but ran into compatibility issues when running their tools under GDB.

Additionally, we would benefit from an automated tool to find target functions. This is difficult in the general case, but a subset of program locations with functions that call dangerous system calls using parameters found in arguments to the function would be a low-hanging fruit. Perhaps work which relaxes constraints on symbolic execution [39] can provide a basis for efficiently finding such paths.

In the end, we believe that if CFI becomes widely deployed, future work will focus on building a tool that is to fine-grained CFI schemes as a ROP compiler is to ASLR; such a tool would rapidly take a vulnerability and characterize which ICS, if any, can be used as ACICS for that vulnerability.

7.2   Hardware and Policy Improvements

The function pointer policy does not yet consider functional programming languages that might be compiled to LLVM intermediary representation, as well as newer features such as C++ lambda functions. If a function does not have global visibility, but can still influence control flow, how should it be handled? LLVM contains a type for `std::function`, which would currently not show up in our globals analysis and needs to be properly handled.

Furthermore, our function pointer policy is relatively simple to implement for C but needs deep library support for C++ objects; the virtual table layout in memory as well as functions such as constructors, copy constructors, and deconstructors all need to be modified to be tag-aware in order to maintain proper security guarantees.

As previously mentioned, it would also be highly beneficial to test tag policies with larger programs. We are currently prevented from doing by bugs in the RISC-V ports of LLVM and GCC; for instance, we could not compile Nginx successfully. Future work should examine Nginx as well as Apache, BIND, and other popular server applications which are often targets for remote attack.

In future work, the `tagenforce` instruction might enable a mode which permanently turns on tag enforcement for the current process. More generally, it might selectively enable different policies for different applications or even application sub-components. For instance, an application author might know that a certain module of their software can run with the strictest possible hardware-enforced pointer policies and enable it for that module—while another module might need to perform just-in-time compilation and strange control flow operations which would violate any policy we have defined. Aside from security segmentation, dynamically enabling policies also

has the advantage of allowing better backwards compatibility—programs or program parts which generate false positive traps when under restrictive policies can be run under less restrictive versions until they can be adapted by their authors.

7.3   Conclusion

In our modern world where the number of devices interconnected via the Internet is already in the billions and rapidly growing, the need for new computer defenses will only increase. This work underscores just how difficult and sometimes subtle the development of defenses preventing program exploitation can be.

We have seen that current CFI schemes are seriously flawed, both in practical realization and also, unfortunately, in their assumption that enforcement of ideal CFI techniques will be able to stop attackers. Our *Control Jujutsu* attack provides a powerful algorithm that attackers can use to attack any CFI scheme. We hope that this work will motivate the development of principled, fundamental defenses to code reuse attacks. We have also described extensions to the new RISC-V hardware architecture which might one day provide a basis for these more fundamental defenses, and designed and evaluated potential software control-flow policies which might run on that hardware.

Bibliography

[1] clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[2] List of google chrome cves. MITRE Corporation Database CVE Database Listings.

[3] List of mozilla firefox cves. MITRE Corporation Database CVE Database Listings.

[4] Vulnerability summary for cve-2013-2028. Available from MITRE, CVE-ID CVE-2013-2028., 2013.

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

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, ICFEM'05, pages 111–124, Berlin, Heidelberg, 2005. Springer-Verlag.

[7] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *Security and Privacy*, 2008.

[8] O. H. Alhazmi, Y. K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Comput. Secur.*, 26(3):219–228, May 2007.

[9] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for risc-v. Technical report, University of California at Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf, aug 2014.

[10] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, Washington, DC, USA, 2014. IEEE Computer Society.

[11] Erwin Bosman and Herbert Bos. Framing signals-a return to portable shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 243–258. IEEE, 2014.

[12] Alex Bradbury, Gavin Ferris, and Robert Mullins. Tagged memory and minion cores in the lowrisc soc. Memo, University of Cambridge, 2014.

[13] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[14] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. *SIGARCH Comput. Archit. News*, 43(1):117–130, March 2015.

[15] Joel Sandin Daniel Mayer. Time trial: Racing towards practical remote timing attacks. Technical report, Matasano Security Research, aug 2014.

[16] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[17] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *SIGPLAN Not.*, 43(3):103–114, March 2008.

[18] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. *SIGARCH Comput. Archit. News*, 43(1):487–502, March 2015.

[19] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. of IEEE S&P*, 2015.

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

[21] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, SecuCode '09, pages 19–26, New York, NY, USA, 2009. ACM.

[22] Michael Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

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

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

[25] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014.

[26] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[27] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, 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.

[28] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 276–291. IEEE, 2014.

[29] 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.

[30] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 81–96, Washington, D.C., August 2015. USENIX Association.

[31] Ali José Mashtizadeh, Andrea Bittau, David Mazières, and Dan Boneh. Cryptographically enforced control flow integrity. *CoRR*, abs/1408.1451, 2014.

[32] Swamy Shivaganga Nagaraju, Cristian Craioveanu, Elia Florio, and Matt Miller. Software vulnerability exploitation trends. Technical report, Microsoft Corporation, 2013.

[33] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual*

IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, pages 175:175–175:184, New York, NY, USA, 2014. ACM.

[34] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44(6):245–258, June 2009.

[35] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM.

[36] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[37] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA, 2013. USENIX Association.

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

[39] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., August 2015. USENIX Association.

[40] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. Pin: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[41] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 54–65, New York, NY, USA, 2014. ACM.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[43] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[44] SUSE, https://www.suse.com/documentation/sles-12/book_sle_tuning/data/sec_tuning_power_cpu.html. *Power Management at CPU Level*, suse linux enterprise server 12 edition, jul 2015.

[45] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *Proceedings of the 23rd Usenix Security Symposium*, San Diego, CA, 2014.

[46] Denys Vlasenko. Busybox: The swiss army knife of embedded linux. http://www.busybox.net.

[47] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[48] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[49] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.