Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure

Enes Göktaş*, Benjamin Kollenda†, Philipp Koppe†, Erik Bosman*,
Georgios Portokalidis‡, Thorsten Holz†, Herbert Bos* and Cristiano Giuffrida*

* *Computer Science Institute, Vrije Universiteit Amsterdam, The Netherlands*
Email: e.goktas@vu.nl, erik@minemu.org, herbertb@cs.vu.nl, giuffrida@cs.vu.nl
† *Horst Görtz Institut for IT-Security (HGI), Ruhr-Universität Bochum, Germany*
Email: benjamin.kollenda@rub.de, philipp.koppe@rub.de, thorsten.holz@rub.de
‡ *Department of Computer Science, Stevens Institute of Technology*
Email: gportoka@stevens.edu

Abstract—**Address-space layout randomization is a well-established defense against code-reuse attacks. However, it can be completely bypassed by just-in-time code-reuse attacks that rely on information disclosure of code addresses via memory or side-channel exposure. To address this fundamental weakness, much recent research has focused on detecting and mitigating information disclosure. The assumption being that if we perfect such techniques, we will not only maintain layout secrecy but also stop code reuse.**

In this paper, we demonstrate that an advanced attacker can mount practical code-reuse attacks even in the complete absence of information disclosure. To this end, we present *Position-Independent Code-Reuse Attacks*, **a new class of code-reuse attacks relying on the** *relative* **rather than** *absolute* **location of code gadgets in memory. By means of** *memory massaging*, **the attacker first makes the victim program generate a** *rudimentary* **ROP payload (for instance, containing code pointers that target instructions "close" to relevant gadgets). Afterwards, the addresses in this payload are patched with small offsets via** *relative memory writes*. **To establish the practicality of such attacks, we present multiple Position-Independent ROP exploits against real-world software. After showing that we can bypass ASLR in current systems without requiring information disclosures, we evaluate the impact of our technique on other defenses, such as fine-grained ASLR, multi-variant execution, execute-only memory and re-randomization. We conclude by discussing potential mitigations.**

1. Introduction

No longer able to execute shellcode directly, today's exploits depend on reusing code from the original program to build a malicious payload. Code-reuse attacks (CRAs) chain existing code together by positioning the addresses of carefully selected code fragments, called *gadgets*, in memory. A hijacked code pointer is used to redirect execution to the first gadget, which uses one of the prepared address to execute the next gadget through an indirect call, jump, or return instruction, and so on. Address space layout randomization (ASLR) [1], and newer randomization techniques proposed in literature [2]–[14], mitigate CRAs by introducing uncertainty on the location of gadgets during exploitation. In response, modern exploits rely primarily on information disclosure vulnerabilities [15], [16] to exfiltrate code addresses, while other avenues of attack include using side-channels to reduce entropy [17], [18] and brute-forcing [19]. As a result, some of the most recent proposals have focused on preventing information disclosures altogether. Examples include execute-only memory [20]–[25], destructive code reads [26], [27], and multi-variant execution [28]–[30]. Others have focused on mitigating the effects of leaks through continuous code re-randomization [12], [31]–[36].

With all the recent work on defeating information disclosure, this paper attempts to answer the question: *Is it possible to perform code-reuse attacks even without information disclosure?* More specifically, is it possible to construct *position-independent payloads* that are resilient to ASLR and do not rely on side-channels? The answer, based on our work, is *yes*. We show that, while harder, it is in practice possible to construct payloads without ever learning a single code address—an approach that we term *position-independent code reuse*.

In current CRAs, the attacker needs to locate all the code fragments that will be part of his payload, prepare the payload in memory, and hijack the control flow to execute it. Not surprisingly, most advanced defenses today target the attackers' ability to locate gadgets. Approaches focusing on execute-only memory [20]–[25] and destructive code reads [26], [27] aim to prevent leaking pointers contained in code [15] and the on-the-fly disassembling of code to locate appropriate gadgets [16]. Multi-variant execution [28]–[30] aims to *detect* pointer leakage, while constant re-randomization [12], [31]–[36] of code aims to limit the "life-span" of leaked pointers. Fine-grained randomization [7], [23], occurring at the function- or basic block-level, also aim to limit the usefulness of leaked pointers by introducing more entropy. *How would these defenses fare against position-independent code reuse?* We show that approaches based on execute-only memory, destructive code-reads, and

multi-variant execution would fail to stop it. On the other hand, improved randomization techniques, even though not bulletproof, fare better at preventing this new type of code reuse.

The main contribution of this paper is the introduction of a new type of code-reuse attack that does not depend on information leaks. To define this new attack, we first challenge the concept that the attacker needs to explicitly control a region of memory, where a pre-constructed payload will be placed. In contrast, we show that careful *memory massaging* allows an attacker to make the *victim program* generate a rudimentary ROP payload "skeleton" for us. For instance, we are able to make the program call (and perhaps return from), a set of functions with return addresses that remain intact on the stack, possibly in stale function frames.

Second, we utilize partial pointer overwrites to "bump" the pointers in the massaged area, changing their least-significant bits. The partial overwrite enables us to redirect pointers to gadgets in the vicinity of the previously stored pointer, leaving the most significant bits untouched, which are usually the ones that change due to randomization techniques. This technique is inspired by past work on partial EIP overwrites [37], and notably partial pointer overwrites appeared in recent attacks in the wild [38].

With these extra degrees of freedom, interesting new attack possibilities arise. Specifically, if the program generates the code pointers in the ROP chain for us, we do not need to know the code addresses ourselves—we can just *use* them. If not, using partial pointer overwrites we may modify, say, the least significant byte of a code pointer, and target gadgets at an offset of up to 256 bytes relative to the indirect branch's original target. The resulting payload constitutes a position-independent ROP attack (PIROP) and works even in the absence of information disclosure capabilities.

To demonstrate that PIROP works against real-world applications, we present several proof-of-concept (PoC) attacks that completely bypass ASLR. Specifically, we introduce several PIROP exploits against Asterisk, a popular server which provides telephone communication services, and Mozilla Firefox to show the practical viability of the proposed technique. While these attacks are sophisticated and may only be available when the right vulnerability is present, they demonstrate that information leaks are not an indispensable component for CRAs against complex applications.

In addition, we evaluate the impact of PIROP on more advanced defenses such as fine-grained ASLR, multi-variant execution, execute-only memory, and re-randomization. We find that while some can be completely bypassed, others may still be effective, even if weaker than originally thought. For instance, our findings indicate that function-level randomization is not very effective against PIROP, unlike basic block-level randomization.

In summary, our contributions are the following:

- We introduce *Position-Independent Code-Reuse Attacks*, a novel technique that does not require information disclosure to bypass ASLR, ignoring defenses that focus on code readability and leakage prevention.

- We develop several *Position-Independent ROP* (PIROP) exploits against both servers (Asterisk) and browsers (Firefox) to demonstrate the practical feasibility of our technique. Recordings of PIROP exploits can be found at: https://www.vusec.net/projects/pirop.

- We evaluate the expressiveness of PIROP on several real-world programs and analyze how various CRA defenses fair against PIROP.

2. Technical Background

In this section, we provide background information on the techniques this research is encompassing.

Code reuse. In the absence of any memory defense in computer systems in the past, attackers directly injected shellcode through memory corruption vulnerabilities. With the widely deployed memory defense Data Execution Prevention (DEP), performing code-reuse attacks became the prominent way for exploiting memory corruption vulnerabilities. Code-reuse attacks allow attackers to achieve the same goals as with injected shellcode by executing *code chunks already available* in the program.

A requirement to achieve these goals is calling functions. With DEP in place, attackers are still able to call functions by means of the code-reuse technique called *return-into-libc* (RILC) [39]. Attackers write the memory address of the desired function on the stack along with the required arguments, and then trigger the execution of this function by letting the program change the control flow to the function with a *return* instruction. By preparing multiple RILC call frames on the stack, multiple functions can be chained for execution.

With the emergence of 64-bit architectures, RILC attacks have become less prominent, given that the calling conventions on 64-bit architectures is different than in 32-bit architectures: to speed up performance, the first few arguments of functions are transferred through processor registers instead of the stack memory. Consequently, performing 64-bit RILC attacks is non-trivial because it is difficult to control the necessary argument registers prior to the control-flow diversion. *Return-Oriented Programming* (ROP) [40] can overcome this challenge and has de-facto replaced RILC in practical attacks.

In ROP, so-called *gadgets*, small code chunks that end in *return* instructions, are chained together to perform a desired operation. An attacker writes the ROP payload, i.e., code pointers (gadget addresses) and data operands, in memory and then tricks the program into considering this payload as its new stack. The return instructions then consume the code pointers, and the other instructions in the gadgets consume the data operands on the stack. The expressive power of ROP is due to every byte of the original code being targetable. This enables Turing-complete computations.

In response to several defense techniques, other flavors of code reuse similar to ROP have evolved. For example, *Jump-Oriented Programming* (JOP) [41] can be used to bypass some execution monitoring defenses [42] and *Counterfeit Object-Oriented Programming* (COOP) [43] can

be used to bypass some Control-Flow Integrity (CFI) [44] defenses. In this paper, we introduce a novel variant of code reuse named PIROP that, in turn, seeks to bypass both randomization and information disclosure defenses.

Besides the ability to divert the program's control flow to the first gadget to start off the chain, attackers today need two other capabilities. First, to construct the chains, attackers need control over a region of memory such as the stack so that they can explicitly store the payload (consisting of code pointers and operands). Second, they need to determine the appropriate set of code pointers to store in this region. Depending on the defenses deployed, the set may be more or less restricted. For instance, in the absence of additional measures such as control-flow integrity (CFI) [44], the code pointers may target any byte in the text segment. CFI, shadow stacks, and similar measures reduce the set of targetable functions, e.g., to legitimate callsites and function entry points. Whatever the available set of targets, the attacks must locate them and place them in the right locations—typically the stack.

ASLR. Address-Space Layout Randomization (ASLR) [1] is a lightweight defense against code-reuse attacks. The goal is to make the locations of the code gadgets unknown to the attacker. The popular and widely deployed version of ASLR randomizes the base location of loaded modules. However, this is vulnerable to information disclosure, given that a single pointer leak can compromise the defense [15].

Weaknesses in ASLR motivated the development of finer-grained randomization techniques. Such techniques typically permute objects in the text segment or randomize the padding between such objects. Existing permutation-based techniques reorganize individual code pages [3], [6], [9], [14], individual functions [2], [4], [5], [12], [20], [23], or intra-function elements (e.g., basic blocks [8], instructions [7], [10], or data flow [7], [20], [23], [25]). Similarly existing padding-based techniques typically operate at the function level [4], [5], [12] or within individual functions (e.g., randomly padding basic blocks [12], [13], [22], [23], [25], [36] or instructions [10], [11], [45]).

Defenses against information disclosure. Although ASLR can be bypassed with a single pointer leak, even fine-grained ASLR techniques are vulnerable to *advanced* information disclosure attacks in which attackers can *repeatedly* leak information at requested locations in memory [16]. To further harden randomization solutions, researchers have proposed several defenses against advanced information disclosure.

Several Multi-Variant eXecution (MVX) [28]–[30] solutions detect information leaks by comparing execution results of multiple versions of a running process. Due to different randomized address-space layouts of the processes, the execution observably diverges whenever an attacker tries to leak information from the memory space. This strategy is alone insufficient against information disclosure attacks that rely on side channels. However, many MVX systems also enforce Address Space Partitioning (ASP, or non-overlapping address spaces) [28], [29], [46]–[50] across

variants to ensure that, even if code pointers are somehow leaked, an attacker cannot rely on such pointers to mount code reuse without being detected [50].

Execute-only-Memory (XoM) is another defense against advanced information disclosure. XoM defenses disable the read permissions on the code using either software- [20], [21] or hardware-based [22]–[25] techniques. Similar to XoM, destructive code read [26], [27] defenses make the code sections execute-only but at the same time support reading data that is intermingled with the code, e.g., jump tables. The key idea is to ensure that every byte read from the code sections can no longer be used to execute instructions.

Another popular approach is to periodically operate re-randomization during the execution. This strategy counters information disclosure by invalidating all the previously leaked code pointers. Existing re-randomization solutions either operate re-randomization at predetermined time intervals [12], [31], [32] or every time particular events mark opportunities for attacks (e.g., crashes [33], I/O operations [35], syscalls [36], or control-flow decisions [34]).

3. Bypassing Modern Defenses via PIROP

The defenses discussed in the previous section have one goal in common: by randomizing the location of code in memory and preventing information disclosure, they assume that code-reuse attacks are no longer possible. In the following, we challenge this assumption and demonstrate a new class of code-reuse attacks named *position-independent code-reuse attacks*, which can operate even in complete absence of information disclosure for both code and data sections.

3.1. Threat Model

Throughout the rest of this paper, we use the following threat model, similar to the one considered in prior work [16]:

- *Data execution prevention:* We assume that DEP is in place, ensuring a memory page can be marked either writable or executable but not both, a standard technique deployed on most computer systems nowadays.
- *Address space layout randomization:* We assume that ASLR is in place, randomizing the base address of binary and library images, as well as data memory, like stack, heap, and other (memory) mapped regions. Also a standard technique on most computer systems today.
- *Out-of-bounds memory write:* We assume that the target application contains a vulnerability that enables the attacker to corrupt memory by writing outside the boundaries of an existing buffer or object. Specifically, the vulnerability should allow an attacker to operate a non-linear relative memory write, that is writing at an *attacker-controlled offset from* a program pointer. Multiple vulnerability types can provide such "functionality", like out-of-bounds accesses on arrays, type

confusion, and use-after-frees (UAF). The vulnerability should also allow writing values smaller than the word size used by the program (e.g., 1 or 2 bytes, or even just bits). Non-linear memory write bugs can be as simple as the example shown in Listing 1 below and they have become more common [38], [51], [52]: according to an analysis by Microsoft, about 60% of the heap corruption vulnerabilities detected in the last three years represent such non-linear vulnerabilities [53].

- *Control-flow hijacking*: The previous overwrite bug can be used to corrupt a code pointer and hijack control flow, as is commonly the case in such attacks.
- *Known application*: We assume that the attacker has a copy of the targeted application, but the exact code layout of the application running at the target is not known.

Listing 1: Example of small, relative overwrite bug.

```
/* index is controlled by the attacker */
void write_array(char *array, int index, char byte)
{
  ...
  array[index] = byte;
  ...
}
```

Most importantly, we do not assume a memory disclosure vulnerability that enables an attacker to leak code pointers, which is a fundamental requirement in existing attacks [16], [43]. If the attacker can reliably leak code pointers, all the deployed randomization schemes are no longer effective. Effectively, we assume an attacker is unable to exploit disclosed information due to:

1) absence of information disclosure vulnerabilities (ideal case)
2) presence of defenses that actively detect information leaks (e.g., MVX), or
3) techniques that mitigate the impact of information disclosure via XoM or destructive code reads, raising the bar against traditional code-reuse attacks.

3.2. Position-independent Code Reuse

A common assumption in the literature is that randomization techniques combined with perfect information disclosure defenses can stop code-reuse attacks [50]. We now introduce a new way of performing code-reuse attacks called *Position-Independent Return-Oriented Programming* (PIROP) that can bypass these protection mechanisms. PIROP's essence is that disclosing information to derandomize the randomization may be unnecessary and a *generative* approach to code reuse is possible. A PIROP attack essentially *reuses* the *randomized* code pointers produced by the program without ever leaking them.

A PIROP attack can be conceptually split in four steps. The first step, and the most crucial one, is *stack massaging*, which forces the victim program to generate a rudimentary ROP payload on the stack on the attacker behalf. The second and third steps rely on a *relative memory write* primitive

to patch the code pointers and data operands in the ROP payload (as necessary). The last step is to make the program execute the prepared ROP payload for actual exploitation.

(i) Stack massaging. In code-reuse attacks, the exploitation payload consists of code pointers and data operands. Traditionally, an attacker first discloses the code pointers, i.e., locations of the gadgets, and then creates a chain of them by carefully placing the code pointers (along with data operands) in a specific order. Afterwards, the attacker forces the victim program to write this entire ROP payload in memory. Since in our threat model disclosing code pointers is no longer possible, an attacker can no longer craft a ROP payload with traditional techniques.

The goal of stack massaging is to overcome the absence of information disclosure capabilities by forcing the program to create the intended ROP payload on behalf of the attacker. By carefully making the program process certain attacker-controlled inputs, she can massage the stack and have the program itself place the desired (randomized) code pointers and data operands into the right position in memory.

While the proposed technique could be, in principle, generalized to arbitrary memory massaging and code-reuse attacks, the stack is an ideal means to lure the program into generating a code-reuse payload for us, since programs already continuously and nearly contiguously place code pointers (i.e., return addresses) on the stack. At every function call, the return address is pushed on the stack and some temporary space is also reserved for the variables of the called function. The more nested and recursive function calls are made, the more code pointers are active on the stack. Once the program returns from these functions, the code pointers will be left stale on the stack. Subsequently, the program can be stimulated to produce code pointers on the stack again with a different input to the program. An interesting point to note for the stack is that reserved variable locations are not always initialized and used, which means that, if surgically placed code pointers can survive multiple stack massaging steps (i.e., not overwritten in subsequent program inputs), the stack will contain a collection of code pointers from different attacker-controlled code paths. In other words, the stack will contain an attacker-controlled ROP payload written in memory *without disclosing any information*.

During the stack massaging step, the attacker first determines which code pointers and data can be massaged on the stack by executing the vulnerable input and variations. The objective is to find massageable code pointers that can provide gadgets with critical operations such as executing system calls and stack pivoting. Next, the attacker seeks to align the code pointers and data on the stack such that these can be patched where necessary. To find suitable alignments, the attacker needs to find inputs that provide fine-grained control over the callstack layout. To reduce the input-exploration space, the attacker can use simple heuristics, such as focusing on input-controlled recursive functions, input-controlled allocas, input-controlled JIT-ted stack frames, or simply locating different inputs that yield different callstack depths. To craft our exploits, we

used such heuristics to aid our manual analysis, but the approach could be automated to reduce the exploitation effort. We leave fully automatic PIROP exploit generation as future work.

(ii) Code pointer patching. The code pointers in the generated ROP payload in memory point each to executable code gadgets. In an ideal case, the gadgets in the ROP payload may be usable as is in the exploit. Although this is perhaps rare in practice, in such a case the code pointers can be left unpatched to mount a *patchless* PIROP exploit. The code pointers can also be left unpatched when the corresponding gadgets are neither harmful nor functional in which case they will just behave as no-operation gadgets.

However, sometimes the code pointers may be harmful (i.e., it can hinder the exploitation) or the exploitation may require a certain operation that is not easily available in the code gadgets reachable from the massaged code pointers. To resolve these issues, code pointers can be patched in place in the ROP payload to point to different code locations. If we only patch a few selected bits in the code pointers using a *relative memory write* primitive (e.g., a non-linear buffer overflow [54]), we can redirect each patched pointer to a location *relative* to the original target. This is to ensure the ROP payload remains randomization-agnostic, while expanding the gadget set at our disposal.

The extent to which we can expand the gadget set depends on the capabilities of the relative memory write primitive and on those of the underlying randomization and information disclosure defenses. For example, a memory write primitive that can overwrite a memory location after every N bytes is not useful if the writable bytes do not overlap with the actual bytes that we seek to overwrite. A non-linear memory corruption primitive that can *increment* or *decrement* massaged code pointers with specific offsets would be the ideal case, but, in practice, as we will show, the ability to overwrite single bytes or bits is sufficient for our purposes.

The level of randomization defines to what degree we can patch the code pointers. For performance reasons, randomization techniques generally provide no or very little entropy in the least significant bits of code pointers. Hence, we can simply focus code pointer patching on these bits to expand our gadget set. The number of reachable code locations directly depends on the number of low-order bits without entropy, as dictated by the randomization granularity.

As an example, assume a randomization technique that leaves the 8 least significant bits of code pointers non-randomized. We also assume that the attacker knows to which code offset in the corresponding module, i.e., executable or library, the code pointers belong, because she can control the input to the program and knows when and where the corresponding code pointers of the code offsets are produced. Given a target code pointer, for example `0x..7400`, the attacker can rely on a relative memory write primitive to patch this code pointer to any value between `0x..7400` and `0x..74FF`. If the primitive is only limited to setting the least significant 2 bits to 1, an attacker can only round up the code pointer to one where both bits are set to 1.

(iii) Operand patching. As in a traditional ROP attack, the code gadgets executed in the exploitation step operate on data that is carefully prepared in the ROP payload. This data is intermingled in the payload with the code pointers. Note that when the stack is being massaged, the payload area between the code pointers will be filled up with data. In the best case for an attacker, she can set this data directly by controlling the input to the program. If setting the data accordingly is not possible or the payload also contains randomized data pointers that need patching, she can use a memory write primitive to set up the data.

As observed earlier, the weaker the memory write primitive, the more difficult the exploitation. That is, a weaker write primitive might require executing more gadgets just to get the right data, which will also require setting up a larger ROP payload with more gadgets. Fortunately, since a program execution continuously operates on pointer and non-pointer data, reusable data operands are also often found on the stack.

(iv) ROP payload execution. Once the fully prepared ROP payload is in place, the program's execution has to be directed to it to finalize the exploit. Altering the normal execution of a program is done by changing the target of an indirect branch instruction, which can be a `call*`, a `jmp*` or a `ret` instruction. An attacker has to lure the program into feeding an unexpected target to one of these instructions to divert the control flow. This can be also done through a position-independent memory corruption vulnerability.

For example, to alter the target of a `call*` or `jmp*` instruction, a *use-after-free* vulnerability can be used in absence of information disclosure. If the set of targets that can be fed to these indirect instructions is not satisfying, i.e., does not change the control flow accordingly, a relative memory write primitive could be used to further alter the targets and alter the control flow as intended.

In contrast, altering the target of a `ret` instruction has to be done on the stack, where the targets of such instructions (i.e., return addresses) are located. For example, an attacker can overwrite the least significant byte of a return address on the stack using a stack-based buffer overflow to start off the ROP chain and finalize the exploit.

4. Proof-of-Concept Attacks

In this section, we validate PIROP by describing several proof-of-concept attacks on Asterisk, a popular server for telephony communication, and on Mozilla Firefox. Moreover, we argue that attacks in the wild are moving towards adopting a similar (albeit less general) approach— for instance, Chris Evans' attack on the FLIC gstreamer decoder [38]. Collectively, they show that PIROP (1) enables different types of exploitation, (2) requires a small set of gadgets, and (3) can be applied to different real-world applications, both clients and servers, and different defenses.

We initially present all our position-independent attacks assuming ASLR or other page-granular randomization vari-

ants [9]. In other words, the content of a page is always the same, no matter where the module is in memory and the 12 least significant bits are fixed. Thus, when we know one code pointer, we may look for suitable gadgets close to it—subject to the restrictions that apply to the available memory write primitive. In later sections, we will revisit this assumption and evaluate the effectiveness of our PIROP attacks against finer-grained ASLR techniques.

4.1. Attacks on the Asterisk Server

The first case study targets version v1.8.10.1 of Asterisk and CVE-2012-5976 [52] on 64-bit Ubuntu 12.04 LTS.

Vulnerability. The vulnerability allows an attacker to control the size parameter of an `alloca` function call through the *Content-length* header field in an HTTP request to the manager on the server. Without further bounds checking, the server subtracts this size from the stack pointer by means of `alloca`. Asterisk handles every HTTP request with a new thread with its own stack and providing a size larger than `0x7c000` (Asterisk's default stacksize on 64-bit architectures[1]) allows the attacker to target the stack of another thread. An example of the request is:

```
POST /asterisk/manager HTTP/1.0
Host: asterisk.example.com
Content-length: 507916          <-- large value

The server will  write this sentence in the
memory region allocated with the alloca() function.
```

We use this vulnerability both for *stack massaging*, and for a byte-level *write primitive* for the stack. Stack massaging occurs when we provide multiple requests with different content-length fields to make the program fill the stack with code pointers in the form of return addresses at different offsets. We obtain our write primitive similarly, by providing content which the server subsequently stores in the allocated memory at a desired offset.

4.1.1. Launching a shell with PIROP gadgets.
In the first exploit, we launch a shell on the server, from a client connected via the network, using only PIROP gadgets. Later, we will show an alternative attack that does code injection and requires fewer unique code pointers to be patched.

Stack massaging. To massage the stack and populate it with code pointers, we repeatedly trigger the `alloca` vulnerability while feeding it different sizes through the *Content-length* field of multiple requests. By subtracting the stack pointer with the given size, `alloca` creates a stack buffer to contain the request's content data. Subsequently, the program processes the remaining parts of the request which includes calling several functions. These function calls spill return addresses at the *adjusted* stack pointer. Depending on whether the request carries data, the return addresses differ. Specifically, in the absence of data, the program spills 6 return addresses, and 3 return addresses otherwise.

1. or 0x3c000 on 32-bit architectures

To exploit the system, we patch the return addresses to make them point to interesting gadgets at a relative offset 'close' to the original address. Specifically, since the 12 least significant bits do not vary under page-level randomization, we can patch them in any way we see fit. We therefore use the byte-level write primitive to overwrite the least significant byte of a return address. The primitive does not permit us to overwrite the second byte, because 4 of its bits are randomized. The obvious question is whether the remaining set of gadgets is sufficient for attackers.

To answer this question, we perform gadget analysis on the available code pointers. To minimize side effects and other complexity, we restrict our gadgets to at most 15 instructions that may however include conditional jumps. Since the patch ranges are small, we consider call instructions also as valid instructions in gadgets as long as they call functions that are leaf nodes, i.e. functions calling not any other function, to further increase the gadget space. We call such call instructions *leaf calls*.

Launching a shell is only possible if we have gadgets that can execute a program (so we can start `/bin/sh` to begin with), and also gadgets that can prepare the arguments. For the former, we found a syscall gadget that we use to execute a `dup2` system call and an `execve` system call, while for the latter we found a pair of gadgets to move a value from the stack into `rax` (which contains the syscall number) in two steps, and also simple gadgets that pop `rdi` and `rsi` (the first and second arguments) from the stack. We set the third argument `rdx`, required by the `execve` system call, to 0 by moving the return value of the call to `dup2` (which gets stored in `rax`) into `rdx`.

We use the `dup2` system call to duplicate a socket file descriptor into the standard input file descriptor of Asterisk for ensuring that the shell can receive commands through the network. Initially, Asterisk's standard input and output are set to `/dev/null`. Since `execve` preserves these, without `dup2`, the launched shell would have no way to get input and would exit. Because the file descriptor numbers are unknown, we first saturate the connection limit of Asterisk so that we can randomly pick a socket file descriptor number and later probe for the connection that feeds input to the shell. The complete exploit uses a total of 1 non-patched and 24 patched code pointers (gadgets), for which we massaged the stack by means of 9 manipulated calls to `alloca`. 3 patched code pointers lead to gadgets with a leaf call.

Code pointer patching. After making the program spill the code pointers on the stack, we patch their least significant bytes so they will point to the picked gadgets at a small offset from the original addresses. However, our byte-level write primitive is contiguous and if we patch a code pointer, everything preceding it will be overwritten, including the other code pointers.

To remedy the problem, we create a memory write primitive that resembles a non-linear memory write primitive—capable of writing at any desired offset in an area of code pointers without spoiling any of the others. By sending the content data of the request byte-by-byte over the network, we can essentially pause any time without tearing down the

connection. When we continue to send the data, Asterisk continues writing at the paused offset within the allocated buffer. We achieve the desired memory write primitive as follows: (1) we create a connection C_x, which causes Asterisk to allocate a new thread T_x with a fresh stack S_x; (2) we repeat the process and create a second connection C_y, which causes a fresh thread stack S_y to be allocated lower than stack S_x in the address space; (3) through C_x, we send an *incomplete* request that triggers the `alloca` vulnerability to make T_x's stack pointer point into stack S_y and writes the *partial* content data into the *allocated* buffer; (4) we massage our rudimentary ROP payload on stack S_y; (5) finally, we send the remaining data of the delayed request to overwrite the data within the rudimentary ROP payload. By controlling the size of the content data partially sent with the incomplete request, we can control the offset at which the remaining data gets written within the payload. To affect multiple distinct offsets, we send multiple partial requests over different connections.

Operand patching. Another important step in PIROP is the patching of the data operands consumed by the gadgets. We use the memory write primitive as we did for code pointer patching and patch up the arguments of the `dup2` system call to small file descriptor numbers and the second argument of the `execve` system call to 0 (`NULL`). We set the third argument of `execve` through the return value of `dup2`, which is 0.

The more challenging part is setting the first argument of `execve`, which should be a pointer to the program name string "/bin/sh". We can write the string with the write primitive. However, we cannot write the string pointer, since we do not know its memory address. Fortunately, during memory massaging, the program also spills heap and stack pointers on the stack including a stack pointer to the `alloca`-allocated buffer, which we can use for our purposes. We preserve this buffer pointer by ensuring that it aligns with unused gaps in subsequent massaging steps.

ROP payload execution. To execute the ROP payload the stack pointer has to point to the first gadget. We ensure that a thread's stack pointer points to the first gadget by massaging and patching the payload on this particular thread. This thread belongs to a connection that is waiting for its first request. After the ROP payload has been finalized on this thread's stack, sending its awaited first request starts off the gadget chain because the payload is aligned such that its first gadget is at the thread's stack pointer.

4.1.2. Minimizing pointer patching. To show the effectiveness of PIROP on stronger defenses than page-granular randomization, we showcase another attack on Asterisk with even more restrictions. Specifically, we restrict our pointer patching strategy to (i) minimize the number of patched gadgets and (ii) ensure each patched gadget always remains in the same function as the original code pointer spilled on the stack by the program. This is to showcase how PIROP fares against fine-grained, function-level randomization. Under these restrictions, we demonstrate a more advanced exploit on Asterisk in which we perform code injection,

disable the DEP protection through the `mprotect` system call, and execute the shellcode. Like in the previous attack, the shellcode starts a shell through `execve` and `dup2`.

The exploit first makes the heap readable, writable and executable, and then copies the shellcode in chunks to the executable heap. Like in the previous attack, we also utilize the unused gaps and the constructed memory write primitive to patch the code pointers and data operands. We also apply the same technique for control-flow hijacking. In this exploit, we used 16 iterations for the massaging and just 5 different original code pointers, two of which did not require any modifications whatsoever and could be used in a patchless fashion in the final ROP payload. Eventually, we used 21 non-patched and 14 patched code pointers (gadgets) in this attack, in which 1 patched code pointer led to a gadget with a leaf call. We also noticed a number of other patchless gadgets available on the stack that point to PLT stubs in *glibc* (e.g., *free*), which suggests that a fully patchless exploit with less ambitious goals (e.g., injecting a *use-after-free* vulnerability in the program) may be also possible.

4.2. Attacks on Firefox

To substantiate our approach, we also show that other types of programs such as browsers are vulnerable to PIROP. For this demonstration, we adopt a vulnerability similar to CVE-2016-1977 [51] into Mozilla Firefox 50.0.1, which allows a stack-based out-of-bound bit set. In the original vulnerability the bit set is limited to a range of 256 byte. For our proof-of-concept exploit we only extended the reachable memory region not to require additional effort to further escalate the relative write primitive. While not as unrestrained as the vulnerability used in Asterisk, the vulnerability still allows an attacker to patch pointers and operands in the ROP payload and can also effect patching writes at the bit granularity.

Vulnerability. The vulnerability does not reside in a component exclusive to Firefox, but instead is located inside the font library Graphite2 [55] which is used to render complex smart fonts. A key feature of Graphite2 is allowing fonts to define special handling for rendering complex glyphs. To enable this feature, it is possible to embed a basic script in the form of bytecode into the font, which is interpreted at runtime. The corresponding scripting language is called *Graphite Description Language* (GDL) [56]. The features of GDL allow for example exchanging glyphs for other glyphs of the same font or combining parts of them. Commonly this is used to represent complex combinations of characters that would otherwise require an individual glyph for every combination. Providing a smart font rendering allows supporting a large variety of complex writing systems.

The GDL is compiled to bytecode and embedded into the font by the Graphite2 tool chain. When such a font is loaded, it is passed as a buffer to the library which processes it and allows characters to be rendered with it afterwards. The interpretation is performed by a stack-based VM included with the Graphite2 library. During the initial

processing, different checks are performed to ensure the validity of the bytecode. One of these checks calls the `Machine::Code::decoder::analysis::set_ref` function which does not properly check the bounds of a buffer. As this buffer is allocated on the stack, this provides us with a basic relative memory write primitive. However, as the code only allows flags to be set and not cleared, we are also limited to setting bits in the target byte. This is still enough to change the target of a return address on the stack. Also it allows us to change individual bits instead of a whole byte when compared to the Asterisk vulnerability, giving us more precise control to bypass more fine-grained randomization defenses.

Stack massaging. In contrast to the Asterisk exploit, we do not possess the ability to extend the stack by an arbitrary length using `alloca`. Therefore we cannot bridge the gap to other thread's stacks and change values in them. Thus, our patching and execution steps must happen within the same stack. Also the stack-relative bit set is limited to writing after the stack-allocated array. This still allows targeting return addresses in stack frames generated earlier in the call chain. Hence, we can leverage return addresses of any functions that lead to the execution of the vulnerability. In addition, the design of the Firefox JavaScript engines gives us a great amount of flexibility: we can place a wide variety of different return addresses on the stack if we direct JavaScript execution along different paths we control. For this purpose, we used detours such as the `eval` function and JavaScript callbacks. Furthermore, the JavaScript engine reuses the native CPU stack to implement the JavaScript stack. This means that, whenever we trigger a call in JavaScript, a return address to the appropriate interpreter entry point is placed on the stack. If JIT-compiled code is used, this return address is replaced by the actual address of the generated code triggering the font load. We can use this property to create recursive code that places a high number of return addresses on the stack. Notably, we can achieve full stack massaging capabilities, as we have precise control over the offsets between the return addresses on the stack. In particular, we can add multiples of the native word width by leveraging the JavaScript call stack and passing as many arguments as we need to self-controlled JavaScript functions.

Code pointer patching. Once we have forced the browser to place the necessary code pointers at the right locations in the stack, we patch the resulting ROP payload as necessary. For this purpose, we force the load of a specifically crafted font. During the loading of the font, the vulnerable function will be executed and set the bits we indirectly specify via the embedded GDL. It should be noted that we are not limited to a single write. The GDL environment allows us to perform multiple patching operations during a single font loading. Using this method, we can patch as many return addresses as needed and thus construct more complex gadget chains.

Argument preparation. The next step is to patch the actual operands passed to the gadgets. One possibility is to patch existing data already spilled in the ROP payload. However, as we can control arguments being placed on the native stack from JavaScript, we can place the correct operands ourselves. The number of arguments that are passed to a function in JavaScript are placed on the native stack in double-precision floating-point format of 64-bit length. We inverted the encoding in order to place arbitrary values on the native stack. It should be noted that there are some minor limitations, i.e., we cannot write every number due to floating point error handling. We use this mechanism to place immediate operands on the stack that are consumed by our gadgets, and most importantly, we use it to prepare the required function arguments. To support data pointer operands, we can patch existing (randomized) data pointers spilled on the stack by following the same strategy adopted for code pointers.

ROP payload execution. After finalizing the stack with the gadget addresses and the corresponding operands, we can trigger the chain by simply letting the font loading return as normal—assuming the font loading succeeds. We then return from our JavaScript function. Thus, the stack deconstructs until it reaches the first return address we tampered with, which redirects control to our ROP chain. Initially, we compute the location of our argument string on the stack. We achieve this by leveraging multiple gadgets that add the correct offset to a register, which coincidentally contains a close stack address. We then transfer this value to `rdx` and use another gadget to place the syscall number of the system function in `rax`. Finally, we use a syscall gadget to invoke the system function. It should be noted that we can easily prepare long strings in order to execute arbitrary command payloads on the target machine.

5. Evaluation

To demonstrate the expressiveness of PIROP, we evaluate the availability of gadgets in Asterisk 1.8.10.1, Apache 2.4.25, Nginx 1.20.2, Lighttpd 1.4.45 and Firefox 50.0.1.

For the gadget analysis, we extended the Ropper [57] multi- architecture gadget analysis tool to support a user-defined, initial address set and recursive disassembly. This enables us to search and utilize novel gadget types containing direct unconditional and conditional branches as well as direct calls to leaf functions, e.g. functions without calls.

We collect the potentially targetable code locations in the servers by running their test suites and tracing the servers' execution with the GDB debugger. Every time a new request arrives, we store a snapshot of the stack. Next, we clear the lower part of the stack by writing dummy values. Dummy values help us identify unused locations, i.e., gaps on the stack. As explained previously, gaps are valuable for stack massaging, because we can fill them with code or data pointers using other requests. For every instance of the server, we also store a snapshot of the loaded libraries. For Firefox we use a similar method, but limit our analysis for simplicity on callstacks that are directly related to the vulnerability we use in our exploit. We break on every execution of the vulnerable font loading function and save the callstack leading to its execution. By modifying the JavaScript code leading up to the font load we created

different paths resulting in a bigger set of potential gadgets. Unlike in the proof-of-concept Firefox exploit, we are not aiming to get specific gadgets in the JIT-compiled code during this process. Finally, we combine the snapshots of the stacks with the loaded libraries and perform the gadget analysis with the extended Ropper tool. Table 1, shows the number of stack dumps processed.

Application	#Stack dumps
Asterisk 1.8.10.1	131
Httpd 2.4.25	216
Nginx 1.20.2	507
Lighttpd 1.4.45	208
Firefox 50.0.1	47

TABLE 1: Analyzed stack dumps for the applications considered.

Gadget lengths. Subject to the level of randomization, PIROP patches the code pointers spilled by the program to make them point to the gadget at a relative offset from the original address. In Fig 1a and 1b, we show the number of gadgets available, with different lengths, respectively, when an attacker has a byte level memory write primitive as in the Asterisk exploit or a 4-bit memory write primitive as in the Firefox exploit. We define gadget length as the number of instructions in the gadget. In the proof-of-concept exploits, while the smaller gadgets enabled popping values into registers from the stack, the longer gadgets enabled operations like mathematical computations and memory loads by dereferencing registers.

The results show that there is an abundance of gadgets available in every gadget length bucket in all evaluated applications. Although gadgets with 0 leaf calls are dominating the numbers, there are still many gadgets with 1 leaf call but significantly less than gadgets with 0 leaf calls. Among the gadgets reachable with the byte level memory write primitive, we also found gadgets with 2 and 3 leaf calls, but there were few of them, namely in total 11 gadgets had 2 leaf calls in the gadget length buckets from 13 to 15 of Nginx and 1 gadget had 3 leaf calls in the gadget length bucket 15 of Nginx. Among the gadgets reachable with the 4-bit memory write primitive, we found no gadget that has 2 or more leaf calls. In Firefox we had not found any gadget with 1 or more leaf calls, which we believe is due to Firefox being a much more complex application and for being a C++ application with many indirect virtual calls. Finally, we note that the number of gadgets reachable with the byte level memory write primitive is an order of magnitude more than the ones reachable with the 4-bit memory write primitive.

Gadget semantics. To successfully perform a code-reuse attack, one will need gadgets with the desired functionalities. Table 2 classifies the gadgets according to their semantics, based on their instructions. This group of semantics are used by Ropper. We further extended this group by adding the instructions `not reg` and `syscall` to the corresponding categories. During the analysis we assign categories to the found gadgets. In Fig 2a and 2b, we respectively show the number of gadgets, that are reachable

Category	Instructions (Intel syntax)
Stack Pivot (SP)	`sub rsp, num`
	`add rsp, num`
	`mov rsp, mem`
	`mov rsp, reg`
	`xchg reg/mem, rsp`
	`add rsp, num`
	`ret num`
Load Mem (LM)	`mov reg, mem`
Write Mem (WM)	`mov mem, reg`
Load Reg (LR)	`pop reg`
Clear Reg (CR)	`xor reg, reg`
Inc Reg (IR)	`add reg, 1`
	`inc reg`
Sub Reg (SR)	`sub reg, reg`
Add Reg (AR)	`add reg, reg`
Xchg Reg (XR)	`xchg reg, reg`
Invert Reg (NR)	`neg reg`
	`not reg`
Syscall (S)	`int 0x80`
	`syscall`

TABLE 2: Instructions used to assign categories to gadgets.

by patching a byte or 4-bits in the code pointers, in each category.

To perform critical operations on the system, gadgets with syscall instructions are crucial to have at reach. Furthermore the availability of a syscall gadget can dictate whether a successful exploit might be possible or not. Among the gadgets reachable by patching a byte, the number of gadgets with syscalls ranges from 748 in Asterisk to 46 in Firefox. Among the gadgets reachable by patching 4 bits, the number of gadgets with syscalls for the applications Asterisk, Nginx and Lighttpd is respectively 24, 12, and 16. The code pointers in the collected stack dumps for Apache and Firefox did not provide a gadget when patching 4 bits in the code pointers. In such a case, an attacker can opt for expanding the patching capabilities with the already reachable gadgets by emulating a write primitive that can patch more bits of the code pointers.

Although memory operating instructions are much more prevalent than data computing instructions, the overall results indicate that the expressiveness is good enough with the amount of gadgets available in the categories.

Effectiveness against Defenses. Table 3 shows the effectiveness of PIROP against ASLR techniques through entropy numbers calculated for our different PIROP exploits on Asterisk (AST1, AST2) and Firefox. The entropy of each exploit provides a measure of its exploitation probability across different randomization techniques and is computed by summing up the entropy of its individual gadgets. The entropy of a gadget is computed by taking the log with base 2 of the gadget's number of possible configurations. For each randomization technique, the entropy numbers are computed without taking into account the effect of any other randomization technique to assess the effect of the individual techniques on the exploits.

In the *single random base*, *module-level* and *page-level* randomization techniques, the code is page-aligned which means that, regardless of where the code is mapped, the
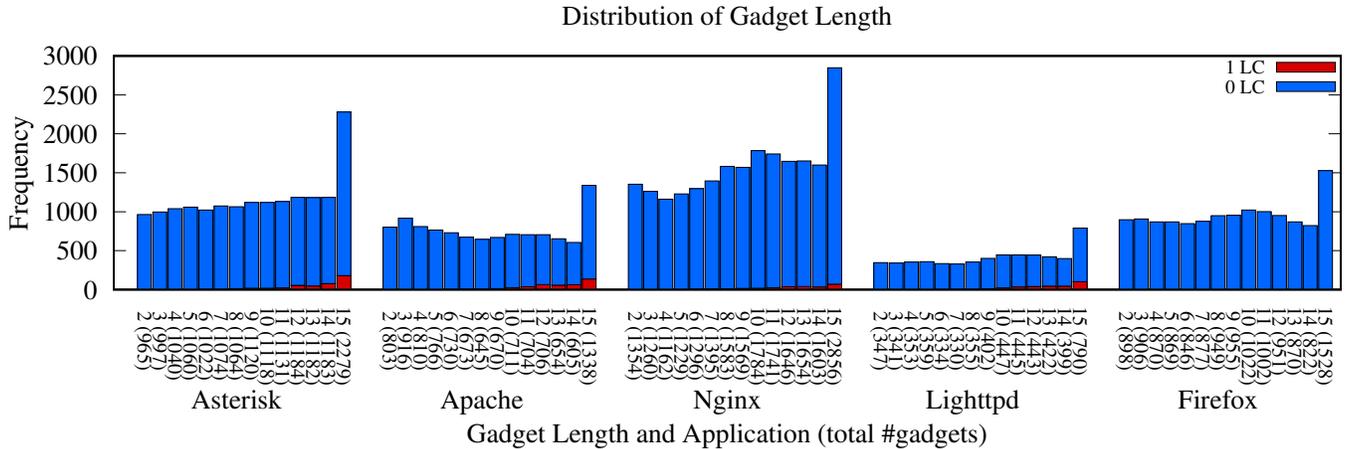
Distribution of Gadget Length

(a) Gadgets with 8-bits patched code pointers.

Distribution of Gadget Length
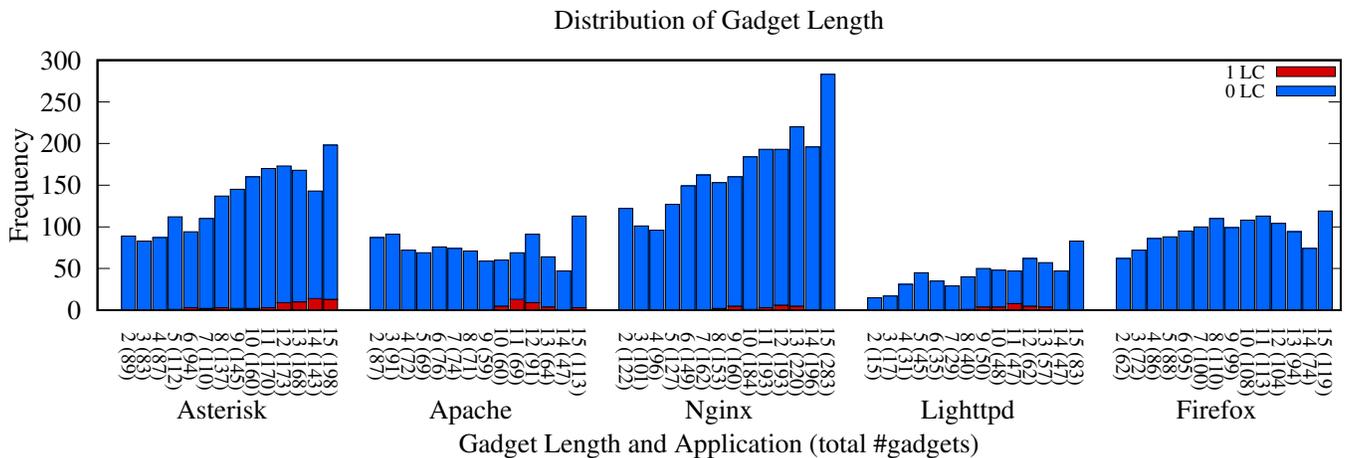
(b) Gadgets with 4-bits patched code pointers.

Figure 1: Number of gadgets available in applications for different gadget lengths, split by the number of leaf calls (LCs). The number in parentheses indicates the total number of gadgets for each bar.

least significant 12 bits in all code addresses are fixed in every randomization iteration. Because the PIROP exploits operate on the portion of these 12 non-randomized bits, the entropy for these exploits is 0, making these randomization techniques completely ineffective against PIROP.
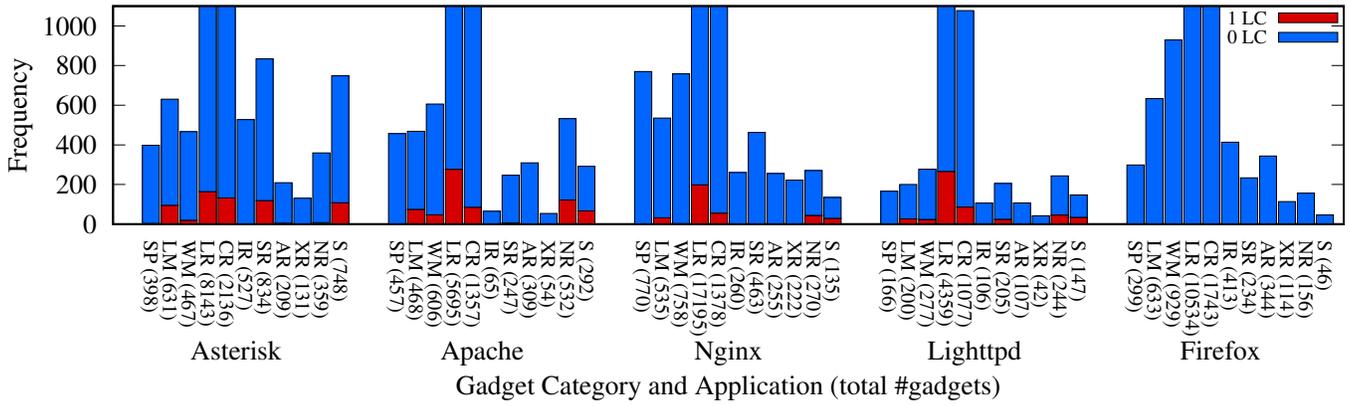
For the *function-level* randomization technique, since compilers tend to align functions to 16-byte aligned boundaries[2]. , the least significant 4 bits of each code address remain non-randomized. With a byte write primitive, each patched gadget of our Asterisk exploits, which is always in the same function as the original gadget, has an entropy of 4 bits. Since we patch 6 different code pointers to get the gadgets in the process creation Asterisk exploit and 3 different code pointers in the code injection exploit, the accumulated entropies are respectively 24 bits and 12 bits.

In the Firefox exploit, the 4-bit write primitive operates on the non-randomized least significant 4 bits of code pointers. For this reason, the entropy for each gadget (and thus for the resulting exploit) is 0. This shows that function-level randomization can be significantly weakened or made completely ineffective by PIROP.

Similar to function-level randomization, for *basic block-level* randomization, the entropy of each gadget is also given by the number of patch values that are valid in at least one of all the possible randomization iterations. Conservatively assuming that basic blocks have no alignment when randomized, all possible patch values are valid when patching code pointers to get to the gadgets. Since we patch the last byte in the Asterisk exploits and the basic blocks have no alignment, the entropy for a single basic block is 8 bits. In the process creation Asterisk exploit we retrieve gadgets from 6 different basic blocks and in the code injection exploit from 3 different basic blocks, which translate to the

2. Since instruction fetch units commonly operate on 16-byte aligned blocks [58], compilers tend to align branch targets like function and loop entry points to 16 bytes for optimization purposes.

Available Gadgets in the Functional Categories

(a) Gadgets with 8-bits patched code pointers.

Available Gadgets in the Functional Categories
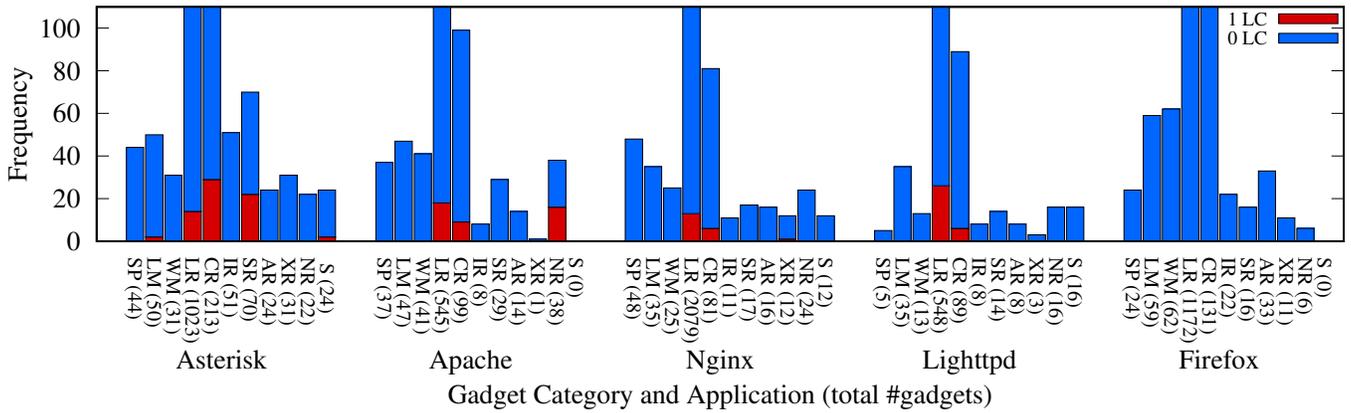
(b) Gadgets with 4-bits patched code pointers.

Figure 2: Number of gadgets, split by the number of leaf calls (LCs), that are assigned a certain category. The mapping of the abbreviated category is depicted in Table 2. The numbers in parentheses depict the total number of gadgets covered in the stacked bars.

entropy number 48 bits for the former exploit and 24 bits for the latter. However, in practice, the entropy is lower, because functions with a small number of basic blocks have a few basic block permutations and thus a lower entropy. This is reflected in the final numbers reported in Table 3. The Firefox exploit leverages gadgets from 6 different basic blocks and modifies up to 4 bits of a given return addresses, which gives an entropy of 24 bits with 4 bits of entropy for each basic block. However, on average we patch only 2.33 out of 4 bits decreasing the actual entropy to 14.00. The functions contain a sufficiently large number of basic blocks to not further reduce the entropy. Based on these results, it can be noted that basic block-level randomization makes successful PIROP exploitation much harder.

For *callee-saved register stack slot* randomization, the entropy of each gadget is determined by the number of permutations of the effectively used register restoring in-

structions. In the process creation Asterisk exploit, we use 3 function epilogues, yielding a total entropy number of about 10.08 bits. Although in the code injection exploit against Asterisk we also use 3 function epilogues, the total entropy is about 10.49 bits due to the usage of a different number of register restoring instructions in different epilogues. The Firefox exploit does not utilize whole function epilogues and instead only relies on smaller gadgets. Thus, the number of actually used register restoring instructions is lower. In total 2 gadgets contain such instructions resulting in an entropy of 7.75 bits. These numbers indicate that callee-saved register stack slot randomization has an impact, but provides insufficient entropy against PIROP to deter practical attacks.

For *callee-saved register allocation* randomization, the entropy per gadget is the number of permutations of the effectively used callee-saved registers in the gadget. Since overlapping gadgets share the degree of randomness, we

Code Randomization Techniques	Entropy PIROP Exploits			Bypassed (byte-level write)	Bypassed (bit-level write)	Comments
	AST1	AST2	FF			
Single random base [1]	0	0	0	●	●	
Module-level [6], [14]	0	0	0	●	●	
Page-level [3], [9]	0	0	0	●	●	
Function-level [2], [4]–[6], [12], [20], [23]	24	12	0	○	●	Functions 4-bit aligned
Basic block-level [3], [8], [9]	31.64	20.64	14.00	○	○	
Callee-saved register stack slots [7], [10], [13], [23]	10.08	10.49	7.75	○	○	
Callee-saved register allocation [7], [20], [23], [25], [45]	5.17	12.08	7.49	○	○	
Defenses against information disclosures						
MVX [28]–[30], [46]–[50]				●	●	
XoM [20]–[25]				●	●	
Destructive code reads [26], [27]				●	●	
Re-randomization [12], [31]–[36]				●	●	Requires live pointers

TABLE 3: Effectiveness of PIROP against state-of-the-art code randomization and information disclosure defense techniques, including the entropy in the exploits under different code randomization techniques. Note that our analysis refers to the individual defense techniques rather than full existing solutions (○='severely weakened', ●='completely bypassed')

consider the longest gadget among them. In the process creation Asterisk exploit, we effectively use 2 callee-saved registers in two different gadgets. Since there are 6 possible callee-saved registers, the entropy for this exploit is 5.17 bits (i.e. $log_2(6 * 6)$). In the code injection Asterisk exploit, we have 3 different gadgets in which we, respectively, effectively use 3, 1 and 1 callee-saved registers. This gives an entropy of 12.08 bits. In the Firefox exploit 2 gadgets use 2 and 1 callee-saved registers, respectively, which results in an entropy of 7.49 bits. Again, these numbers indicate that callee-saved register allocation randomization, while effective, provides insufficient entropy against PIROP to deter practical attacks.

Moreover, PIROP also bypasses all defenses against information disclosure listed in Table 3. MVX [28]–[30], [46]–[50] can detect deviant behavior when randomized pointers are directly leaked or leaked pointers are used for exploitation. With PIROP, the attack is fully position-independent, that is requires no disclosure and no attacker-provided pointers (only offsets), thus the absolute addresses are no longer relevant. Moreover, since PIROP never reads the code, XoM [20]–[25] and destructive code reads [26], [27] are equally powerless. Finally, PIROP is also generally resilient to traditional code re-randomization, given that even patched code pointers are re-randomized correctly to their patched re-randomized counterparts by the defense. This, however, assumes only live pointers on the stack are used in the attack (e.g., as done in our Firefox exploit).

6. Mitigations

Defending against PIROP attacks is possible, but retaining the performance and deployability advantages of traditional code randomization is challenging. In the following, we discuss several potential approaches to mitigate or prevent the attacks proposed earlier.

Stopping PIROP primitives. A practical way to stop PIROP attacks is to remove the necessary primitives. For example, to eliminate relative memory write primitives, we may rely on memory safety solutions [59]–[64] or, alternatively, data-flow integrity solutions [65], [66]. However, these solutions incur nontrivial performance overhead, reducing the performance benefits of ASLR. Also, in rare cases, it *may* be possible to exploit a system without patching.

An alternative is to thwart PIROP's stack massaging primitive, either by means of stack randomization techniques [4], [5], [12], [67], or shadow stacks [67]–[71]. Unfortunately, stack randomization is not cheap in terms of performance, while efficient implementations of shadow stacks are hard to implement securely [17], [18], since they themselves rely on randomization. In general, eliminating stack massaging alone stops PIROP, but not necessarily other variants of position-independent code reuse attacks.

Improving ASLR. Fine-grained ASLR [7], [7], [8], [10]–[13], [20], [22], [23], [23], [25], [25], [36], [45] also mitigates PIROP, but at a cost in performance. Specifically, as shown in Section 5, more efficient solutions such as function-level or register-level randomization may not offer sufficient entropy per patched gadget. Very fine-grained randomization (i.e., at or below the basic block level) significantly raises the bar for an adversary, but is expensive, e.g., because of the less efficient use of the instruction cache.

Code pointer hiding offers a good compromise [22], [23]. While it cannot remove position-independent gadgets in the trampoline tables, one could randomize such tables with high entropy and/or sprinkle them with booby traps [72]. Finally, in MVX [28]–[30], [46]–[50] we could limit the high-entropy randomization to the "slave" variant, which typically has a higher performance budget because of the faster (emulated) system calls [73].

Data space randomization. Data randomization alone (e.g., heap ASLR) is not sufficient to stop PIROP, as the attacker can patch not only code but also data pointers (or avoid them altogether). However, techniques that randomize the data space itself by means of encryption are another avenue for mitigation. Full data space randomization [74] can produce a non-linear randomization of all the bits in the ROP payload, making PIROP's patching step much harder but also introducing nontrivial overheads. More efficient solutions such as PointGuard [75], ASLR-Guard [71], Shuffler [32], LR^2 [20], and CCFI [76] encrypt only code/data pointers, but with different encryption schemes. The stronger

the encryption, the higher the per-patched-gadget entropy, but the lower the performance. As with ASLR, these solutions do not stop the rare cases where patchless exploits may be possible.

Other defenses. Other code-reuse defenses to consider for mitigation include CPI and CFI, although they are both more expensive than randomization. CPI [70] can isolate code pointers and guarantee their integrity, but also needs to protect its safe region [77]. As for CFI, [44], [70], [78]–[90], the more fine-grained the CFI strategy, the better the security. However, even coarse-grained CFI may suffice to reduce the gadget space such that practical attacks become hard.

7. Related Work

We extensively discussed mitigation strategies against PIROP and traditional code reuse in the previous sections. Hence, we only focus on related work describing attacks related to position-independent code reuse.

Advanced code-reuse attacks. After the original ROP attack [91], many advanced forms of code-reuse attacks have been proposed by the community to counter a variety of defenses. For example, jump-oriented programming (JOP) attacks [41], [92] solely rely on forward indirect branches to bypass defenses that only protect the return branches. More recent code-reuse attacks assume even tighter constraints to bypass control-flow integrity (CFI) or related techniques. A first generation of attacks proposes code-reuse techniques to bypass anomaly-based CFI [93], [94] and coarse-grained CFI absent a shadow stack [94]–[96]. A second generation shows that even fine-grained CFI variants are susceptible to practical attacks [43], [97], [98]. Unlike PIROP, all these code-reuse attack techniques focus on defenses that deterministically restrict the (known) gadget set at the attacker's disposal.

Other code-reuse variants target randomization-based defenses. JIT-ROP attacks [16], [34] demonstrate that an attacker can rely on arbitrary read primitives to disclose arbitrary code pointers in code/data sections and craft a just-in-time ROP payload against fine-grained ASLR. SROP [99] highlights that an attacker can also rely on particular harder-to-randomize code gadgets and reduce the information disclosure surface. More recently, inference attacks [100] show that an attacker can disclose the location of relevant gadgets by leaking the location of related gadgets via disclosure code reads. Finally, counterfeit object-oriented programming (COOP) [43], address-oblivious code reuse [101], and Newton [102] show that JIT code-reuse attacks are possible by solely relying on code pointers disclosed by data sections. Unlike all these attacks, PIROP demonstrates that code-reuse attacks are even possible in complete absence of information disclosure from code or data sections, providing a concrete upper bound to the security of ASLR and information disclosure defenses.

Advanced exploitation primitives. The primitives used in PIROP draw inspiration from techniques adopted in previous exploitation strategies. For example, our stack massag-

ing primitive is inspired by heap massaging techniques [103] to facilitate the exploitation of temporal (e.g., use-after-free) vulnerabilities. Similar techniques have also been recently applied to physical memory to facilitate exploitation of hardware (e.g., Rowhammer) vulnerabilities [104], [105]. Unlike all these techniques, PIROP relies on memory massaging primitives to lure a victim into generating a code-reuse payload on her behalf as a prelude to exploitation.

Similarly, the adoption of relative memory write primitives in PIROP is inspired by partial pointer overwrites used to facilitate 32-bit RILC exploitation [37] or reduce the information disclosure effort in modern exploits [38]. Relative memory write primitives have also been used before to bypass stack cookies, e.g., using a non-linear stack-based buffer overflow [54]. Unlike all these techniques, PIROP relies on relative memory write primitives to operate multiple and targeted modifications to a full ROP payload for disclosure-resilient position-independent code reuse exploitation on modern 64-bit architectures.

Attacks against ASLR. Many existing attacks have demonstrated weaknesses in ASLR techniques. Traditional attacks against randomization rely on memory disclosure to leak code pointers from code/data sections and bypass the protection [15], [16], [34], [106]. Other attacks demonstrate that, even in absence of memory disclosure, an attacker can bypass coarse-grained randomization (traditional ASLR) via a variety of side channels, such as control flow [107], [108], memory deduplication [109], cache [110], [111], TLB [111], crashes [19], [112], exceptions [17], [113], [114], and memory allocations [18]. Unlike all these techniques, PIROP demonstrates that information disclosure is not a fundamental precondition for modern code-reuse attacks and practical exploits are still possible even with fine-grained randomization and no disclosure capabilities.

8. Conclusion

ASLR and CFI are the most prominent solutions to efficiently mitigate code-reuse exploits. Recent attacks against ideal CFI implementations have established an upper bound for their security guarantees.

In this paper, we investigated a similar upper bound for ASLR techniques. We assumed an ideal scenario, that is an attacker with no information disclosure capabilities to leak the code layout. We demonstrated practical attacks via a method we call Position-Independent Return-Oriented Programming (PIROP). Rather than leaking code pointers and using a just-in-time code-reuse strategy, attackers can lure the victim into generating a position-independent code-reuse payload, later tweaked and used for exploitation. Our research shows that PIROP bypasses common ASLR implementations completely, and significantly weakens several more advanced defenses. Finally, our analysis suggests that, rather than offering a competing solution, code randomization may be better combined with CFI to guarantee sufficient protection.

Acknowledgments

References

[1] PaX Team, "Address space layout randomization (ASLR)," 2003, pax.grsecurity.net/docs/aslr.txt.

[2] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," in *PETS*, 2016.

[3] S. Crane, A. Homescu, and P. Larsen, "Code randomization: Haven't we solved this problem yet?" in *SecDev*, 2016.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits," in *USENIX SEC*, 2005.

[5] ——, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *USENIX SEC*, 2003.

[6] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *ACSAC*, 2006.

[7] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *S&P*, 2012.

[8] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *CCS*, 2012.

[9] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX SEC*, 2014.

[10] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *S&P*, 2012.

[11] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *CGO*, 2013.

[12] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX SEC*, 2012.

[13] H. Koo and M. Polychronakis, "Juggling the gadgets: Binary-level code randomization using instruction displacement," in *ASIACCS*, 2016.

[14] M. Sun, J. C. Lui, and Y. Zhou, "Blender: Self-randomizing address space layout for android apps," in *RAID*, 2016.

[15] P. Vreugdenhil, "Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit," 2010. [Online]. Available: https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-webkit-exploit/

[16] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[17] E. Goktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *USENIX SEC*, 2016.

[18] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *USENIX SEC*, 2016.

[19] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *S&P*, 2014.

[20] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A. R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *NDSS*, 2016.

[21] 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 *CCS*, 2014.

[22] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table randomization and protection against function-reuse attacks," in *CCS*, 2015.

[23] 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 *S&P*, 2015.

[24] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *CODASPY*, 2015.

[25] J. Gionta, W. Enck, and P. Larsen, "Preventing kernel code-reuse attacks through disclosure resistant code diversification," in *CNS*, 2016.

[26] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *CCS*, 2015.

[27] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *ASIACCS*, 2016.

[28] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *DSN*, 2016.

[29] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, "Secure and efficient application monitoring and replication," in *USENIX ATC*, 2016.

[30] R. Gawlik, P. Koppe, B. Kollenda, A. Pawlowski, B. Garmany, and T. Holz, "Detile: Fine-grained information leak detection in script engines," in *DIMVA*, 2016.

[31] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *CODASPY*, 2016.

[32] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *OSDI*, 2016.

[33] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make ASLR win the clone wars: Runtime re-randomization," in *NDSS*, 2016.

[34] L. Davi, C. Liebchen, A. R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *NDSS*, 2015.

[35] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *CCS*, 2015.

[36] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the code space to counter disclosure attacks," in *EuroS&P*, 2017.

[37] Anonymous, "Bypassing PaX ASLR protection," *Phrack magazine*, vol. 11, no. 59, 2002.

[38] C. Evans, "A scriptless 0day exploit against Linux desktops," http://scarybeastsecurity.blogspot.nl/2016/11/0day-exploit-advancing-exploitation.html.

[39] Solar Designer, "Return-to-libc attack," BugTraq, August 1997.

[40] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Info. & System Security*, vol. 15, no. 1, 2012.

[41] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ASIACCS*, 2011.

[42] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *ACM STC*, 2009.

[43] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *S&P*, 2015.

[44] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[45] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating return-oriented programming through gadget-less binaries," in *ACSAC*, 2010.

[46] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *EuroSys*, 2009.

[47] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *USENIX SEC*, 2006.

[48] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "GHUMVEE: Efficient, effective, and flexible replication," in *FPS*, 2012.

[49] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicae for defeating memory error exploits," in *IPCCC*, 2007.

[50] S. Volckaert, B. Coppens, and B. De Sutter, "Cloning your gadgets: Complete ROP attack immunity with multi-variant execution," *TDSC*, vol. 13, no. 4, 2016.

[51] H. Fuhrmannek, "Bug 1248876 - (CVE-2016-1977) Graphite2 Machine::Code::decoder::analysis::set_ref stack out of bounds bit set," https://bugzilla.mozilla.org/show_bug.cgi?id=1248876.

[52] E. Intelligence, "DoS? Then Who Was Phone?" http://blog.exodusintel.com/tag/cve-2012-5976/.

[53] M. Miller, "Heap corruption issues reported to Microsoft," 2017. [Online]. Available: https://twitter.com/epakskape/status/851479629873332224

[54] S. Gross, "Strengths and weaknesses of LLVM's SafeStack buffer overflow protection," http://blog.includesecurity.com/2015/11/LLVM-SafeStack-buffer-overflowprotection.html.

[55] S. International, "Graphite Home," http://graphite.sil.org/.

[56] ——, "Graphite Font development," http://scripts.sil.org/cms/scripts/page.php?site_id=projects&item_id=graphite_devFont#fontdev.

[57] S. Schirra, "Ropper," https://github.com/sashs/Ropper.

[58] Intel, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, June 2016.

[59] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *ISMM*, 2010.

[60] ——, "SoftBound: Highly compatible and complete spatial memory safety for C," in *PLDI*, 2009.

[61] E. van der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *EuroSys*, 2017.

[62] K. Koning, T. Kroes, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *EuroSys*, 2018.

[63] A. Milburn, H. Bos, and C. Giuffrida, "Safeinit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *NDSS*, 2017.

[64] I. Haller, J. Yuseok, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *CCS*, 2016.

[65] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *OSDI*, 2006.

[66] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *S&P*, 2008.

[67] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *NDSS*, 2015.

[68] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ASIACCS*, 2015.

[69] "Clang's SafeStack," http://clang.llvm.org/docs/SafeStack.html.

[70] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI*, 2014.

[71] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *CCS*, 2015.

[72] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in *NSPW*, 2013.

[73] P. Hosek and C. Cadar, "Varan the unbelievable: An efficient n-version execution framework," in *ASPLOS*, 2015.

[74] S. Bhatkar and R. Sekar, "Data space randomization," in *DIMVA*, 2008.

[75] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard TM: protecting pointers from buffer overflow vulnerabilities," in *USENIX SEC*, 2003.

[76] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *CCS*, 2015.

[77] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *EuroSys*, 2017.

[78] B. Niu and G. Tan, "Per-input control-flow integrity," in *CCS*, 2015.

[79] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *OSDI*, 2006.

[80] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *USENIX SEC*, 2013.

[81] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX SEC*, 2014.

[82] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *ACSAC*, 2011.

[83] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *S&P*, 2010.

[84] 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 *S&P*, 2013.

[85] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *DIMVA*, 2015.

[86] V. van der Veen, E. Göktaş, M. Contag, A. Pawloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *S&P*, 2016.

[87] V. van der Veen, D. Andriesse, E. Goktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *CCS*, 2015.

[88] M. Zhang and R. Sekar, "Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks," in *ACSAC*, 2015.

[89] B. Niu and G. Tan, "Modular control-flow integrity," in *PLDI*, 2014.

[90] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *NDSS*, 2015.

[91] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[92] S. Checkoway, L. Davi, A. Dmitrienko, A. R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *CCS*, 2010.

[93] E. Goktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *USENIX SEC*, 2014.

[94] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX SEC*, 2014.

[95] L. Davi, A. R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX SEC*, 2014.

[96] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *S&P*, 2014.

[97] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX SEC*, 2015.

[98] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. C. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *CCS*, 2015.

[99] E. Bosman and H. Bos, "Framing signals—a return to portable shellcode," in *S&P*, 2014.

[100] K. Snow, R. Rogowski, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *S&P*, 2016.

[101] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity," in *NDSS*, 2017.

[102] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *CCS*, 2017.

[103] A. Sotirov, "Heap feng shui in JavaScript," *Black Hat Europe*, 2007.

[104] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a needle in the software stack," in *USENIX SEC*, 2016.

[105] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer attacks on mobile platforms," in *CCS*, 2016.

[106] F. Serna, "The info leak era on software exploitation," in *BlackHat USA*, 2012.

[107] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *S&P*, 2015.

[108] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *CCS*, 2014.

[109] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory deduplication as an advanced exploitation vector," in *S&P*, 2016.

[110] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, 2017.

[111] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *S&P*, 2013.

[112] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS*, 2004.

[113] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding," in *NDSS*, 2016.

[114] B. Kollenda, E. Goktas, T. Blazytko, P. Koppe, R. Gawlik, R. K. Konoth, C. Giuffrida, H. Bos, and T. Holz, "Towards automated discovery of crash-resistant primitives in binaries," in *DSN*, 2017.