Missing the Point(er):
On the Effectiveness of Code Pointer Integrity[1]

Isaac Evans*, Sam Fingeret†, Julián González†, Ulziibayar Otgonbaatar†, Tiffany Tang†,
Howard Shrobe†, Stelios Sidiroglou-Douskos†, Martin Rinard†, Hamed Okhravi*

†MIT CSAIL, Cambridge, MA
Email: {samfin, jugonz97, ulziibay, fable, hes, stelios, rinard}@csail.mit.edu
*MIT Lincoln Laboratory, Lexington, MA
Email: {isaac.evans, hamed.okhravi}@ll.mit.edu

Abstract—**Memory corruption attacks continue to be a major vector of attack for compromising modern systems. Numerous defenses have been proposed against memory corruption attacks, but they all have their limitations and weaknesses. Stronger defenses such as complete memory safety for legacy languages (C/C++) incur a large overhead, while weaker ones such as practical control flow integrity have been shown to be ineffective. A recent technique called code pointer integrity (CPI) promises to balance security and performance by focusing memory safety on code pointers thus preventing most control-hijacking attacks while maintaining low overhead. CPI protects access to code pointers by storing them in a safe region that is protected by instruction level isolation. On x86-32, this isolation is enforced by hardware; on x86-64 and ARM, isolation is enforced by information hiding. We show that, for architectures that do not support segmentation in which CPI relies on information hiding, CPI's safe region can be leaked and then maliciously modified by using data pointer overwrites. We implement a proof-of-concept exploit against Nginx and successfully bypass CPI implementations that rely on information hiding in 6 seconds with 13 observed crashes. We also present an attack that generates no crashes and is able to bypass CPI in 98 hours. Our attack demonstrates the importance of adequately protecting secrets in security mechanisms and the dangers of relying on difficulty of guessing without guaranteeing the absence of memory leaks.**

I. INTRODUCTION

Despite considerable effort, memory corruption bugs and the subsequent security vulnerabilities that they enable remain a significant concern for unmanaged languages such as C/C++. They form the basis for attacks [14] on modern systems in the form of code injection [40] and code reuse [49, 14].

The power that unmanaged languages provide, such as low-level memory control, explicit memory management and direct access to the underlying hardware, make them ideal for systems development. However, this level of control comes at a significant cost, namely lack of memory safety. Rewriting systems code with managed languages has had limited success [24] due to the perceived loss of control that mechanisms such as garbage collection may impose, and the fact that millions of lines of existing C/C++ code would need to be ported to provide similar functionality. Unfortunately,

retrofitting memory safety into C/C++ applications can cause significant overhead (up to 4x slowdown) [36] or may require annotations [37, 28].

In response to these perceived shortcomings, research has focused on alternative techniques that can reduce the risk of code injection and code reuse attacks without significant performance overhead and usability constraints. One such technique is Data Execution Prevention (DEP). DEP enables a system to use memory protection to mark pages as non-executable, which can limit the introduction of new executable code during execution. Unfortunately, DEP can be defeated using code reuse attacks such as return-oriented programming [11, 17], jump-oriented programming [10] and return-into-libc attacks [56].

Randomization-based techniques, such as Address Space Layout Randomization (ASLR) [43] and its medium- [30], and fine-grained variants [57] randomize the location of code and data segments thus providing probabilistic guarantees against code reuse attacks. Unfortunately, recent attacks demonstrate that even fine-grained memory randomization techniques may be vulnerable to memory disclosure attacks [52]. Memory disclosure may take the form of direct memory leakage [53] (i.e., as part of the system output), or it can take the form of indirect memory leakage, where fault or timing side-channel analysis attacks are used to leak the contents of memory [9, 47]. Other forms of randomization-based techniques include instruction set randomization (ISR) [8] or the multicompiler techniques [26]. Unfortunately, they are also vulnerable to information leakage attacks [53, 47].

Control flow integrity (CFI) is a widely researched runtime enforcement technique that can provide practical protection against code injection and code reuse attacks [3, 61, 62]. CFI provides runtime enforcement of the intended control flow transfers by disallowing transfers that are not present in the application's control flow graph (CFG). However, precise enforcement of CFI can have a large overhead [3]. This has motivated the development of more practical variants of CFI that have lower performance overhead but enforce weaker restrictions [61, 62]. For example, control transfer checks are relaxed to allow transfers to *any* valid jump targets as opposed to the *correct* target. Unfortunately, these implementations have been shown to be ineffective because they allow enough

---

valid transfers to enable an attacker to build a malicious payload [21].

A recent survey of protection mechanisms [55] shows that most available solutions are either (*a*) incomplete, (*b*) bypassable using known attacks, (*c*) require source code modifications or (*d*) impose significant performance overhead.

Recently a new technique, code pointer integrity (CPI), promises to bridge the gap between security guarantees and performance/usability. CPI enforces selective memory safety on code pointers (i.e., it does not protect data pointers) without requiring any source code modifications. The key idea behind CPI is to isolate and protect code pointers in a separate *safe region* and provide runtime checks that verify the code pointer correctness on each control transfer. Since modification of a code pointer is necessary to implement a control hijacking attack, the authors of CPI argue that it is effective against the most malicious types of memory corruption attacks. As code pointers represent a small fraction of all pointers, CPI is significantly more efficient than established techniques for enforcing complete memory safety (average 2.9% for C, 8.4% for C/C++) [31].

In this paper, we present an attack on CPI that uses a data pointer vulnerability to launch a timing side-channel that leaks information about the protected safe region. Our attack takes advantage of two design weaknesses in CPI. First, on architectures that do not support segmentation protection, such as x86-64 and ARM, CPI uses information hiding to protect the safe region. Second, to achieve the low performance overhead, CPI focuses protection on code pointers. Since the safe region is kept in the same memory space as the code it is protecting, to avoid expensive context switches, it is also subject to leakage and overwrite attacks. We show that an attacker can disclose the location of the safe region using a timing side-channel attack. Once the location of a code pointer in the safe region is known, the metadata of the pointer is modified to allow the location of a ROP chain. Then the pointer is modified to point to a ROP chain that can successfully complete the hijacking attack.

In our evaluation of CPIs implementation, we discovered a number of implementation flaws that can facilitate an attack against CPI. In this paper, we focus on an attack that exploits a flaw in the use of information hiding to protect the safe region for architectures that do not provide hardware isolation (e.g., x86-64 and ARM). In other words, for the x86-64 and ARM architectures, we assume the weakest assumptions for the attacker. In fact, the only assumption necessary for an attacker to break CPI is control of the stack, which is consistent with other code reuse attacks and defenses in the literature [49, 23, 57]. For the remainder of the paper, when referring to CPI, we are referring to the information-hiding based implementations of CPI.

At a high level our attack works as follows. First, by controlling the stack, we use a data pointer overwrite to redirect a data pointer to a random location in memory map (mmap) which is used by CPI. Using a timing side-channel attack, we leak large parts of the safe region. We then use a data-pointer overwrite attack to modify the safe region and tamper with base and bounds information for a code pointer that we need for the actual payload. This can be summarized in the following steps:

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

2) **Data Collection:** Using the data pointer vulnerability, we measure round-trip response times to our attack application in order to collect the timing samples. We create a mapping between the smallest cumulative delay slope and byte 0, and the largest slope and byte 255. We use these two mappings to interpolate cumulative delay slopes for all possible byte values (0-255). This enables us to read the contents of specific memory locations with high accuracy.

3) **Locate Safe Region:** Using information about the possible location of the safe region with respect to the randomized location of mmap, we launch a search that starts at a reliably mapped location within the safe region and traverse the safe region until we discover a sequence of bytes that indicates the location of a known library (e.g., the base of libc). Under the current implementation of CPI, discovering the base of libc allows us to trivially compute the base address of the safe region. Up to this point, the attack is completely transparent to CPI and may not cause any crash or detectable side effect.

4) **Attack Safe Region:** Using the safe region table address, we can compute the address of any code pointer stored in the safe region. At this point, we can change a code pointer and any associate metadata to enable a control hijacking attack (e.g., a ROP gadget). CPI does not detect the redirection as a violation because we have already modified its safe region to accept the new base and bound for the code pointer.

In the CPI paper, the authors argue that leaking large parts of memory or brute-forcing the safe region causes a large number of crashes that can be detected using other means [31]. We show that this assumption is incorrect and in fact leaking large parts of the safe region can happen without causing any crash in the target process. Another assumption in CPI is that if there is no pointer into the safe region, its location cannot be leaked. We show that this assumption is also incorrect. By jumping into a randomly selected location in mmap, the attack can start leaking the safe region without requiring any pointer to it.

To evaluate our attack, we construct a proof-of-concept attack on a CPI-protected version on Nginx [45]. Our evaluation shows that in Ubuntu Linux with ASLR, it takes 6 seconds to bypass CPI with 13 crashes. Our analysis also shows

that an attack can be completed without any crashes in ∼98 hours for the most performant and complete implementation of CPI. This implementation relies on ASLR support from the operating system.

A. Contributions

This paper make the following contributions:

- **Attack on CPI:** We show that an attacker can defeat CPI, on x86-64 architectures, assuming only control of the stack. Specifically, we show how to reveal the location of the safe region using a data-pointer overwrite without causing any crashes, which was assumed to be impossible by the CPI authors.
- **Proof of Concept Attack on Nginx:** We implement a proof-of-concept attack on a CPI protected version of the popular Nginx web server. We demonstrate that our attack is accurate and efficient (it takes 6 seconds to complete with only 13 crashes).
- **Experimental Results:** We present experimental results that demonstrate the ability of our attack to leak the safe region using a timing side-channel attack.
- **Countermeasures:** We present several possible improvements to CPI and analyze their susceptibility to different types of attacks.

Next, Section II describes our threat model which is consistent with CPI's threat model. Section III provides a brief background on CPI and the side-channel attacks necessary for understanding the rest of the paper. Section IV describes our attack procedure and its details. Section V presents the results of our attack. Section VI describes a few of CPI's implementation flaws. Section VII provides some insights into the root cause of the problems in CPI and discusses possible patch fixes and their implications. Section VIII describes the possible countermeasures against our attack. Section IX reviews the related work and Section X concludes the paper.

II. THREAT MODEL

In this paper, we assume a realistic threat model that is both consistent with prior work and the threat model assumed by CPI [31]. For the attacker, we assume that there exists a vulnerability that provides control of the stack (i.e., the attacker can create and modify arbitrary values on the stack). We also assume that the attacker cannot modify code in memory (e.g., memory is protected by DEP [41]). We also assume the presence of ASLR [43]. As the above assumptions prevent code injection, the attacker would be required to construct a code reuse attack to be successful.

We also assume that CPI is properly configured and correctly implemented. As we will discuss later, CPI has other implementation flaws that make it more vulnerable to attack, but for this paper we focus on its design decision to use information hiding to protect the safe region.

III. BACKGROUND

This section presents the necessary background information required to understand our attack on CPI. Specifically,

the section begins with an overview of CPI and continues with information about remote leakage attacks. For additional information, we refer the reader to the CPI paper [31] and a recent remote leakage attack paper [47].

A. CPI Overview

CPI consists of three major components: static analysis, instrumentation, and safe region isolation.

1) Static Analysis: CPI uses type-based static analysis to determine the set of sensitive pointers to be protected. CPI treats all pointers to functions, composite types (e.g., arrays or `structs` containing sensitive types), universal pointers (e.g., `void*` and `char*`), and pointers to sensitive types as sensitive types (note the recursive definition). CPI protects against the redirection of sensitive pointers that can result in control-hijacking attacks. The notion of sensitivity is dynamic in nature: at runtime, a pointer may point to a benign integer value (non-sensitive) and it may also point to a function pointer (sensitive) at some other part of the execution. Using the results of the static analysis, CPI stores the metadata for checking the validity of code pointers in its safe region. The metadata includes the value of the pointer and its lower and upper thresholds. An identifier is also stored to check for temporal safety, but this feature is not used in the current implementation of CPI. Note that static analysis has its own limitations and inaccuracies [33] the discussion of which is beyond the scope of this paper.

2) Instrumentation: CPI adds instrumentation that propagates metadata along pointer operations (e.g. pointer arithmetic or assignment). Instrumentation is also used to ensure that only CPI intrinsic instructions can manipulate the safe region and that no pointer in the code can directly reference the safe region. This is to prevent any code pointers from disclosing the location of the safe region using a memory disclosure attack (on code pointers).

3) Safe Region Isolation: On the x86-32 architecture CPI relies on segmentation protection to isolate the safe region. On architectures that do not support segmentation protection, such as x86-64 and ARM, CPI uses information hiding to protect the safe region. There are two major weaknesses in CPI's approach to safe region isolation in x86. First, the x86-32 architecture is slowly phased out as systems migrate to 64-bit architectures and mobile architectures. Second, as we show in our evaluation, weaknesses in the implementation of the segmentation protection in CPI makes it bypassable. For protection in the x86-64 architecture, CPI relies on the size of the safe region (2^{42} bytes), randomization and sparsity of its safe region, and the fact that there are no direct pointers to its safe region. We show that these are weak assumptions at best.

CPI authors also present a weaker but more efficient version of CPI called Code Pointer Separation (CPS). CPS enforces safety for code pointers, but not pointers to code pointers. Because the CPI authors present CPI as providing the strongest security guarantees, we do not discuss CPS and the additional safe stack feature further. Interested readers can refer to the

original publication for more in-depth description of these features.

B. Side Channels via Memory Corruption

Side channel attacks using memory corruption come in two broad flavors: fault and timing analysis. They typically use a memory corruption vulnerability (e.g., a buffer overflow) as the basis from which to leak information about the contents of memory. They are significantly more versatile than traditional memory disclosure attacks [54] as they can limit crashes, they can disclose information about a large section of memory, and they only require a single exploit to defeat code-reuse protection mechanisms.

Blind ROP (BROP) [9] is an example of a fault analysis attack that uses the fault output of the application to leak information about memory content (i.e., using application crashes or freezes). BROP intentionally uses crashes to leak information and can therefore be potentially detected by mechanisms that monitor for an abnormal number of program crashes.

Seibert, *et al.* [47] describe a variety of timing- and fault-analysis attacks. In this paper, we focus on using timing channel attacks via data-pointer overwrites. This type of timing attack can prevent unwanted crashes by focusing timing analysis on allocated pages (e.g., the large memory region allocated as part of the safe region).

Consider the code sequence below. If `ptr` can be over-written by an attacker to point to a location in memory, the execution time of the `while` loop will be correlated with the byte value to which `ptr` is pointing. For example, if `ptr` is stored on the stack, a simple buffer overflow can corrupt its value to point to an arbitrary location in memory. This delay is small (on the order of nanoseconds); however, by making numerous queries over the network and keeping the fastest samples (cumulative delay analysis), an attacker can get an accurate estimate of the byte values [47, 16]. In our attack, we show that this type of attack is a practical technique for disclosing CPI's safe region.

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

C. Memory Entropy

One of the arguments made by the authors of the CPI technique is that the enormous size of virtual memory makes guessing or brute force attacks difficult if not impossible. Specifically, they mention that the 48 bits of addressable space in x86-64 is very hard to brute force. We show that in practice this assumption is incorrect. First, the entropy faced by an attacker is not 48 bits but rather 28 bits: the entropy for the base address of the mmap, where CPI's safe region is stored, is 28 bits [39]. Second, an attacker does not need to know the exact start address of mmap. The attacker only needs to redirect the data pointer to *any* valid location inside mmap. Since large parts of the mmap are used by libraries and the

CPI safe region, an attacker can land inside an allocated mmap page with high probability. In our evaluation we show that this probability is as high as 1 for the average case. In other words, since the size of the mmap region is much larger than the entropy in its start address, an attacker can effectively land in a valid location inside mmap without causing crashes.

IV. Attack Methodology

This section presents a methodology for performing attacks on applications protected with CPI. As outlined in Section II, the attacks on CPI assume an attacker with identical capabilities as outlined in the CPI paper [31]. The section begins with a high-level description of the attack methodology and then proceeds to describe a detailed attack against Nginx [45] using the approach.

At a high level, our attack takes advantage of two design weaknesses in CPI. First, on architectures that do not support segmentation protection, such as x86-64 and ARM, CPI uses information hiding to protect the safe region. Second, to achieve low performance overhead, CPI focuses protection on code pointers (i.e., it does not protect data pointers). This section demonstrates that these design decisions can be exploited to bypass CPI.

Intuitively, our attack exploits the lack of data pointer protection in CPI to perform a timing side channel attack that can leak the location of the safe region. Once the location of a code pointer in the safe region is known, the code pointer, along with its metadata, is modified to point to a ROP chain that completes the hijacking attack. We note that using a data-pointer overwrite to launch a timing channel to leak the safe region location can be completely transparent to CPI and may avoid any detectable side-effects (i.e., it does not cause the application to crash).

The attack performs the following steps:
1) Find data pointer vulnerability
2) Gather data
   - Identify statistically unique memory sequences
   - Collect timing data on data pointer vulnerability
3) Locate safe region
4) Attack safe region

Next, we describe each of these steps in detail.

A. Vulnerability

The first requirement to launch an attack on CPI is to discover a data pointer overwrite vulnerability in the CPI-protected application. Data pointers are not protected by CPI; CPI only protects code pointers.

The data pointer overwrite vulnerability is used to launch a timing side-channel attack [47], which, in turn, can leak information about the safe region. In more detail, the data pointer overwrite vulnerability is used to control a data pointer that is subsequently used to affect control flow (in our case, the number of iterations of a loop) and can be used to reveal the contents of the pointer (i.e., byte values) via timing information. For example, if the data pointer is stored on the

stack, it can be overwritten using a stack overflow attack; if it is stored in heap, it can be overwritten via a heap corruption attack.

In the absence of complete memory safety, we assume that such vulnerabilities will exist. This assumption is consistent with the related work in the area [50, 12]. In our proof-of-concept exploit, we use a stack buffer overflow vulnerability similar to previous vulnerabilities [1] to redirect a data pointer in Nginx.

B. Data Collection

Given a data-pointer vulnerability, the next step is collect enough data to accurately launch a timing side-channel attack that will reveal the location of the safe region.

The first step is to generate a request that redirects the vulnerable data pointer to a carefully chosen address in memory (see Section IV-C). Next, we need to collect enough information to accurately estimate the byte value that is dereferenced by the selected address. To estimate the byte value, we use the cumulative delay analysis described in Equation 1.

$$byte = c \sum_{i=1}^{s} (d_i - baseline) \qquad (1)$$

In the above equation, $baseline$ represents the average round trip time (RTT) that the server takes to process requests for a byte value of zero. d_i represents the delay sample RTT for a nonzero byte value, and s represents the number of samples taken.

Once we set $byte = 0$, the above equation simplifies to:

$$baseline = \frac{\sum d_i}{s}$$

Due to additional delays introduced by networking conditions, it is important to establish an accurate $baseline$. In a sense, the $baseline$ acts as a band-pass filter. In other words, we subtract the $baseline$ from d_i in Eq. 1 so that we are only measuring the cumulative differential delay caused by our chosen loop.

We then use the set of delay samples collected for byte 255 to calculate the constant c. Once we set $byte = 255$, the equation is as follows:

$$c = \frac{255}{\sum_{i=1}^{s} (d_i) - s * baseline}$$

Once we obtain c, which provides of the ratio between the byte value and cumulative differential delay, we are able to estimate byte values.
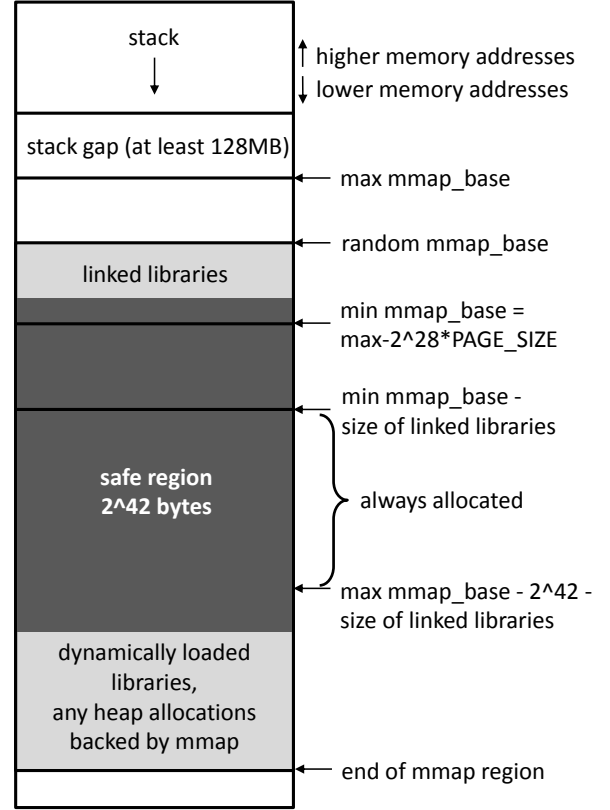

Fig. 1. Safe Region Memory Layout.

C. Locate Safe Region

Figure 1 illustrates the memory layout of a CPI-protected application on the x86-64 architecture. The stack is located at the top of the virtual address space and grows downwards (towards lower memory addresses) and it is followed by the stack gap. Following the stack gap is the base of the mmap region ($mmap\_base$), where shared libraries (e.g., libc) and other regions created by the mmap() system call reside. In systems protected by ASLR, the location of $mmap\_base$ is randomly selected to be between max_mmap_base (located immediately after the stack gap) and min_mmap_base. min_mmap_base is computed as:

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

where $aslr\_entropy$ is 2^{28} in 64-bit systems, and the $page\_size$ is specified as an operating system parameter (typically 4KB). The safe region is allocated directly after any linked libraries are loaded on $mmap\_base$ and is 2^{42} bytes. Immediately after the safe region lies the region in memory where any dynamically loaded libraries and any mmap-based heap allocations are made.

Given that the safe region is allocated directly after all linked libraries are loaded, and that the linked libraries are linked deterministically, the location of the safe region can

be computed by discovering a known location in 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. Our attack works even if countermeasures are employed to allocate the safe region in a randomized location as we discuss later.

To discover the location of a known library, such as the base of libc, the attack needs to scan every address starting at min_mmap_base, and using the timing channel attack described above, search for a signature of bytes that uniquely identify the location.

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

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

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

1) Redirect a data pointer into the always allocated part of the safe region (see Fig. 1).
2) Go back in memory by the size of libc.
3) Scan some bytes. If the bytes are all zero, goto step 2. Else, scan more bytes to decide where we are in libc.
4) Done.

Note that discovery of a page that resides in libc directly reveals the location of the safe region.

Using this procedure, the number of scans can be reduced to:

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

Here $libc\_size$, in our experiments, is approximately 2^{21}. In other words, the estimated number of memory scans is: $2^{28} * 2^{12}/2^{21} = 2^{19}$. This non-crashing scan strategy is depicted on the left side of Fig. 2.

We can further reduce the number of memory scans if we are willing to tolerate crashes due to dereferencing an address not mapped to a readable page. Because the pages above $mmap\_base$ are not mapped, dereferencing an address above
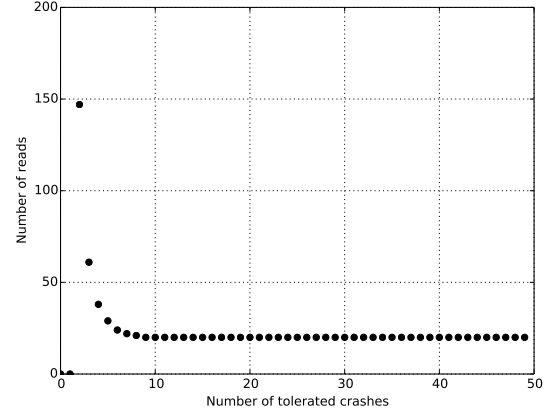

Fig. 3. Tolerated Number of Crashes

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

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

More generally, given that T crashes are allowed for our scanning, we would like to characterize the minimum number of page reads needed to locate a crashing boundary under the optimum scanning strategy. A reason for doing that is when $T < 19$, our binary search method is not guaranteed to find a crashing boundary in the worst case.

We use dynamic programming to find the optimum scanning strategy for a given T. Let $f(i, j)$ be the maximum amount of memory an optimum scanning strategy can cover, incurring up to i crashes, and performing j page reads. Note that to cause a crash, you need to perform a read. Thus, we have the recursion

$$f(i, j) = f(i, j - 1) + f(i - 1, j - 1) + 1$$

This recursion holds because in the optimum strategy for $f(i, j)$, the first page read will either cause a crash or not.
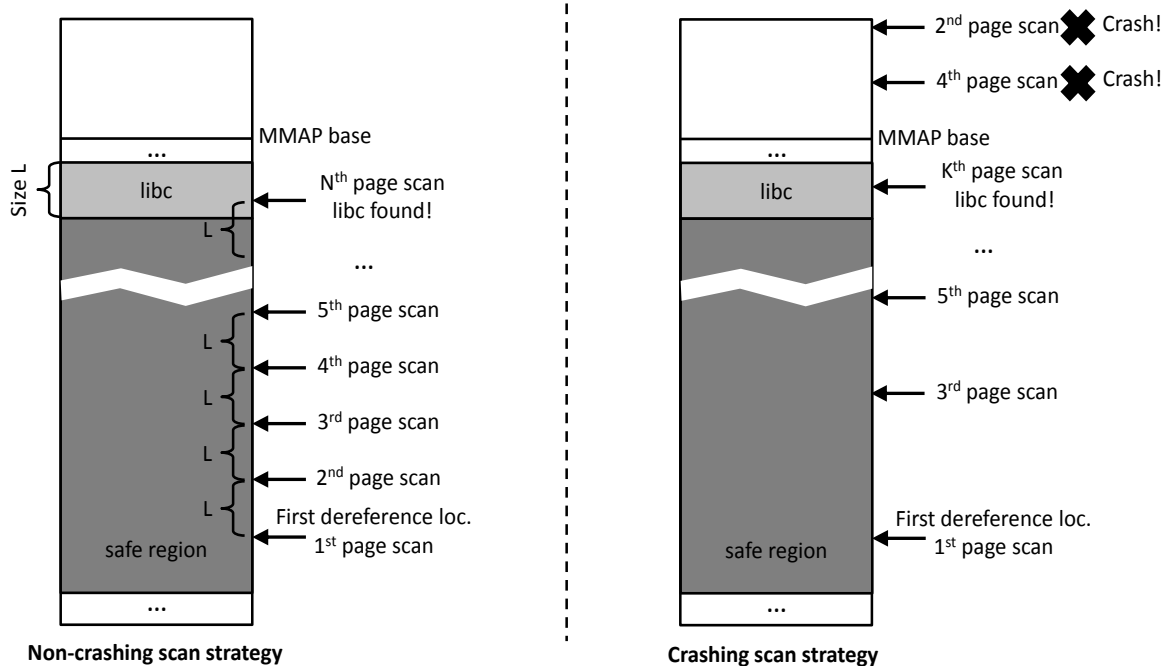
Fig. 2. Non-Crashing and Crashing Scan Strategies.

When a crash happens, it means that libc is below the first page we read, thus the amount of memory we have to search is reduced to a value that is at most $f(i-1, j-1)$. As for the latter case, the amount we have to search is reduced to a value that is at most $f(i, j-1)$.

Having calculated a table of values from our recursion, we can use it to inform us about the scanning strategy that incurs at most T crashes. Fig. 3 shows the number of reads performed by this strategy for different T values.

Because we know the layout of the library region in advance, when we find a crash boundary we know that subtracting $8 * libc\_size$ from x will guarantee an address in the safe region because this amount is greater than the size of all linked libraries combined. Thus, at most 8 more reads will be needed to locate an address in libc. The crashing scan strategy is depicted on the right side of Fig. 2.

We can still obtain a significant improvement even if the application does rerandomize its address space when it restarts after a crash. Suppose that we can tolerate T crashes on average. Rather than begin our scan at address:

$$min\_mmap\_base = $$
$$max\_mmap\_base - aslr\_entropy * page\_size \quad (2)$$

we begin at:

$$max\_mmap\_base - \frac{1}{T+1} \left(aslr\_entropy * page\_size \right)$$

With probability $\frac{1}{T+1}$, it will be the case that $mmap\_base$ is above this address and we will not crash, and the number

of reads will be reduced by a factor of $\frac{1}{T+1}$. With probability $1 - \frac{1}{T+1}$, this will crash the application immediately and we will have to try again. In expectation, this strategy will crash T times before succeeding.

Note that in the rerandomization case, any optimal strategy will choose a starting address based on how many crashes can be tolerated and if this first scan does not crash, then the difference between consecutive addresses scanned will be at most $libc\_size$. If the difference is ever larger than this number, then it may be the case that libc is jumped over, causing a crash, and all knowledge about the safe region is lost due to the rerandomization. If the difference between consecutive addresses x, y satisfies $y - x > libc\_size$, then replacing x by $y - libc\_size$ and shifting all addresses before x by $y - libc\_size - x$ yields a superior strategy since the risk of crashing is moved to the first scan while maintaining the same probability of success.

Once the base address of mmap is discovered using the timing side channel, the address of the safe region table can be computed as follows:

$$table\_address = libc\_base - 2^{42}$$

D. Attack Safe Region

Using the safe region table address, the address of a code pointer of interest in the CPI protected application, `ptr_address`, can be computed by masking with the `cpi_addr_mask`, which is `0x00fffffffff8`, and then multiplying by the size of the table entry, which is 4.

Armed with the exact address of a code pointer in the safe region, the value of that pointer can be hijacked to point to a library function or the start of a ROP chain to complete the attack.

E. Attack Optimizations

A stronger implementation of CPI might pick an arbitrary address for its safe region chosen randomly between the bottom of the linked libraries and the end of the mmap region. Our attack still works against such an implementation and can be further optimized.

We know that the safe region has a size of 2^{42} bytes. Therefore, there are $2^{48}/2^{42} = 2^6 = 64$ possibilities for where we need to search. In fact, in a real world system like Ubuntu 14.04 there are only $2^{46.5}$ addresses available to mmap on Ubuntu x86-64 –thus there is a $\frac{1}{25}$ chance of getting the right one, even with the most extreme randomization assumptions. Furthermore, heap and dynamic library address disclosures will increase this chance. We note that CPI has a unique signature of a pointer value followed by an empty slot, followed by the lower and upper bounds, which will make it simple for an attacker to verify that the address they have reached is indeed in the safe region. Once an address within the safe region has been identified, it is merely a matter of time before the attacker is able to identify the offset of the safe address relative to the table base. There are many options to dramatically decrease the number of reads to identify exactly where in the safe region we have landed. For instance, we might profile a local application's safe region and find the most frequently populated addresses modulo the system's page size (since the base of the safe region must be page-aligned), then search across the safe region in intervals of the page size at that offset. Additionally, we can immediately find the offset if we land on any value that is unique within the safe region by comparing it to our local reference copy.

We can now make some general observations about choosing the variable of interest to target during the search. We would be able to search the fastest if we could choose a pointer from the largest set of pointers in a program that has the same addresses modulo the page size. For instance, if there are 100 pointers in the program that have addresses that are 1 modulo the page size, we greatly increase our chances of finding one of them early during the scan of the safe region.

Additionally, the leakage of any locations of other libraries (making the strong randomization assumption) will help identify the location of the safe region. Note that leaking all other libraries is within the threat model of CPI.

V. Measurements and Results

We next present experimental results for the attack described in Section IV on Nginx 1.6.2, a popular web server. We compile Nginx with clang/CPI 0.2 and the `-flto -fcpi` flags. Nginx is connected to the attacker via a 1Gbit wired LAN connection. We perform all tests on a server with a quad-core Intel i5 processor with 4 GB RAM.

A. Vulnerability

We patch Nginx to introduce a stack buffer overflow vulnerability allowing the user to gain control of a parameter used as the upper loop bound in the Nginx logging system. This is similar to the effect that an attacker can achieve with (CVE-2013-2028) seen in previous Nginx versions [1]. The vulnerability enables an attacker to place arbitrary values on the stack in line with the threat model assumed by CPI (see Section II). We launch the vulnerability over a wired LAN connection, but as shown in prior work, the attack is also possible over wireless networks [47].

Using the vulnerability, we modify a data pointer in the Nginx logging module to point to a carefully chosen address. The relevant loop can be found in the source code in `nginx_http_parse.c`.

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

The data pointer vulnerability enables control over the number of iterations executed in the loop. Using the timing analysis presented in Section IV, we can distinguish between zero pages and nonzero pages. This optimization enables the attack to efficiently identify the end of the safe region, where nonzero pages indicate the start of the linked library region.

B. Timing Attack

We begin the timing side channel attack by measuring the HTTP request round trip time (RTT) for a static web page (0.6 KB) using Nginx. We collect 10,000 samples to establish the average baseline delay. For our experiments, the average RTT is $3.2ms$. Figure 4 and 5 show the results of our byte estimation experiments. The figures show that byte estimation using cumulative differential delay is accurate to within 2% (± 20).
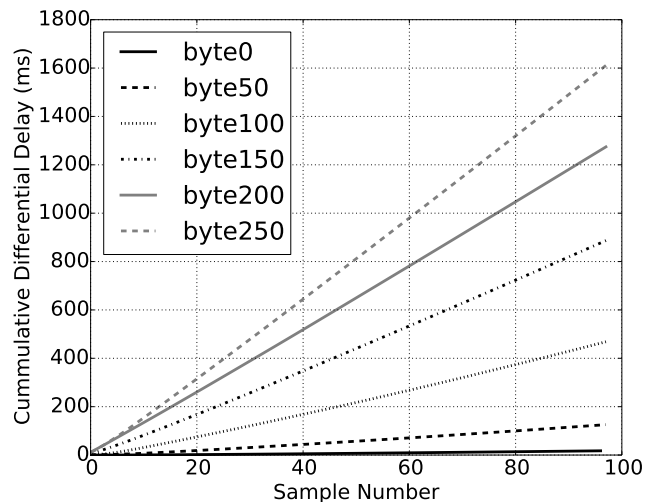

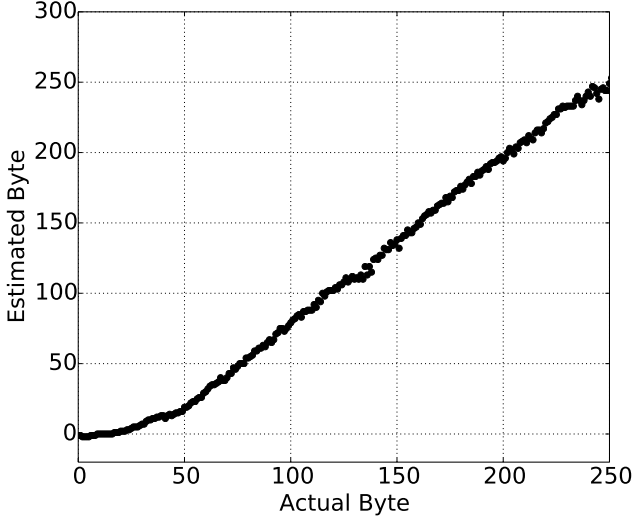Fig. 4. Timing Measurement for Nginx 1.6.2 over Wired LAN

Fig. 5. Observed Byte Estimation

C. Locate Safe Region

After we determine the average baseline delay, we redirect the `nelts` pointer to the region between address `0x7bfff73b9000` and `0x7efff73b9000`. As mentioned in the memory analysis, this is the range of the CPI safe region we know is guaranteed to be allocated, despite ASLR being enabled. We pick the the top of this region as the first value of our pointer.

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

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

In our experiments, page offset 4048 yielded the highest proportion of non-zero values, with 414 out of the 443 pages of libc having a nonzero byte at that offset. This would give our strategy an error rate of $1 - 414/443 = 6.5\%$. We note that we can reduce this number to 0 by scanning two bytes per page instead at offsets of our choice. In our experiments, if we scan the bytes at offsets 1272 and 1672 in any page of libc, one of these values is guaranteed to be nonzero. This reduces

our false positive rate at the cost of a factor of 2 in speed. In our experiments, we found that scanning 5 extra bytes in addition to the two signature bytes can yield 100% accuracy using 30 samples per byte and considering the error in byte estimation. Figure 6 illustrates the sum of the chosen offsets for our scan of zero pages leading up to libc. Note that we jump by the size of libc until we hit a non-zero page. The dot on the upper-right corner of the figure shows the first non-zero page.

In short, we scan $30*7 = 210$ bytes per size of libc to decide whether we are in libc or the safe region. Table I summarizes the number of false positives, i.e. the number of pages we estimate as nonzero, which are in fact 0. The number of data samples and estimation samples, and their respective fastest percentile used for calculation all affect the accuracy. Scanning 5 extra bytes (in addition to the two signature bytes for a page) and sampling 30 times per bytes yields an accuracy of 100% in our setup. As a result, the attack requires $(2+5)*2^{19}*30 = 7*2^{19}*30 = 110,100,480$ scans on average, which takes about 97 hours with our attack setup.

Once we have a pointer to a nonzero page in libc, we send more requests to read additional bytes with high accuracy to determine which page of libc we have found. Figure 7 illustrates that we can achieve high accuracy by sending $10,000$ samples per byte.

Despite the high accuracy, we have to account for errors in estimation. For this, we have developed a fuzzy $n-$gram matching algorithm that, given a sequence of noisy bytes, tells us the libc offset at which those bytes are located by comparing the estimated bytes with a local copy of libc. In determining zero and nonzero pages, we only collect 30 samples per byte as we do not need very accurate measurements. After landing in a nonzero page in libc, we do need more accurate measurements to identify our likely location. Our measurements show that $10,000$ samples are necessary to estimate each byte to within 20. We also determine that reading 70 bytes starting at page offset 3333 reliably is enough for the fuzzy $n-$gram matching algorithm to determine where exactly we are in libc. This offset was computed by looking at all contiguous byte sequences for every page of libc and choosing the one which required the fewest bytes to guarantee a unique match. This orientation inside libc incurs additional $70*10,000 = 700,000$ requests, which adds another hour to the total time of the attack for a total of 98 hours.

After identifying our exact location in libc, we know the exact base address of the safe region:

$$safe\_region\_address = libc\_base - 2^{42}$$

D. Fast Attack with Crashes

We can make the above attack faster by tolerating 12 crashes on average. The improved attack uses binary search as opposed to linear search to find libc after landing in the safe region as described in section IV-C. We also use an alternative strategy for discovering the base of libc. Instead of sampling individual pages, we continue the traversal until we observe a crash that

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

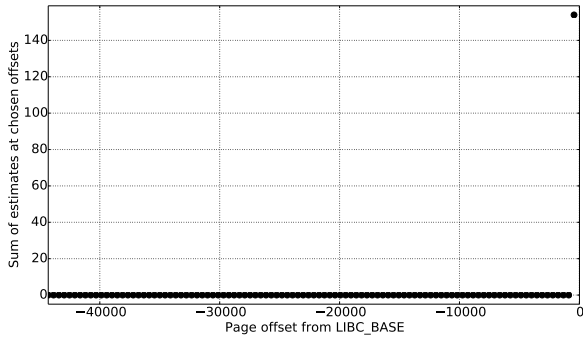Fig. 7. Actual Bytes Estimation of a Nonzero Page in LIBC.

Fig. 6. Estimation of Zero Pages in Safe Region.

indicates the location of the non-readable section of libc. This reveals the exact address of libc. In our setup, the binary search caused 11 crashes; discovering the base of libc required an additional 2 crashes.

E. Attack Safe Region

After finding the safe region, we then use the same data pointer overwrite to change the `read_handler` entry of the safe region. We then modify the base and bound of the code pointer to hold the location of the system call (`sysenter`). Since we can control what system call `sysenter` invokes by setting the proper values in the registers, finding `sysenter` allows us to implement a variety of practical payloads. After this, the attack can proceed simply by redirecting the code pointer to the start of a ROP chain that uses the system call. CPI does not prevent the redirection because its entry for the code pointer is already maliciously modified to accept the ROP chain.
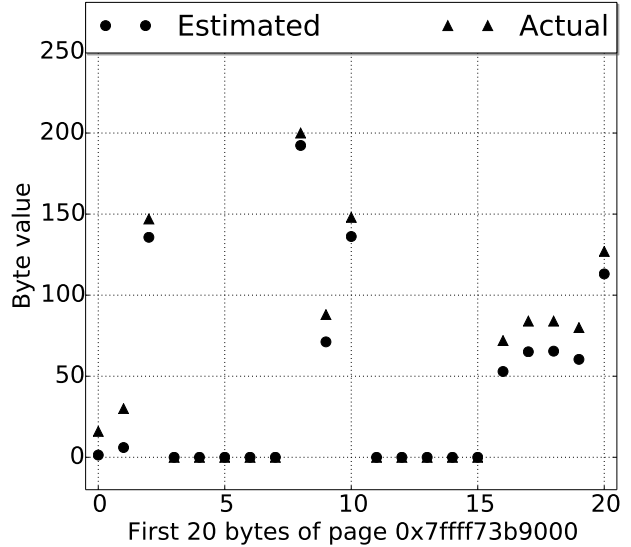
The entire crashing attack takes 6 seconds to complete.

F. Summary

In summary, we show a practical attack on a version of Nginx protected with CPI, ASLR and DEP. The attack uses a data pointer overwrite vulnerability to launch a timing side channel attack that can leak the safe region in 6 seconds with 13 observed crashes. Alternatively, this attack can be completed in 98 hours without any crashes.

VI. IMPLEMENTATION FLAWS OF CPI

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

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

In the default implementation (simpletable), the location of the table is stored in a static variable

(`__llvm__cpi_table`) which is not zeroed after its value is moved into the segment register. Thus, it is trivially available to an attacker by reading a fixed offset in the data segment. In the two alternate implementations, the location of the table is not zeroed because it is never protected by storage in the segment registers at all. Instead it is stored as a local variable. Once again, this is trivially vulnerable to an attack who can read process memory, and once disclosed will immediately compromise the CPI guarantees. Note that zeroing memory or registers is often difficult to perform correctly in C in the presence of optimizing compilers [44].

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

It is unclear exactly how the published CPI implementation intends to use the segment registers on 32-bit systems. The simpletable implementation, which uses the `%gs` register, warns that it is not supported on x86, although it compiles. We note that using the segment registers may conflict in Linux with thread-local storage (TLS), which uses the `%gs` register on x86-32 and the `%fs` register on x86-64 [18]. As mentioned, the default implementation, simpletable, does not support 32-bit systems, and the other implementations do not use the segment registers at all, a flaw noted previously, so currently this flaw is not easily exposed. A quick search of 32-bit libc, however, found almost 3000 instructions using the `%gs` register. Presumably this could be fixed by using the `%fs` register on 32-bit systems; however, we note that this may cause compatibility issues with applications expecting the `%fs` register to be free, such as Wine (which is explicitly noted in the Linux kernel source) [2].

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

VII. DISCUSSION

In this section we discuss some of the problematic CPI design assumptions and discuss possible fixes.

A. Design Assumptions

1) Enforcement Mechanisms: First, the authors of CPI focus on extraction and enforcement of safety checks, but they do not provide enough protection for their enforcement mechanisms. This is arguably a hard problem in security, but the effectiveness of defenses rely on such protections. In the published CPI implementation, protection of the safe region is very basic, relying on segmentation in the 32-bit architecture

and the size of the safe region in the 64-bit one. However, since the safe region is stored in the same address space to avoid performance expensive context switches, these protections are not enough and as illustrated in our attacks they are easy to bypass. Note that the motivation for techniques such as CPI is the fact that existing memory protection defenses such as ASLR are broken. Ironically, CPI itself relies on these defenses to protect its enforcement. For example, relying on randomization of locations to hide the safe region has many of the weaknesses of ASLR that we have illustrated.

2) Detecting Crashes: Second, it is assumed that leaking large parts of memory requires causing numerous crashes which can be detected using other mechanisms. This in fact is not correct. Although attacks such as Blind ROP [9] and brute force [51] do cause numerous crashes, it is also possible on current CPI implementations to avoid such crashes using side-channel attacks. The main reason for this is that in practice large number of pages are allocated and in fact, the entropy in the start address of a region is much smaller than its size. This allows an attacker to land correctly inside allocated space which makes the attack non-crashing. In fact, CPI's implementation exacerbates this problem by allocating a very large mmap region.

3) Memory Disclosure: Third, it is also implicitly assumed that large parts of memory cannot leak. Direct memory disclosure techniques may have some limitations. For example, they may be terminated by zero bytes or may be limited to areas adjacent to a buffer [54]. However, indirect leaks using dangling data pointers and timing or fault analysis attacks do not have these limitations and they can leak large parts of memory.

4) Memory Isolation: Fourth, the assumption that the safe region cannot leak because there is no pointer to it is incorrect. As we show in our attacks, random searching of the mmap region can be used to leak the safe region without requiring an explicit pointer into that region.

To summarize, the main weakness of CPI is its reliance on secrets which are kept in the same space as the process being protected. Arguably, this problem has contributed to the weaknesses of many other defenses as well [59, 51, 54, 47].

B. Patching CPI

Our attacks may immediately bring to mind a number of patch fixes to improve CPI. We considered several of these fixes here and discuss their effectiveness and limitations. Such fixes will increase the number of crashes necessary for successful attacks, but they cannot completely prevent attacks on architectures lacking segmentation (x86-64 and ARM).

1) Increase Safe Region Size: The first immediate idea is to randomize the location of the safe region base within an even larger mmap- allocated region. However, this provides no benefit: the safe region base address must be strictly greater than the beginning of the returned mmap region, effectively increasing the amount of wasted data in the large region but not preventing our side channel attack from simply continuing to scan until it finds the safe region. Moreover, an additional

register must be used to hide the offset and then additional instructions must be used to load the value from that register, add it to the safe region segment register, and then add the actual table offset. This can negatively impact performance.

2) Randomize Safe Region Location: The second fix can be to specify a fixed random address for the mmap allocation using `mmap_fixed`. This has the advantage that there will be much larger portions of non-mapped memory, raising the probability that an attack might scan through one of these regions and trigger a crash. However, without changing the size of the safe region an attacker will only need a small number of crashes in order to discover the randomized location. Moreover, this approach may pose portability problems; as the mmap man page states, "the availability of a specific address range cannot be guaranteed, in general." Platform-dependent ASLR techniques could exacerbate these problems. There are a number of other plausible attacks on this countermeasure:

- Unless the table spans a smaller range of virtual memory, attacks are still possible based on leaking the offsets and knowing the absolute minimum and maximum possible `mmap_fixed` addresses, which decrease the entropy of the safe region.
- Induce numerous heap allocations (at the threshold causing them to be backed by mmap) and leak their addresses. When the addresses jump by the size of the safe region, there is a high probability it has been found. This is similar to heap spraying techniques and would be particularly effective on systems employing strong heap randomization.
- Leak the addresses of any dynamically loaded libraries. If the new dynamically loaded library address increases over the previous dynamic library address by the size of the safe region, there is a high probability the region has been found.

3) Use Hash Function for Safe Region: The third fix can be to use the segment register as a key for a hash function into the safe region. This could introduce prohibitive performance penalties. It is also still vulnerable to attack as a fast hash function will not be cryptographically secure. This idea is similar to using cryptography mechanisms to secure CFI [35].

4) Reduce Safe Region Size: The fourth fix can be to make the safe region smaller. This is plausible, but note that if mmap is still contiguous an attacker can start from a mapped library and scan until they find the safe region, so this fix must be combined with a non-contiguous mmap. Moreover, making the safe region compact will also result in additional performance overhead (for example, if a hashtable is being used, there will be more hashtable collisions). A smaller safe region also runs a higher risk of running out of space to store "sensitive" pointers more easily.

In order to evaluate the viability of this proposed fix, we compiled and ran the C and C++ SPECint and SPECfp 2006 benchmarks [22] with several sizes of CPI hashtables on an Ubuntu 14.04.1 machine with 4GB RAM. All C benchmarks were compiled using the `-std=gnu89` flag (clang requires

this flag for `400.perlbench` to run). In our setup, no benchmark compiled with the CPI hashtable produced correct output on `400.perlbench`, `403.gcc` and `483.xalancbmk`.

Table II lists the overhead results for SPECint. NT in the table denotes "Not terminated after 8 hours". In this table, we have listed the performance of the default CPI hashtable size (2^{33}). Using a hashtable size of 2^{26}, CPI reports that it has run out of space in its hashtable (i.e. it has exceed a linear probing maximum limit) for `471.omnetpp` and `473.astar`. Using a hashtable size of 2^{20}, CPI runs out of space in the safe region for those tests, as well as `445.gobmk` and `464.h264ref`. The other tests incurred an average overhead of 17% with the worst case overhead of 131% for `471.omnetpp`. While in general decreasing the CPI hashtable size leads to a small performance increase, these performance overheads can still be impractically high for some real-world applications, particularly C++ applications like `471.omnetpp`.

Table III lists the overhead results for SPECfp. IR in the table denotes "Incorrect results." For SPECfp and a CPI hashtable size of 2^{26}, two benchmarks run out of space: `433.milc` and `447.dealII`. In addition, two other benchmarks return incorrect results: `450.soplex` and `453.povray`. The `453.povray` benchmark also returns incorrect results with CPI's default hashtable size.

TABLE II
SPECint 2006 Benchmark Performance by CPI flavor

Benchmark	No CPI	CPI simpletable	CPI hashtable
401.bzip2	848 sec	860 (1.42%)	845 (-0.35%)
429.mcf	519 sec	485 (-6.55%)	501 (-3.47%)
445.gobmk	712 sec	730 (2.53%)	722 (1.40%)
456.hmmer	673 sec	687 (2.08%)	680 (1.04%)
458.sjeng	808 sec	850 (5.20%)	811 (0.37%)
462.libquantum	636 sec	713 (12.11%)	706 (11.01%)
464.h264ref	830 sec	963 (16.02%)	950 (14.46%)
471.omnetpp	582 sec	1133 (94.67%)	1345 (131.10%)
473.astar	632 sec	685 (8.39%)	636 (0.63%)
400.perlbench	570 sec	NT	NT
403.gcc	485 sec	830 (5.99%)	NT
483.xalancbmk	423 sec	709 (67.61%)	NT

TABLE III
SPECfp 2006 Benchmark Performance by CPI flavor

Benchmark	No CPI	CPI simpletable	CPI hashtable
433.milc	696 sec	695 (-0.14%)	786 (12.9%)
444.namd	557 sec	571 (2.51%)	574 (3.05%)
447.dealII	435 sec	539 (23.9%)	540 (24.1%)
450.soplex	394 sec	403 (2.28%)	419 (6.34%)
453.povray	250 sec	IR	IR
470.lbm	668 sec	708 (5.98%)	705 (5.53%)
482.sphinx3	863 sec	832 (-3.59%)	852 (-1.27%)

To evaluate the effectiveness of a scheme which might dynamically expand and reduce the hashtable size to reduce the attack surface at the cost of an unknown performance penalty and loss of some real-time guarantees, we also ran the SPEC benchmarks over an instrumented hashtable implementation to discover the maximum number of keys concurrently

resident in the hashtable; our analysis showed this number to be 2^{23} entries, consuming 2^{28} bytes. However, some tests did not complete correctly unless the hashtable size was at least 2^{28} entries, consuming 2^{33} bytes. Without any other mmap allocations claiming address space, we expect $\frac{2^{46}}{2^{28}} = 2^{18}$ crashes with an expectation of 2^{17}, or $\frac{2^{46}}{2^{33}} = 2^{13}$ crashes with an expectation of 2^{12}. This seems to be a weak guarantee of the security of CPI on programs with large numbers of code pointers. For instance, a program with 2GB of memory in which only 10% of pointers are found to be sensitive using a CPI hashtable with a load factor of 25% would have a safe region of size $(2 * 10^9/8 * 8\% * 4 * 32$ bytes$)$. The expected number of crashes before identifying this region would be only slightly more than 2^{14}. This number means that the hashtable implementation of CPI is not effective for protecting against a local attacker and puts into question the guarantees it provides on any remote system that is not monitored by non-local logging. As a comparison, it is within an order of magnitude of the number of crashes incurred in the Blind ROP [9] attack.

5) Use Non Contiguous Randomized mmap: Finally, the fifth fix can be to use a non-contiguous, per-allocation randomized mmap. Such non-contiguous allocations are currently only available using customized kernels such as PaX [43]. However, even with non-contiguous allocations, the use of super pages for virtual memory can still create weaknesses. An attacker can force heap allocation of large objects, which use mmap directly to generate entries that reduce total entropy. Moreover, knowing the location of other libraries further reduces the entropy of the safe region because of its large size. As a result, such a technique must be combined with a reduction in safe region size to be viable. More accurate evaluation of the security and performance of such a fix would require an actual implementation which we leave to future work.

The lookuptable implementation of CPI (which was non-functional at the time of our evaluation) could support this approach by a design which randomly allocates the address of each subtable at runtime. This would result in a randomized scattering of the much smaller subtables across memory. There are, however, only $\frac{2^{46}}{32*2^{22}\text{entries}} = 2^{19}$ slots for the lookup table's subtable locations. The expectation for finding one of these is $\frac{2^{19}}{2K}$ crashes, where K is the number of new code pointers introduced that cause a separate subtable table to be allocated. If there are 2^5 such pointers (which would be the case for a 1GB process with at least one pointer across the address space), that number goes to 2^{13} crashes in expectation, which as previously argued does not provide strong security guarantees.

We argue that we can identify a subtable because of the recognizable CPI structure, and search it via direct/side-channel attacks. While we cannot modify any arbitrary code pointer, we believe that it is only a matter of time until an attacker discovers a code pointer that enables remote code execution.

VIII. Possible Countermeasures

In this section we discuss possible countermeasures against control hijacking attacks that use timing side channels for memory disclosure.

a) Memory Safety: Complete memory safety can defend against all control hijacking attacks, including the attack outline in this paper. Softbound with the CETS extensions [36] enforces complete spatial and temporal pointer safety albeit at a significant cost (up to 4x slowdown).

On the other hand, experience has shown that low overhead mechanisms that trade off security guarantees for performance (e.g., approximate [48] or partial [5] memory safety) eventually get bypassed [9, 52, 21, 11, 17].

Fortunately, hardware support can make complete memory-safety practical. For instance, Intel memory protection extensions (MPX) [25] can facilitate better enforcement of memory safety checks. Secondly, the fat-pointer scheme shows that hardware-based approaches can enforce spatial memory safety at very low overhead [32]. Tagged architectures and capability-based systems can also provide a possible direction for mitigating such attacks [58].

b) Randomization: One possible defense against timing channel attacks, such as the one outlined in this paper, is to continuously rerandomize the safe region and ASLR, before an attacker can disclose enough information about the memory layout to make an attack practical. One simple strategy is to use a worker pool model that is periodically re-randomized (i.e., not just on crashes) by restarting worker processes. Another approach is to perform runtime rerandomization [20] by migrating running process state.

Randomization techniques provide probabilistic guarantees that are significantly weaker than complete memory safety at low overhead. We note that any security mechanism that trades security guarantees for performance may be vulnerable to future attacks. This short term optimization for the sake of practicality is one reason for the numerous attacks on security systems [9, 52, 21, 11, 17].

c) Timing Side Channel Defense: One way to defeat attacks that use side channels to disclose memory is to remove execution timing differences. For example, timing channels can be removed by causing every execution (or path) to take the same amount of time. The obvious disadvantage of this approach is that average-case execution time now becomes worst-case execution time. This change in expected latency might be too costly for many systems. We note here that adding random delays to program execution cannot effectively protect against side channel attacks [19].

IX. Related Work

Memory corruption attacks have been used since the early 70's [6] and they still pose significant threats in modern environments [14]. Memory unsafe languages such as C/C++ are vulnerable to such attacks.

Complete memory safety techniques such as the SoftBound technique with its CETS extension [36] can mitigate memory corruption attacks, but they incur large overhead to the

execution (up to 4x slowdown). "fat-pointer" techniques such as CCured [37] and Cyclone [28] have also been proposed to provide spatial pointer safety, but they are not compatible with existing C codebases. Other efforts such as Cling [4], Memcheck [38], and AddressSanitizer [48] only provide temporal pointer safety to prevent dangling pointer bugs such as use-after-free. A number of hardware-enforced memory safety techniques have also been proposed including the Low-Fat pointer technique [32] and CHERI [58] which minimize the overhead of memory safety checks.

The high overhead of software-based complete memory safety has motivated weaker memory defenses that can be categorized into enforcement-based and randomization-based defenses. In enforcement-based defenses, certain correct code behavior that is usually extracted at compile-time is enforced at runtime to prevent memory corruption. In randomization-based defenses different aspects of the code or the execution environment are randomized to make successful attacks more difficult.

The randomization-based category includes address space layout randomization (ASLR) [43] and its medium-grained [30] and fine-grained variants [57]. Different ASLR implementations randomize the location of a subset of stack, heap, executable, and linked libraries at load time. Medium-grained ASLR techniques such as Address Space Layout Permutation [30] permutes the location of functions within libraries as well. Fine-grained forms of ASLR such as Binary Stirring [57] randomize the location of basic blocks within code. Other randomization-based defenses include in-place instruction rewriting such as ILR [23], code diversification using a randomizing compiler such as the multi-compiler technique [27], or Smashing the Gadgets technique [42]. Unfortunately, these defenses are vulnerable to information leakage (memory disclosure) attacks [54]. It has been shown that even one such vulnerability can be used repeatedly by an attacker to bypass even fine-grained forms of randomization [52]. Other randomization-based techniques include Genesis [60], Minestrone [29], or RISE [8] implement instruction set randomization using an emulation, instrumentation, or binary translation layer such as Valgrind [38], Strata [46], or Intel PIN [34] which in itself incurs a large overhead, sometimes as high as multiple times slowdown to the applications.

In the enforcement-based category, control flow integrity (CFI) [3] techniques are the most prominent ones. They enforce a compile-time extracted control flow graph (CFG) at runtime to prevent control hijacking attacks. Weaker forms of CFI have been implemented in CCFIR [61] and bin-CFI [62] which allow control transfers to any valid target as opposed to the exact ones, but such defenses have been shown to be vulnerable to carefully crafted control hijacking attacks that use those targets to implement their malicious intent [21]. The technique proposed by Backes et al. [7] prevents memory disclosure attacks by marking executable pages as non-readable. A recent technique [15] combines aspects of enforcement (non-readable memory) and randomization (fine-grained code randomization) to prevent memory disclosure attacks.

On the attack side, direct memory disclosure attacks have been known for many years [54]. Indirect memory leakage such as fault analysis attacks (using crash, non-crash signal) [9] or in general other forms of fault and timing analysis attacks [47] have more recently been studied.

Non-control data attacks [13], not prevented by CPI, can also be very strong in violating many security properties; however, since they are not within the threat model of CPI we leave their evaluation to future work.

X. Conclusion

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

XI. Acknowledgment

References

[1] Vulnerability summary for cve-2013-2028, 2013.

[2] Linux cross reference, 2014.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

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

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[6] J. P. Anderson. Computer security technology planning study. volume 2. Technical report, DTIC Document, 1972.

[7] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[8] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 281–289, New York, NY, USA, 2003. ACM.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[10] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proc. of the 6th ACM Symposium on Info., Computer and Comm. Security*, pages 30–40, 2011.

[11] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proc. of the 17th ACM CCS*, pages 559–572, 2010.

[13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security*, volume 5, 2005.

[14] X. Chen, D. Caselden, and M. Scott. New zero-day exploit targeting internet explorer versions 9 through 11 identified in targeted attacks, 2014.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy*, 2015.

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

[17] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[18] U. Drepper. Elf handling for thread-local storage, 2013.

[19] F. Durvaux, M. Renauld, F.-X. Standaert, L. v. O. tot Oldenzeel, and N. Veyrat-Charvillon. *Efficient removal of random delays from embedded software implementations using hidden markov models*. Springer, 2013.

[20] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012.

[21] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

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

[23] J. Hiser, A. Nguyen, M. Co, M. Hall, and J. Davidson. Ilr: Where'd my gadgets go. In *IEEE Symposium on Security and Privacy*, 2012.

[24] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, et al. An overview of the singularity project. 2005.

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

[26] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Diversifying the software stack using randomized nop insertion. In *Moving Target Defense*, pages 151–173. 2013.

[27] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. *Moving Target Defense*, pages 77–98, 2011.

[28] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[29] A. D. Keromytis, S. J. Stolfo, J. Yang, A. Stavrou, A. Ghosh, D. Engler, M. Dacier, M. Elder, and D. Kienzle. The minestrone architecture combining static and dynamic analysis techniques for software security. In *SysSec Workshop (SysSec), 2011 First*, pages 53–56. IEEE, 2011.

[30] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proc. of ACSAC'06*, pages 339–348. Ieee, 2006.

[31] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. 2014.

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

[33] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[35] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[36] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

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

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

[39] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding focus in the blur of moving-target techniques. *IEEE Security & Privacy*, 12(2):16–26, Mar 2014.

[40] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[41] OpenBSD. Openbsd 3.3, 2003.

[42] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[43] PaX. Pax address space layout randomization, 2003.

[44] C. Percival. How to zero a buffer, Sept. 2014.

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

[46] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 36–47. IEEE Computer Society, 2003.

[47] J. Seibert, H. Okhravi, and E. Soderstrom. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Nov 2014.

[48] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[49] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[50] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of ACM CCS*, pages 552–561, 2007.

[51] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. of ACM CCS*, pages 298–307, 2004.

[52] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[53] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proc. of EuroSec'09*, pages 1–8, 2009.

[54] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of EuroSec '09*, 2009.

[55] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[56] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proc. of RAID'11*, pages 121–141, 2011.

[57] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[58] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, M. Vadera, and K. Gudka. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*, 2015.

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

[60] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *Security & Privacy, IEEE*, 7(1):26–33, 2009.

[61] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

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