What Cannot be Read, Cannot be Leveraged? Revisiting Assumptions of JIT-ROP Defenses

Giorgi Maisuradze
CISPA, Saarland University
Saarland Informatics Campus
gmaisura@mmci.uni-saarland.de

Michael Backes
CISPA, Saarland University
Saarland Informatics Campus
backes@mpi-sws.org

Christian Rossow
CISPA, Saarland University
Saarland Informatics Campus
crossow@mmci.uni-saarland.de

Abstract

Despite numerous attempts to mitigate code-reuse attacks, Return-Oriented Programming (ROP) is still at the core of exploiting memory corruption vulnerabilities. Most notably, in JIT-ROP, an attacker dynamically searches for suitable gadgets in executable code pages, even if they have been randomized. JIT-ROP seemingly requires that (i) code is *readable* (to find gadgets at run time) and (ii) executable (to mount the overall attack). As a response, Execute-no-Read (XnR) schemes have been proposed to revoke the read privilege of code, such that an adversary can no longer inspect the code after fine-grained code randomizations have been applied.

We revisit these "inherent" requirements for mounting JIT-ROP attacks. We show that JIT-ROP attacks can be mounted without ever reading any code fragments, but instead by *injecting* predictable gadgets via a JIT compiler by carefully triggering useful displacement values in control flow instructions. We show that defenses deployed in all major browsers (Chrome, MS IE, Firefox) do not protect against such gadgets, nor do the current XnR implementations protect against code injection attacks. To extend XnR's guarantees against JIT-compiled gadgets, we propose a defense that replaces potentially dangerous direct control flow instructions with indirect ones at an overall performance overhead of less than 2% and a code-size overhead of 26% on average.

1 Introduction

Code-reuse attacks, such as Return-Oriented Programming (ROP), enable an attacker to bypass Execute-XOR-Write (X^W) policies by suitably chaining existing small code fragments (so-called *gadgets*). One of the most prominently explored concepts to defend against such attacks involves randomizing programs so that an attacker can no longer reliably identify and chain such gadgets, whether by code transformations [19, 6, 14], data region hardening [5, 23], or whole address space randomization [4]. However, a novel class of attacks, dubbed JIT-ROP, allows for code reuse even for such diversified programs [31]. JIT-ROP leverages a memory disclosure vulnerability in combination with a scripting environment—which is part of all modern browsers—to read existing code parts, notably *after* they were randomized. Once the code has been read (e.g., using a memory disclosure vulnerability), an attacker can dynamically discover and chain gadgets for conventional code-reuse attacks.

Mounting a successful JIT-ROP attack seemingly requires the ability to (i) read code fragments and identify suitable gadgets (otherwise the adversary would not know what to combine) and to (ii) execute them (so that the overall attack can be mounted). The recently proposed Execute-no-Read (XnR) schemes [2, 11, 12] consequently strive to eliminate JIT-ROP attacks by ensuring that executable code is *non-readable*, i.e., marking code sections as executable-only while explicitly removing the read privilege. Hence an adversary can no longer inspect the code after fine-grained code randomization techniques have been applied and should thus fail to identify suitable gadgets. As a more pointed statement: what cannot be read, cannot be leveraged.

Our contributions: In this paper, we carefully revisit these seemingly inherent requirements for mounting a successful JIT-ROP attack. As our overall result, we show that JIT-ROP attacks can often be mounted without ever reading any code fragments, but by instead injecting *arbitrary, predictable* gadgets via a JIT compiler and by subsequently assembling them to suitable ROP chains without reading any code pages.

As a starting point, we show how to obtain expressive unaligned gadgets by encoding specially-crafted constants in instructions. Prior research has already shown that *explicit* constants in JavaScript statements, e.g., in assignment statements like `x = 0x12345678`, can be used to generate unaligned gadgets [1]. Browsers started to fix such vulnerabilities, e.g., by blinding explicit con-

stants (i.e., XOR-ing them with a secret key), and/or by applying fine-grained code randomization techniques (cf. Athanasakis *et al.* [1]). We show that *implicit* constants in JIT-compiled code can be exploited in a similar manner, and are hence far more dangerous in the JIT-ROP setting than commonly believed. To this end, we generate JavaScript code that emits specific offsets in relative jumps/calls in the JIT-compiled code. We show that both relative jumps and relative calls can be used to encode attacker-controllable values in an instruction's displacement field. These values can later be used as unaligned gadgets, i.e., an attacker that controls the jump or call destination (or source) can predict the displacement and thereby generate deterministic gadgets on-the-fly, without the need to ever read them before use. We demonstrate the impact of our attack by injecting almost arbitrary two- or three-byte-wide gadgets, which enable an attacker to perform arbitrary system calls, or, more generally, obtain a Turing-complete instruction set. We show that all major browsers (Chrome, Internet Explorer, Firefox) are susceptible to this attack, even if code randomization schemes such as NOP insertion (like in Internet Explorer) are in place.

The ability to create controllable JIT-compiled code enables an adversary to conveniently assemble ROP chains without the requirement to ever read code. This challenges current XnR instantiations in that code does not have to be readable to be useful for ROP chains, highlighting the need to complement XnR with effective code pointer hiding and/or code randomization schemes also in JIT-compiled code. Unless XnR implementations additionally protect JIT-compiled code, they do not prevent attackers from reusing predictable attacker-generated gadgets, and hence from mounting JIT-ROP attacks. We stress that a complete XnR implementation that offered holistic code coverage (i.e., hiding code-pointers also in JIT-compiled code) may also be effective against our attack. However, maintaining XnR's guarantees also for JIT-compiled and attacker-controlled code imposes additional challenges in practical settings: First, fine-grained code randomization schemes that are implemented in XnR do not add security against implicit constants, and they hence make gadget emissions, proposed in this paper, possible. In particular, the widely deployed concepts of register renaming and instruction reordering do not affect our proposed unaligned gadgets. Moreover, fine-grained code randomization techniques commonly deployed in browsers (as NOP insertion in IE) are not sufficient either, as the attacker can test the validity of its gadgets before using them. Second, the lack of code pointer hiding in JIT-compiled code in current XnR instantiations constitutes an additional vector of attack, since adversaries can then still leverage our attack to encode constants in relative calls.

We finally explored how to extend XnR's guarantees against implicit constants in JIT-compiled code. One option would be to extend the use of call trampolines in XnR schemes also to JIT compilers, as suggested by Crane *el al.* [11, 12]. However, this will replace existing direct calls with direct jumps to trampolines, which also encode implicit constants. Furthermore, trampolines will introduce new relative offsets in their direct call instructions. As the locations of trampolines are not hidden (e.g., they can be revealed by reading the return address on stack), in the presence of an unprotected code pointer, the attacker will be able to predict encoded constants by leaking either the caller or the callee address. As an orthogonal alternative, in this paper we propose to (i) replace relative addressing with indirect calls and (ii) blind (i.e., reliably obfuscate) all explicit constants used to prepare the indirect calls. We implement our defense in V8, the JavaScript engine of Chrome, and show that our proposal imposes less than 2% performance and 26% code size overhead, while effectively preventing the attacks described in this paper.

The summarized contributions of our paper are:

- We present a novel class of attacks that encode ROP gadgets in implicit constants of JIT-compiled code. We thereby show that reading code fragments is *not* necessarily a prerequisite for assembling useful gadgets in order to mount a JIT-ROP attack.
- We demonstrate that all three major browsers (Chrome, Internet Explorer, Firefox) are susceptible to our proposed attack.
- We discuss potential shortcomings when using XnR to protect JIT-compiled code. We show that the underlying assumptions that XnR schemes build upon (such as code randomization) have to be carefully evaluated in the presence of JIT-compiled code.
- We implement a defense in V8 that replaces relative calls/jumps with indirect control flow instructions. This effectively prevents the attack proposed in this paper by removing dangerous implicit constants, exhibiting a performance overhead of 2% and a code size overhead of 26%.

The remainder of this paper is structured as follows. Section 2 provides background information on code-reuse attacks. Section 3 describes our threat model. Section 4 introduces the fundamentals of our attack and demonstrates its efficacy against three major browsers. Section 5 introduces an efficient defense against our attack. Section 6 discusses the implications of our work. Section 7 describes related work and Section 8 concludes the paper with a summary of our findings.

2 Background

We will use this section to provide background information on code-reuse attacks. We start by explaining ROP, and then provide insights on JIT-ROP, which collects code on-the-fly and thus evades existing randomization schemes like ASLR. Finally, we describe Execute-no-Read (XnR), a new defensive scheme that aims to protect against code-reuse attacks (including JIT-ROP).

2.1 Return Oriented Programming (ROP)

ROP has emerged since the wide deployment of Data Execution Prevention (DEP), which is a defense technique against regular stack overflow vulnerabilities. DEP, making the writable regions of the memory non-executable, forbids the attacker to execute the shellcode directly on the stack. As a response, attackers switched to code-reuse attacks, in which they execute *existing* code instead of injecting new code. ROP, proposed by Shacham [29], is a generalized version of the `ret-to-libc` attack [22], which redirects the control flow of the program to existing code, such as the program's code or imported functions (e.g., in `libc`). In ROP, an attacker uses short instruction sequences (called gadgets) ending with a control flow instruction (e.g., `ret`). Return instructions are used to chain multiple gadgets together by providing their addresses as the return values on the stack. Checkoway *et al.* [8] showed that it is possible to launch ROP attacks without using return instructions, i.e., via leveraging other control flow changing instructions such as indirect jumps or calls.

Code-reuse remains a popular attack technique and has triggered a variety of defensive schemes. Most prominent, and deployed in most operating systems, is Address Space Layout Randomization (ASLR). ASLR randomizes the base addresses of memory segments and prevents an attacker from predicting the addresses of gadgets. Although ASLR is effective for pre-computed gadget chains, ASLR has known shortcomings in that it only randomizes base addresses and is too coarse-grained. An attacker can thus reveal the memory layout of an entire ASLR-protected segment with a single leaked pointer. To address this problem, fine-grained ASLR randomization schemes have been proposed that add randomness inside the segment [16, 20, 25] (we refer the reader to Larsen's survey [21]).

2.2 JIT-ROP

To counter ASLR, Snow *et al.* proposed a new attack technique, called *just-in-time* code reuse (JIT-ROP) [31]. By leveraging the fact that an adversary is able to read randomized code sections, JIT-ROP undermines fine-grained ASLR schemes. JIT-ROP is based on the following assumptions: (i) a memory disclosure vulnerability, allowing the attacker to read data at arbitrary locations, (ii) at least one control flow vulnerability, (iii) a scripting environment running code provided by the attacker. The basic idea of the JIT-ROP is the following:

(J1) Repeatedly using the memory disclosure vulnerability, the attacker follows the code pointers in the memory to read as many code pages as possible.
(J2) From the read code pages, JIT-ROP extracts gadgets (e.g, Load, Store, Jump, Move) and collects useful API function calls (e.g., `LoadLibrary`, `GetProcAddress`).
(J3) Given the gadgets and API functions, the JIT-ROP framework takes an exploit, written in a high-level language, as an input and compiles it to a chain of gadgets and function calls that perform a code-reuse attack.
(J4) Finally, the control flow vulnerability is used to jump to the beginning of the compiled gadget chain.

JIT-ROP demonstrates that an adversary may be able to run code-reuse attacks even in the case of fine-grained ASLR or code randomization, as she can read the code and function pointers *after* they have been randomized.

2.3 Execute-no-Read (XnR)

In an attempt to close the security weakness that JIT-ROP has demonstrated, researchers suggest marking code sections as *non-readable*. Such Execute-no-Read (XnR) schemes were proposed by Backes *et al.* [2], and were strengthened by Crane *et al.* [11, 12] shortly thereafter. The common goal is to prevent step (J1) of the JIT-ROP attack, as the attacker can no longer dynamically search for gadgets in non-readable code sections.

XnR: Lacking support for XnR pages in the current hardware, Backes *et al.* implemented XnR in software by marking code pages as `non-present` and checking the permissions inside a custom pagefault handler [2]. To increase the efficiency of this scheme, the authors propose to leave a window of N pages present. This exposes a few readable pages to the attacker, but prevents her from reading *arbitrary* code pages. As the authors suggest, at low window size ($N = 3$), the likelihood that an attacker can leverage code-reuse attacks using only the present code pages is negligible.

Readactor(++): Crane *et al.* suggested Readactor [11] and Readactor++ [12], both of which leverage hardware support to realize XnR. The authors suggest using Extended Page Tables (EPT), which were introduced recently to the hardware to assist virtualized environments. While regular page tables translate virtual addresses into physical ones, EPTs add another layer of indirection

and translate physical addresses of a VM to physical addresses of the host. EPTs allow marking pages as (non-)readable, (non-)writable, or (non-)executable, allowing enforcement of XnR in hardware. In addition, Readactor(++) hides code pointers by creating trampolines, and replacing all code pointers in readable memory with trampoline pointers. The underlying assumption of Readactor(++) is that fine-grained code diversification techniques are in place, such as function permutation, register allocation randomization, and callee-saved register save slot reordering.

Despite the fact that Readactor hid code pointers, the layout of some function tables (e.g., import tables or vtables) stayed the same. This allows an adversary to guess and reuse the function pointers from these tables [27]. Readactor++ fixed this issue by randomizing these tables (to get rid of the predictable layout) and randomly injecting pointers to illegal code (to forbid function fingerprinting by executing it).

Alternative XnR Designs: Gionta *et al.* [15] proposed HideM, which, using a split TLB technology, differentiates between memory accesses and only allows instruction fetches to access code pages. Further, HideM considers the data in the executable memory pages that need to be read and uses read policies to guarantee their security. However, although HideM might be used to enforce non-readable code, it highly depends on the hardware support (e.g., split TLB). Furthermore, HideM does not hide code pointers. In addition, Pereira *et al.* [26] designed a technique similar to Readactor(++) that aims to get non-readable code for mobile devices in ARM. One of the advantages of their approach, called Leakage-Resilient Layout Randomization (LR^2), is that it is implemented in software and does not require the underlying hardware support. LR^2 achieves this by splitting the memory space in half (into code and data pages) and instrumenting load instructions to forbid the attacker to read code. LR^2 also optimizes the use of trampolines by creating only a single trampoline for each callee instead of encoding one for each callee-caller pair.

Summarizing XnR: Even though the current XnR implementations mark JIT-compiled code as non-readable, existing prototypes allow to leak JIT code pointers via JIT-compiled code. We will show that an adversary that controls the JavaScript code can still run code-reuse attacks on the code generated by the JIT compilers.

2.4 JIT-Compiled Gadgets

JIT compilation remains a major challenge for XnR implementations. During JIT compilation, the JIT engine (e.g., of a browser) compiles JavaScript code into assembly instructions to optimize performance. This is done by converting each JavaScript statement into a sequence of corresponding assembly instructions, making the code output of the JIT compiler predictable. The deterministic JIT compilation allows an attacker to influence the code output by controlling the JavaScript code. For example, Blazakis [7] and Athanasakis *et al.* [1] propose to craft special JavaScript statements that JIT-compile into gadgets. Consider a statement with an immediate value, such as the assignment `var a=0x90909090`. The JIT engine will compile this into a sequence of assembly instructions, one of them being a `mov eax,0x90909090` instruction that encodes the attacker-chosen immediate value. After the compilation, the attacker can jump in the middle of the instruction and use the bytes of the immediate value as an unaligned gadget, such as four consecutive `nop` instructions in our simple example.

JIT Compiler Defenses: Modern JavaScript compilers prevent such unaligned gadgets by *constant blinding*. Instead of directly emitting constants in native code, compilers `XOR` them with randomly generated keys, making the resulting constants unpredictable. After constant blinding, the aforementioned JavaScript statement will be compiled to the following assembly instructions, effectively removing the attacker-controlled constant:

```
mov eax, (0x90909090 ⊕ KEY)
xor eax, KEY
```

For performance reasons, modern browsers only blind large constants. For example, Chrome and IE blind constants containing three or more bytes, giving the attacker a chance to emit arbitrary two-byte gadgets. Athanasakis *et al.* [1] demonstrated that two-byte gadgets are sufficient to mount a successful ROP attack, provided that (i) code sections are readable, and (ii) available gadgets happen to be followed by a `ret` instruction.

While constant blinding protects against such gadget emissions, as we will show, it does not protect against our novel form of JIT-compiled *implicit constant* gadgets.

3 Assumptions

We now describe our assumptions that we follow throughout this paper, detailing a threat model and discussing defenses that we assume are in place on the target system. These assumptions are in accordance with the recently proposed defense mechanisms against JIT-ROP, such as XnR [2] and Readactor [11].

3.1 Defense Techniques

We assume that the following defense mechanisms of the operating systems and the target application are in place:

- **Non-Executable Data**: Data Execution Policy (DEP) is enabled on the target system, e.g., by us-

ing the NX-bit support of the hardware, marking writable memory pages non-executable.

- **Address Space Layout Randomization**: The target system deploys base address randomization techniques such as ASLR, i.e., the attacker cannot predict the location of a page without a memory disclosure vulnerability. In addition, we assume popular fine-grained ASLR schemes [20, 33, 16, 25, 17], as suggested by current XnR implementations [11, 12], are applied on the executable, libraries, and JIT-compiled code.
- **Non-Readable Code**: We assume that all code segments are non-readable, with this being either software- [2] or hardware-enforced [11, 12], notably also assuming that JIT-compiled code is non-readable.
- **Hidden Code Pointers**: We assume that all code pointers, except for JIT-compiled ones, are present but anonymized, e.g., via pointer indirections such as trampolines proposed by Readactor. Note that, as mentioned by Crane *et al.*, Readactor(++) could be extended to also hide code pointers in JIT-compiled code. However, there is no implementation that shows this, neither is the performance impact of such a scheme clear. In addition, having the compiler running in the same process as the attacker might give the adversary the ability to read code pointers *during* the compilation process. We thus believe that hiding all possible (direct or indirect) code pointers is a challenging task and the attacker might still be able to leak the required function addresses.
- **JIT Hardening**: We assume modern JIT defenses such as randomized JIT pages, constant blinding, and guard pages (i.e., putting an unmapped page between mapped ones). In our attack, for simplicity, we assume that sandboxing is either disabled or can be bypassed via additional vulnerabilities. In addition, assessing the security of Control Flow Integrity (CFI) defenses in JIT compilers is out of scope of this paper, as our core contribution is to show that an attacker can *inject* gadgets, and not to discuss the actual process of diverting control flow. Instead, we demonstrate the threat of attacker-controlled code emitted by the JIT compiler.

3.2 Threat Model

In the following, we enumerate our assumptions about the attacker. This model is consistent with the threat model of previous attacks such as JIT-ROP [31] and with the XnR-based defense schemes.

- **Memory Disclosure Vulnerability**: We assume that the target program has a memory disclosure vulnerability, which can be exploited repeatedly by the attacker to disclose the *readable* memory space (i.e., we can read data, but cannot read code).
- **Control-Flow Diversion**: We assume that the target program has a control-flow vulnerability, allowing the attacker to divert the control flow to an arbitrary location. Note that this by itself does not allow the attacker to exploit the program, given the lack of ROP gadgets due to fine-grained ASLR and XnR.
- **JavaScript Environment:** We assume that the vulnerable process has a scripting environment supporting JIT compilation, for which the attacker can generate arbitrary JavaScript code. This is common for victims that use a browser to visit an attacker-controlled web site. Similarly, it applies to other programs such as PDF readers.

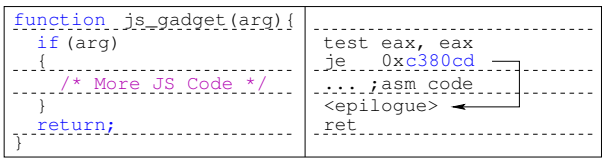
4 JIT-Compiled Displacement Gadgets

In this section, we discuss how an attacker can induce *new* JIT-compiled gadgets by crafting special JavaScript code. Intuitively, we show that an attacker can generate predictable JIT-compiled code such that she can reuse the code without searching for it. We introduce new techniques to trigger predictable gadgets that all modern JavaScript engines happen to generate. We demonstrate that an attacker can create and use almost arbitrary x86/x64 gadgets in modern browsers and their corresponding JavaScript engines, such as Google Chrome (V8), MS Internet Explorer (Chakra), and Mozilla Firefox (SpiderMonkey).

We introduce two techniques to emit gadgets via implicit constants. First, in Section 4.1, we leverage JavaScript's control flow instructions and emit conditional jumps (such as `je 0x123456`) that may encode dangerous offsets. Second, in Section 4.3, we show how an attacker can leverage offsets in direct calls, such as `call 0x123456`, to create gadgets.

4.1 Conditional Jump Gadgets

Our first target is to turn offsets encoded in conditional jumps (in JIT-compiled code) into gadgets. To this end, we use JavaScript statements, such as conditionals (`if/else`) or loops (`for/while`), that are compiled to conditional jumps. Figure 1(A) shows an example. In `js_gadget`, the body of the `if` statement contains a variable-length JavaScript code. After compilation, the `if` statement is converted to a sequence of assembly instructions containing a conditional jump, which, depending on the branch condition, either jumps over the body or falls through (e.g., `je <if_body_size>`). By varying



(A) JavaScript function (B) Disassembly

Figure 1: JavaScript function js_gadget and its corresponding disassembly

the code size inside the `if` body, we change the jump distance and thus the value encoded in the displacement field of the jump instruction in the compiled code. For example, if we aim for a `int 0x80;ret` (`0xcd80c3`) gadget, we have to fill the body of the `if` statement with JavaScript code that is compiled to `0xc380cd` bytes. The size of the JIT-compiled code for each JavaScript statement is fixed by the corresponding JIT compiler, and thus an attacker can precisely generate code of any arbitrary length. Note that the bytes of the size and the emitted gadget are mirrored because of the little-endian format used in x86/x64 architectures. The compiled version of `js_gadget` is shown in Figure 1(B).

Emitting such three-byte gadgets requires large portions of JavaScript code. In case the malicious JavaScript code has to be loaded via the Internet, this might drastically increase the time required for all gadgets to be in place. An attacker could overcome this limitation by utilizing the `eval` function. Instead of having ready-made JavaScript code, we thus use a function that constructs and emits all required gadgets on-the-fly. Such a script to dynamically generate arbitrary gadgets occupies less than one kilobyte.

In a naïve attack instantiation, each additional gadget will increase the overall code size. To counter this potential limitation, we can also embed smaller gadgets into the bigger ones by stacking `if` statements inside the body of another `if` statement, ideally reducing the size of the JavaScript code to the size of the biggest gadget.

Computing addresses of JavaScript functions: In order to use generated gadgets, we have to compute their addresses. We start by revealing the address of the JIT-compiled JavaScript function, which contains emitted gadgets, by employing a memory disclosure vulnerability. We can do this, for example, by passing the function as a parameter to another one, thus pushing its value on the stack. Afterwards, in the callee, we read the stack, revealing the pointer to the function's JavaScript object, which contains the code pointer to the actual (JIT-compiled) function. Note that here we assume that we know the location of the stack. This can be done by chasing the data pointers in the readable memory, until we find a pointer pointing to the stack.

Because of the predictable code output of JIT compilers, we know the offsets inside the JIT-compiled function, at which conditional jumps will be emitted and can thus compute the addresses of emitted gadgets.

4.2 Conditional Jump Gadgets in Browsers

We tested this technique against three modern browsers: Chrome 33 (32-bit)/Chrome 51(64-bit), Firefox 42 (64-bit) and IE 11 (64-bit with 32-bit JavaScript engine). There are some differences that need to be taken into account for each of them. For example, Chrome compiles JavaScript functions the first time they are called, while Firefox and IE interpret them a few times until they are called too often (e.g., around 50 times for IE and 10 times for Firefox) and only then JIT-compile the JavaScript code. Therefore, to trigger the compilation we just call the function multiple times and then wait until it is compiled (which takes a few milliseconds).

Chrome: As each browser has its own JIT compiler, an attacker has to vary the JavaScript code to fill the exact number of bytes in the `if` body. This is just a matter of finding a mapping between JavaScript statements and the number of bytes of their JIT-compiled equivalent. We will demonstrate this by emitting a system call gadget (`int 0x80;ret`) in 32-bit Chrome. To this end, we need to emit `0xcd80c3`, i.e., we need to fill the `if` body with JavaScript code that is JIT-compiled to `0xc380cd` bytes. We use the following two JavaScript statements:

S1: `v=v1+v2`, compiling to `0x10` bytes, and
S2: `v=0x01010101`, compiling to `0xd` bytes.

By combining these two statements, we can generate arbitrary gadgets. In our case, we use S1 `0x0c 38 0c` times (resulting in `0xc3 80 c0` bytes) and S2 once—summing up to `0xc3 80 cd`, our desired gadget.

Note that the JavaScript statement that compiles to `0x10` bytes allows us to control each hex digit of the emitted jump distance except the last one (i.e., the least significant half-byte of the gadgets' first byte). Moreover, any JavaScript statement that compiles to an odd number of bytes allows us to control the least significant half-byte of the distance. Combining these two properties, we can generate any gadget by using these two selected JavaScript statements multiple times.

The sizes of JIT-compiled JavaScript statements differ in 64- and 32-bit versions of Chrome. In 64-bit Chrome we replace S1 with `v=v`, which is compiled to `0x10` bytes. Note, however, that even though the size of S2 in 64-bit is also changed to `0x1b` bytes, we can still use it because it is compiled to an odd number of bytes.

Firefox: To generate arbitrary gadgets for Firefox, we choose the following two statements:

S1: `v=v`, compiled to 8 bytes (two of them to `0x10`), and
S2: `v+=0x1`, compiled to `0x21` bytes.

```
0x0000: call FUN_1      ; 0xe8fb1f0000
0x0005: call FUN_1      ; 0xe8f61f0000
0x000a: ...
0x2000: push ebp        ; 0x5d(@FUN_1)
```

Figure 2: Direct call

```
function js_call_gadget() {
  asm_call();  /* emits a call */
  asm_call();
  /* ... (many asm_call() statements) */
  asm_call();  }
```

Listing 1: JavaScript function *js_call_gadget*

IE: IE deploys JIT-hardening mechanisms that go beyond the protections in Chrome and Firefox. IE (i) has a size limit on code segments generated by the JIT compiler, and (ii) randomly inserts NOPs (i.e., instructions that do not change the program state) in JIT-compiled code. Because of (i), we can only emit two-byte gadgets. Due to (ii), these gadgets are then further modified by inserting NOPs inside the `if` body and, thus, changing the value emitted in the conditional jump. The latter technique is similar to librando [17]. Nevertheless, even with these defenses in place, we can still emit arbitrary two-byte gadgets by measuring the size of the emitted code at run time. We will describe this attack in Section 4.4 in the discussion about Internet Explorer.

4.3 Direct Call Gadgets

We found that conditional jumps are not the only instructions to embed implicit constants that can be indirectly controlled by the attacker. Direct calls (e.g., `call 0x1234560`) are another example of such instructions. In our second approach, we leverage the JavaScript statements that are compiled to instructions containing direct calls.

Direct call constants: Direct calls in x86/x64 change the execution flow of the program by modifying the instruction pointer (`eip/rip`). The constant encoded in a direct call instruction represents a relative address of the callee. That is, the call instruction's displacement field contains the distance between the addresses of the instruction following the call and the callee. Therefore, any two direct call instructions to the same function will encode different constants. For example, in Figure 2, there are two consecutive calls to the function `FUN_1` (at address `0x2000`). The constant encoded in the first call denotes the distance between `FUN_1` and the instruction following the call (i.e., the second call at `0x05`). Therefore, its value is `0x2000-0x5=0x1ffb`, which is `0xfb1f0000` encoded in little-endian.

In the example above, the difference between two consecutive direct call constants is `0x5` (the size of a direct call instruction). In general, the difference is equal to the size of the instructions between two consecutive calls. In our case, we want to use JavaScript statements to emit direct calls in the JIT-compiled code. Therefore, the difference between the constants will be the size of the instructions in which the JavaScript statement is compiled.

To generalize this attack vector, we aim for a JavaScript function similar to `js_call_gadget` (Listing 1). The `asm_call()` statement is a placeholder for any JavaScript statement (not necessarily a function call) that is compiled into a sequence of instructions containing a direct call. The exact statement that replaces the placeholder depends on the target browser.

Finding callee address: Let our goal be to emit a three-byte gadget and fix its third byte to `0xc3` (`ret`). To calculate the constant encoded in the displacement field of a direct call instruction, we have to know the addresses of the call instruction and its destination. The destinations of the emitted call instructions that we have encountered are either helper functions (e.g., inline caches generated by V8) or built-in functions (such as `Math.random` or `String.substring`). The helper functions are JIT-compiled by V8 as regular functions. We can leak their addresses either by stack reading (e.g., by leaking the return address put there by the call instruction inside the helper function), or by reading the V8's heap, where all the references of compiled helper functions are stored. In IE, the built-in functions are located in libraries and thus are randomized via fine-grained ASLR schemes [12, 11, 17]. However, their corresponding JavaScript objects (e.g., `Math.random`) contain the code pointer to the function. Knowing the structure of these JavaScript objects, which are not randomized according to our assumptions, we can get the addresses of built-in functions via a memory disclosure vulnerability. Note that after code pointer hiding, the addresses that the attacker leaks from these JavaScript objects will be the addresses of the trampolines and not the actual functions. Nevertheless, offsets, encoded in call (or jump) instructions, will also be computed relative to the trampolines and thus can be used for calculating emitted constants.

Emitting call instructions: Knowing the address of the callee, the next step is to emit direct call instructions at the correct distance. Given that we cannot influence the address where the function will be compiled, we have to acquire sufficiently large code space to cover all three-byte distances to the callee. To this end, we create a JavaScript function that spans `0x1 00 00 00` bytes after JIT compilation and consists of JavaScript statements emitting direct calls. More precisely, we require the distance between the first and the last emitted direct call

instructions to be at least 0x1 00 00 00 bytes. This way, regardless of where our function is allocated, we will be guaranteed that it covers all possible three-byte distances from the callee, allowing us to emit arbitrary three-byte gadgets by carefully placing direct call instructions.

Emitting required gadgets: Creating such a large function (16 MB) emits many three-byte gadgets, and also covers all two-byte gadgets. For example, if we have a JavaScript statement that generates a call instruction and is compiled to 0x10 bytes of native code, we can create a big function containing this statement 0x10 00 00 times. The compiled function will have 0x10 00 00 direct call instructions 0x10 bytes apart. If we consider the least significant three bytes of the emitted displacement fields of these direct calls, they will have the following form: 0x*Y ** **, where * denotes any hexadecimal digit [0-f] and Y is a constant, which encodes the least significant half-bytes of emitted values.

Because of the little-endian format used in x86/x64 architectures, Y is part of the first byte of emitted gadgets. Therefore, to emit three-byte gadgets, we must be able to set Y accordingly. To this end, we modify the value of Y by varying the size of the instructions before the first direct call. That is, we find any JavaScript statement that compiles to an odd number of bytes, and then use it up to 15 times to get any out of all possible 16 half-bytes. For example, if the least significant half-byte of the call instruction is 0x0 and we want to make it 0xd, and we have a JavaScript statement that compiles to an odd number of bytes (e.g., i+=1 in 32-bit Chrome, 0x13 bytes), we use this statement 15 times (0x13*15=0x11d).

Computing addresses of emitted gadgets: Assuming that the address of the first call instruction in our function is $\texttt{F}_{call}$, and the address of the callee is $\texttt{F}_{dest}$, we can compute three bytes of the displacement field of the first call instruction by $\texttt{C}_1=\texttt{F}_{dest}-(\texttt{F}_{call}+5) \bmod 2^{24}$. If the required gadget is G, then we can compute the distance (dist) between the call instruction, emitting G, and the first call instruction: $\texttt{dist}=\texttt{C}_1-\texttt{G} \bmod 2^{24}$. Using dist, we can calculate the address of the call instruction emitting G ($\texttt{F}_{call}$+dist), and therefore the address of the gadget ($\texttt{G}_{addr}$) which is located 1 byte after the call: $\texttt{G}_{addr}=(\texttt{F}_{call}+\texttt{dist})+1$.

4.4 Direct Call Gadgets in Browsers

We will next discuss techniques that we use to instantiate the attack in three popular browsers.

Firefox: Emission of direct call gadgets is not possible in Firefox, as the baseline JIT compiler of Firefox does not emit direct calls. Although the optimizing JIT compiler of Firefox emits direct calls, e.g., when compiling regular expressions, it only optimizes JavaScript functions after they have been executed more than 1,000 times. Triggering the optimizing compiler on the large functions (as required for our attack) thus makes our attack impractical against Firefox.

Chrome: Chrome compiles most JavaScript statements to direct calls. Consequently, we have a large selection of JavaScript statements with varying post-compilation sizes. We use a statement that is compiled to 0x10 bytes of assembly code (e.g., i=i+j for 32-bit Chrome). For demonstration purposes, we aim to emit a system call gadget (int 0x80;ret), implicitly also revealing all two-byte gadgets. To this end, we create a function shown in Figure 3(A). The function starts with a sequence of JavaScript statements that align the first call instruction to 0xe. After this, the emitted call distances will be calculated relative to 0x3, i.e., (0xe+0x5) mod 0x10, where a direct call is 0x5 bytes large. Considering that the callee is at least half-byte aligned, the lower half-byte of all emitted gadgets' first bytes will be 0xd, i.e., (0x0-0x3) mod 0x10. The alignment code is followed by a sequence of call-generating statements (e.g., i=i+j), each of which compiles to 0x10 byte-long code. The compiled i=i+j statement emits a call instruction at offset 0x*e (due to alignment) as shown in Figure 3(B). Generating a sequence of 0x10 00 00 call instructions, we are guaranteed to have an int 0x80;ret gadget encoded into one of the call instruction constants (Figure 3(C)).

Note that the aforementioned technique is used in the 32-bit version of Chrome. For 64-bit, we use the v++ statement, which is compiled to 0x20 bytes (instead of 0x10) and emits a call instruction. Having 0x20 bytes between call instructions changes the upper half of the least significant byte. For example, after aligning the least significant half-byte to 0xd via padding, the emitted first bytes will be either 0x{0,2,4,6,8,a,c,e}d or 0x{1,3,5,7,9,b,d,f}d, depending on the initial value of the upper half of the least significant byte. We can modify this value to our liking by adding the i=i statement, which is compiled to 0x10 bytes, as padding.

Internet Explorer: For the two main reasons mentioned in Section 4.1 (code size limit and NOP insertion), emitting gadgets is harder in IE. The per-function code size limit forbids us to emit 0x1 00 00 00 bytes of native code, which is required to span all possible third bytes of the constants encoded in call instructions. However, in the following, we describe how an attacker can still encode gadgets in direct calls even in IE.

Emitting calls at correct distance: IE still allows us to create many small functions. These functions will be distributed in the set of pages, each of them being 0x2 00 00 bytes large. We thus allocate many functions (200 in our case), each of them being $\lesssim$0x1 00 00 bytes (i.e., two functions per page). Given the alignment (0x1 00 00) and the size (0x2 00 00) of the spanned code pages, each page will cover two third-bytes of

```
function js_call_gadget_v8(){
  var i,j;
  // Align calls to 0xe
  i = i + j;

  // i=i+j 0x3a86b times
  i = i + j;

  // i=i+j 0xc5793 times
  i = i + j;
}
```

(A) JavaScript function

```
0x*0: ...
0x*7: ...
0x*e: call BINARY_OP_IC              ;e8 8d07feff
...
0x*7: ...
0x*e: call BINARY_OP_IC              ;e8 cd80c3ff
...
0x*7: ...
0x*e: call BINARY_OP_IC              ;e8 8d07fefe
```

(B) i=i+j direct calls (C) Bytes emitted by direct calls

Figure 3: JavaScript function emitting gadgets via direct call constants

the absolute address completely (e.g., from 0x12 34 00 00 to 0x12 35 ff ff). Considering that the callee is not aligned to the same boundary, direct calls emitted in these pages will have three distinct third bytes in their constants, only one of them covered completely. For example, if we assume the callee to be at address 0x12 34 56 70 and the page emitted at 0x01 70 00 00, only the call instructions located in the address range [0x01 70 56 70, 0x01 71 56 70] can emit complete three-byte constants, having 0xc3 as their third byte, i.e., constants from 0x12 34 56 70 - 0x01 70 56 70 = 0x10 c4 00 00 to 0x12 34 56 70 - 0x01 71 56 70 = 0x10 c3 00 00. On the other hand, the ranges [0x01 70 00 00, 0x01 70 56 70] and [0x01 71 56 70, 0x01 71 ff ff] cover only parts of the constants, with 0xc4 and 0xc2 as their third bytes.

After allocating the functions, we dynamically check their addresses to find the one with the correct distance from the callee (using the same technique as described at the end of Section 4.1), i.e., the one having the correct third byte in its direct call instruction's displacement field. Allocating 200 JavaScript functions, each of them containing 0x1 00 00 bytes, is inefficient, especially if the code has to be downloaded to the victim's machine. Therefore, we use `eval` to spam IE's code pages with the dynamically created functions. The problem with evaluated functions is that IE does not emit direct call instructions in them and uses indirect calls instead. Therefore, we use these functions only as temporary placeholders. Once we find any evaluated function at the correct place, we deallocate it to make its place available for the subsequently compiled functions. To deallocate a JavaScript function, we set `null` to all of its references and wait until the garbage collector removes it (typically within less than a second).

Verifying emitted gadgets: At first sight, IE's NOP insertion conflicts with our assumption about the predictability of JIT-compiled code. With NOP insertion, and likewise with many other fine-grained code randomization schemes, we cannot guarantee that the call instruction, which is supposed to emit the gadget, ends up at the correct address. However, because NOPs are inserted at random, compiling the same JavaScript function multiple times actually *increases* the chance that in one of the compiled versions, the `call` instruction ends up at the correct place.

```
function js_call_gadget_IE() {
  // Padding to correct address
  var i = Math.random(); // emit direct call
  check_address(<random cookie value>);
}
```

Listing 2: JavaScript function *js_call_gadget_IE*

Following our threat model, though, we cannot read executable code segments to verify if the compiled call instruction is at the desired place. As an alternative, we read the stack, as shown in Listing 2. In `js_call_gadget_IE()`, the statement `i=Math.random()` emits a direct call. We pad the beginning of the function with a few JavaScript statements to place `i=Math.random()` at *approximately* the correct address, such that the relative address would encode the desired gadget, accounting the randomness induced by NOP insertions. We then check the correctness of the position via `check_address`, a JavaScript function that reads the stack to find the return instruction pointer put there by the `call` instruction.

Using the leaked return address, we can calculate the address of the direct call instruction emitted by `i=Math.random()`, verify that it is at the correct place and if so, use it as a gadget. A simple implementation of `check_address` is shown in Listing 3, where it uses a memory disclosure vulnerability (`mem_read` in this case) to read the stack from some starting point (`ESP_`), until it finds its own parameter (`cookie`). The parameter is a random number, reducing the chance that multiple positions have the same value (note that this chance can be further reduced by using multiple random parameters). After finding the cookie on the stack and thus the address of the parameter, we know the exact offset

```
function check_address(cookie) {
  // ESP_: Any address on stack
  // NEEDED_ADDRESS: address where
  //                 call must reside
  while(mem_read(ESP_) != cookie)
    ESP_ -= 4; // Check next value
  // get return address from parameter
  var ret_ = mem_read(ESP_ - 0xC);
  // get call instruction address
  var call_addr = ret_ - 5;
  return call_addr == NEEDED_ADDRESS;
}
```

Listing 3: JavaScript function *check_address*

from the parameter's address, and from there we can tell where the return address is located. Reading the return instruction pointer, we recover the address of the corresponding call instruction and verify that it is at the correct place (`NEEDED_ADDRESS` in this case). We can add another call to `check_address` before `i=Math.random()` to verify if NOPs are inserted between the emitted call instructions of `i=Math.random()` and `check_address()`. If both checks (`check_address`) succeed, we will be guaranteed that no NOPs were inserted.

IE summary: An attacker can evade both aforementioned defenses and emit three-byte gadgets even in IE. To demonstrate this, we first dynamically create many JavaScript functions to get the correct third byte (`0xc3`). After finding the function at the correct distance from the callee, we replace it with a special function, which, after compilation, emits direct call instructions and checks their positions. We trigger the recompilation of the latter function multiple times, until the checks are true, which means that the gadget is found. In our experiments, spamming the code pages with functions took approximately four seconds, and for most of the time, we found the correct third byte on the first try. Triggering the recompilation of a function takes the following steps: (i) Remove the included JavaScript file from the head of the HTML file, (ii) wait until the function gets removed by the garbage collector, and (iii) include the JavaScript file again and trigger the compilation of the same function. Each iteration of the above steps takes around 2 seconds, most of the delay coming from the second step (waiting for the garbage collector).

For two-byte gadgets, on the other hand, an attacker can discard the third byte. In this case, she can directly compile multiple functions at once, check the positions of emitted direct calls and use the ones with the required displacements.

4.4.1 Proof-of-Concept Gadget Generation

To demonstrate the practicality of the aforementioned gadget emitting techniques, we crafted a special JavaScript code for Chrome and IE, which generated the gadgets required for the exploit. The gadgets that we aimed to generate are the ones used by Athanasakis *et al.* Namely, the set of gadgets to load the registers with the arguments used in a system call and one for the system call itself. We created these gadgets in Chrome 51 (64 bit) and IE 11 (32 bit).

Chrome: For Chrome, we targeted for the following instructions: `pop r8; pop r9; pop rcx; pop rdx` (to prepare the system call arguments) and `int 0x80` (to execute the system call). Being able to emit three-byte gadgets, we encoded these instructions into the following gadgets:

```
pop   r8, ret   ; 4158c3
pop   r9, ret   ; 4159c3
pop  rcx, ret   ;   59c3
pop  rdx, ret   ;   5ac3
int 0x80, ret   ; cd80c3
```

We used both our proposed techniques for the emission of these gadgets. We generated a system call gadget via direct calls. First, we created a string representation of a JavaScript function containing `0x80000 j++` statements (`j++` takes `0x20` bytes), then we created a JavaScript function from it via `eval`, and finally we compiled it by calling the generated function. This gave us a system call gadget, together with all possible two-byte gadgets, hence also covering `pop rcx` and `pop rdx`.

For the generation of `pop r8` and `pop r9` gadgets, we used cascaded `if` statements (also created with `eval`). The JavaScript function generating the aforementioned gadgets is shown in Listing 4. As gadgets `pop r8` and `pop r9` differ by `0x100`, their corresponding `if` statements also have to be `0x100` bytes apart. Note however that in the first `if` body (`F1`), we add `0xed` bytes to fill up the space instead of `0x100`. This is due to the fact that an `if` statement is compiled to `0x13` bytes, which is also added to the distance between relative jumps. To get `0xed` bytes, we use `j=0x1010101` 7 times (`0x1b*7=0xbd`) and `j++;j=i;j=i` (`0x20+0x8+0x8=0x30`). To generate `0xc35841` bytes (`F2`), we use `j=0x1010101 0xd3` times (`0x1b*0xd3=0x1641`), and fill the remaining `0xC34200` bytes by using `j++ 0x61A10` times (`0x20*0x61A10=0xC34200`).

The entire gadget generation process in Chrome took ≈1.3 seconds, in a VirtualBox Virtual Machine running Windows 10 (Intel Core i5-4690 CPU 3.50GHz).

Internet Explorer: As we have mentioned earlier, Internet Explorer, by default, comes with a 32-bit JIT compiler. Therefore, for gadget generation we chose gadgets that would be used in a 32-bit system. For simplicity

```
function popr8r9(r8,r9) {
  var i=0,j=0;
  if(r9) {
  /* F1: fillup 0xed Bytes */
    if (r8) {
    /* F2: fillup 0xc35841 Bytes */
    }
  }
}
```

Listing 4: JavaScript function *popr8r9*

```
function syscallIE() {
  var i=0;
  i=Math.random();
  ... /* 240 times in total */
  check_address();
  i=Math.random();
  ... /* 10 times in total */
  return i;
}
```

Listing 5: JavaScript function *syscallIE*

we used the set: `popa; int 0x80`, where `popa` sets the contents all x86 registers from the stack and `int 0x80` performs the system call.

The first part of the gadget emission process in IE is finding the right distance from the callee, i.e., a page that is `0xc3` bytes away from the callee. This part was done by a JavaScript code, which simply creates and compiles big functions (in our case 200 of them, ≈`0x10 000` bytes each). After finding the correct page, we deallocated it and spammed the page with 16 specially crafted JavaScript functions, each of them covering `0x1000` bytes. For example, the JavaScript function used for emitting a system call (Listing 5) contains 250 `Math.random` calls (each of them compiling to `0xc` bytes). At the correct place between these calls, i.e., when the caller is at approximately the correct distance from `Math.random`, we inserted a call to `check_address` to verify the correctness of the gadget. In case the emitted call is not at the correct place, we deallocated the function and reallocated it again. Note that the reallocation is only needed for three-byte gadgets, where we also want to control the least significant byte. For two-byte gadgets (e.g., for `popa;ret`), we only need to call `check_address` to compute the address of the call instruction, for which we already know that is at the correct place (Listing 6).

In comparison to Chrome, gadget generation in IE is probabilistic and thus the time required for it also differs. There are two sources of the variance. First, generating the large functions to search for the correct third-byte distance from the callee; and second, compiling the gadget-emitting function in the found (correct) page, and

```
function poparetIE() {
  var i=0;
  i=Math.random();
  ... /* 232 times in total */
  check_address();
  i=Math.random();
  ... /*  28 times in total */
  return i;
}
```

Listing 6: JavaScript function *poparetIE*

Defense	Chrome	Firefox	IE
Const. Blinding	✓	×	✓
NOP Insertion	×	×	✓
Code Size Limit	×	×	✓

Table 1: Current defenses in modern browsers

recompiling it until the correct gadget is emitted. In our experiments, we created 200 large functions and got the required third-byte distance for the first time in most of the cases. Compilation of these 200 functions took ≈4 seconds on a physical machine running Windows 10 (Intel Core i5-6200U CPU 2.3GHz). Each recompilation in the second step took 2-3 seconds. We ran the gadget generator in IE 10 times. Generating `popa; ret` and `int 0x80; ret` took on average 32 seconds, 11 and 47 seconds being the fastest and the slowest respectively.

4.4.2 Summary of Defenses and Vulnerabilities

We have shown that an attacker can encode arbitrary gadgets by triggering implicit constants with specially-crafted JavaScript code. Combining this with the ability to leak code pointers, an adversary can guess the addresses of the emitted gadgets without reading any code, thus making the attack possible even if code pages are non-readable.

Table 1 summarizes the defense techniques of modern browsers against code-reuse attacks in JIT-compiled code. Both IE and Chrome deploy constant blinding. Furthermore, IE uses NOP insertion as a fine-grained code randomization scheme, as also suggested in librando [17]. However, as Table 2 shows, none of the modern browsers sufficiently protect against the proposed attacks. Only Firefox "avoids" implicit constants by not using direct calls in baseline JIT compiler, but still exposes implicit constants in relative jumps.

[1] Gadgets up to two bytes can be emitted.

Attack	Chrome	Firefox	IE
Relative Jumps	↯	↯	↯[1]
Direct calls	↯	–	↯

Table 2: Browsers vulnerable to implicit constants

5 Defense

Seeing the threat of implicit constants, we now propose a technique to defend against it. We identify two steps that the attacker needs to take for using implicit constants as gadgets: (i) The attacker must be able to emit the required gadgets, and (ii) she must be able to acquire the necessary information (e.g., leak function pointers) to compute the addresses of the emitted gadgets.

One solution to tackle this problem would be to hide code pointers, e.g., by extending Readactor(++) to also cover the JIT-compiled code, as Crane *et al.* suggested. This would hinder the attacker from executing step (ii). However, this would still allow the attacker to emit arbitrary gadgets by leveraging the implicit constants (step (i)). Furthermore, the fact that the JIT compiler runs in the same process as the attacker makes it challenging to remove all possible code pointers that could, directly or indirectly, reveal the addresses of emitted gadgets. Therefore, we propose an orthogonal defense technique that forbids the attacker to emit the gadgets in the first place (i.e., step (i)). Our defense could be complemented with holistic code pointer hiding techniques to get additional security guarantees.

The main idea of our defense can be split in two parts: (i) We convert direct calls and jumps into indirect ones, such that their destination is taken from a register, and (ii) we use constant blinding to obfuscate the constants that are emitted by step (i) and may potentially contain attacker-controlled gadgets. For step (ii), we use the same cookie that is used by V8 to blind integer constants, and is generated anew before the compilation of each function. Note that the cookie is encoded in non-readable code and cannot be leaked. However, even if the attacker was able to leak the cookie, she could only guess the immediate values emitted in the current function, and any future function will have a different cookie value.

5.1 Removing Implicit Constants from V8

We integrated our defense into V8, Chrome's JavaScript engine. We have chosen V8 due to its popularity and due to the fact that it is vulnerable to both our suggested attacks. Moreover, since V8 JIT-compiles JavaScript directly to the native code, it emits many checks (conditional jumps) and function calls (e.g., calls to inline caches), which makes V8 a suitable candidate for our defense prototype evaluation. For our defense technique, we changed the functions of V8 that are responsible for emitting native code. In total, we modified ≈200 lines of code to account for all the cases of attacker controlled relative calls or jumps.

5.1.1 Conditional Jumps

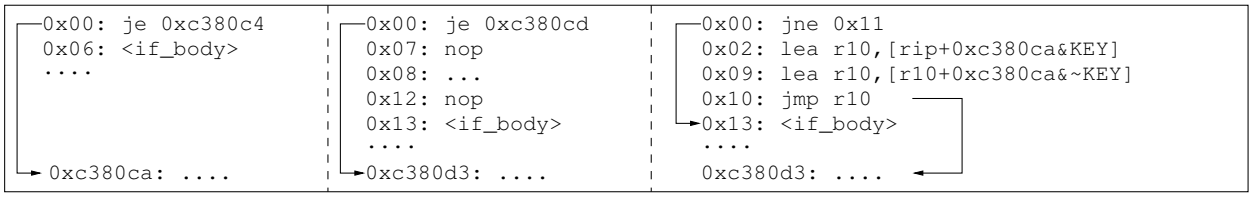
To harden conditional jumps, we modified the native code that is emitted when JavaScript conditionals (such as `if`,`while`,`for`,`do-while`) are compiled. Our basic idea is to switch from relative to absolute jumps, and blind the resulting immediate values. To this end, we first add a padding (a sequence of NOP instructions) to each compiled conditional to reserve the space for later changes. For the hardened version of the conditional jump we need 19 bytes (instead of 6 bytes). We thus append 13 NOP instructions after the existing conditional jump. At the end of the compilation, when the constants of all jumps are calculated, we convert all relative jumps to absolute jumps, eliminating the need to fill a displacement with potential gadgets.

Figure 4 illustrates the steps of the aforementioned modifications. Figure 4(A) shows the compiled `if` statement in original V8. Figure 4(B) shows the same statement with the NOP padding. Finally, Figure 4(C) shows the assembly of the hardened `if` statement. In this final form, the condition of the original jump is inverted and the original long jump (having 4 byte jump distance) is replaced with the 1-byte short jump. Consequently, the new jump is taken if the original condition was false, i.e., the fall-through case. Otherwise, we convert the relative address into the absolute one, by adding it to the current instruction pointer (`rip`). This can be done with a single instruction in x64 (`lea r10,[rip+0xc380ca]`).

As this instruction will still emit the relative address as the displacement, we split it in two instructions. First, we add the current instruction pointer to the relative address AND-ed with a random key (`rip+0xc380ca&KEY`). In the second `lea` instruction, we add the sum to the relative address AND-ed with the inverted (bitwise not) random key, resulting in the desired offset (`rip+0xc380ca`). Note that we use obfuscation by AND-ing the constant with a random key instead of XOR-ing it, because $(A+B \oplus C) \oplus C$ does not equal to $A+B$, while $(A+B \wedge C)+B \wedge \neg C$ does. Moreover, this obfuscation scheme allows us to use `lea` instructions only, which has the advantage of not modifying any flags.

5.1.2 Direct Calls

We mitigate the implicit constants in direct calls by converting the direct calls into indirect ones. To this end, we



(A) Original V8 (B) NOP Padding (C) Modified V8

Figure 4: Steps of JIT hardening in V8

```
0x00:  lea  r10, [rip+ADDRESS&KEY ]
0x07:  lea  r10, [r10+ADDRESS&~KEY]
0x0e:  call r10  ; calls 0xc380d3
0x11:  ....
```

Figure 5: Hardening direct calls

distinguish whether the address of the callee is known at compile time (e.g., when calling built-in functions). If the callee's address is known, we can move it to a scratch register (`r10`) and then execute an indirect call `mov r10,ADDRESS;call r10`. We thus emit the absolute address of the callee as the immediate value of the `mov` instruction, which is not under the control of the attacker and is thus safe—in contrast to the relative address.

If the address is unknown at compile time, we use a similar technique as we did for the conditional jumps, i.e., we convert the relative address into an absolute one (blinding the relative address during the conversion), store it in the scratch register, and then execute an indirect call, as shown in Figure 5.

5.2 Evaluation

To evaluate our defense technique we ran the V8 Benchmark Suite 7 on our modified V8. We performed each benchmark 100 times on both the modified and original V8s, and compared their corresponding averaged results. Table 3 illustrates the average scores that were returned by the benchmark suite, where a higher score indicates better performance. The modified V8 has an average overhead of less than 2%, and the worst overhead less than 3%. The observation that the overhead is negative for the *NavierStokes* benchmark can be explained by statistical variations across the different runs.

Additionally, we tested the modified V8 with microbenchmarks. To this end, we created two JavaScript functions (`ifs_true` and `ifs_false`), both of them containing 1,000,000 `if` statements. The condition of the `if` statement in `ifs_true` is always `true` (i.e., the `if` body is executed), while the condition of `ifs_false` is always `false`. This way the JIT-compiled functions will contain 1,000,000 conditional jump instructions modified by us, each of them testing separate execution paths. Furthermore, evaluation of the expression in the `if` statements is done via a function call. Therefore, both of these functions generate 1,000,000 modified call instructions each and will thus incorporate the overhead caused by the function calls. Each run of the microbenchmark calls each of these functions 10 times. We ran the benchmark 1,000 times. We distinguish the first execution of these functions from the remaining nine, as the first execution is significantly slower due to the JIT-compiler modifying the generated intermediate functions to adjust them to the type information. Because the overhead was dominated mostly by the compiler, we did not see any overhead for the first function execution. For the remaining function executions we had 14,25% overhead in `ifs_false` and 9,81% for `ifs_true`.

Besides computational performance, our defense technique also causes a memory overhead due to added code. To measure this overhead, we compared the sizes of the functions compiled by the original and the modified versions of V8. To get the needed output from V8, we ran it with the `--print-code` flag, which outputs the disassembled code for each function after the compilation together with additional information about the compiled function including the size of the generated instructions. Running the benchmark suite with the aforementioned flag yielded that the total size of the instructions emitted by the original V8 was 1,123 kB, while the modified V8 emitted 1,411 kB, giving 287 kB of additional code, i.e., ≈26% code size overhead. Given the significant size of the benchmark suite, and given that memory of nowadays x86/x64 systems are typically in the range of gigabytes, we think that hundreds of kB of additional code does not cause any bottlenecks on COTS systems.

6 Discussion

6.1 Defense Security Considerations

Our defense follows the general goal to remove unintended gadgets from constants in JIT-compiled code. We tailored our defense implementation towards protecting jump and call offsets. Other offsets may be usable to en-

Benchmark	Original	Modified	Overhead(%)
Richards	36,263	35,555	1.95
DeltaBlue	63,641	62,045	2.51
Crypto	33,366	32,725	1.92
RayTrace	77,198	75,488	2.21
EarleyBoyer	44,900	43,700	2.67
RegExp	6,525	6,414	1.71
Splay	21,095	20,479	2.92
NavierStokes	31,924	31,998	-0.23
Total	**32,255**	**31,662**	**1.96**

Table 3: Scores by the V8 benchmark (higher is faster)

code further gadgets. For example, relative addressing is frequently used in combination with the base pointer, such as when accessing parameters of a JavaScript function. As parameters are stored on the stack, they are accessed relative to the frame pointer (`ebp/rbp`). Each parameter access, after JIT-compilation, emits an assembly instruction, which contains the offset of the parameter from the frame pointer in its displacement field: `mov [ebp+0x0c],0x1`. The number of possible gadgets, in this technique, is restricted by (i) limited stack size (e.g., maximum $2^{16}-1$ (`0xffff`) function parameters in Chrome) and (ii) stack alignment (4 or 8 bytes). In combination, this only allows generating gadgets whose opcodes are multiples of 4 (or 8) and are in the range between `0xc` and `0x40000`, and thus gives the attacker only limited capabilities. The stack size restrictions impose the same limitations on implicit gadgets encoded in relative accesses to function's local variables

While we think that the most important constants are blinded, we cannot exclude the existence of further ways to encode gadgets in assembly instructions. To eradicate all potential gadgets, one could prevent the JIT compiler from creating any potential gadgets (even in unaligned instructions). Most notably, G-Free [24] is a gadget-free compiler, which tries to generate gadget free binaries. However, G-Free requires multiple recompilations and code adjustments to reliably remove all possible gadgets. This will increase the runtime overhead for the JIT compilers, as the compilation time is included in their runtime.

6.2 Fine-Grained Code Randomization

An orthogonal approach to our defense would be to remove the attacker's capability to find the gadget's location (i.e., address). One way of doing so would be to hide code pointers, e.g., via trampolines, as suggested by Crane *et al.* [11]. If code pointers are not hidden, the attacker can read the return instruction pointer on the stack to get a pointer to the created gadget—which represents the current status in XnR implementations. This results in (i) getting access to the gadget, (ii) a possibility to verify the gadget at runtime, and (iii) the ability to retry in the case of a false result. By using similar techniques as we used against NOP insertion, the attacker can defeat fine-grained code randomizations such as the ones underlying Readactor [11]. Even though current XnR implementations do not hide code pointers in JIT-compiled code, XnR's ideal implementation could also expand fully to the JIT-compiled code, e.g., by introducing trampolines. This, together with the fine-grained randomization schemes such as NOP insertion, would successfully protect against our attack. Note, however, that NOP insertion does not remove gadgets, but tries to reduce the chances of the attacker to guess their locations. In contrast, our proposed defense technique removes the gadgets, hence also removing the risk of the attacker doing a guesswork. Combining our technique with the extended XnR implementation would further improve the security guarantees, removing the chances of both emitting the gadgets and leaking the code layout information.

To guard against JIT-compiled gadgets, Wei *et al.* proposed to do several code modifications such as (i) securing immediate values via constant blinding, (ii) modifying internal fields of the instruction (e.g., registers being used), and (iii) randomizing the order of the parameters and local variables to randomize the offsets emitted by them [34]. However, this is not effective against the attacks proposed in Section 4, as the modifications do not secure the displacement fields emitted by relative calls/jumps. Finally, the code randomization proposed by Homescu *et al.* [17] that adds NOP instructions to randomize the code output from the JIT compiler remains ineffective if code pointers in JIT-compiled code are not hidden.

6.3 Attack Generality

A natural question is how the proposed attack generalizes, in particular to other operating systems or CPU architectures. We have evaluated the attacks against Chrome and Firefox running on Linux and IE on Windows. As we exploit properties of the JIT compilers to generate desired gadgets, the choice of the underlying operating system is arbitrary. The proposed attacks rely on the x86 system architecture (32- or 64-bit), though. In RISC architectures, such as ARM and MIPS, instruction lengths are fixed, and execution of unaligned instructions is forbidden by the hardware. However, the attacks may still apply to ARM, as an attacker could emit arbitrary two-byte values in the code if she can force the program to switch to 16-bit THUMB mode. Although this limits the attacker to using a single instruction, it still allows setting the register contents and diverting the control flow

at the same time, e.g., by using a `pop` instruction.

We implemented our defense in the 64-bit version of V8, taking advantage of the x64 architecture's ability to directly read the instruction pointer (`rip`). This simplified the effort of converting relative addresses into absolute ones. Even though one can read the instruction pointer indirectly in 32-bit, e.g., by `call 0x0;pop eax`, such additional memory read instructions would increase the performance overhead. In addition, 32-bit features only eight general-purpose registers. While in x64 we could freely use a scratch register (`r10` for Chrome), in x86 we would likely need to save and restore the register. Similar defenses in x86-32 are thus possible, but come at an additional performance penalty. However, given that 64-bit systems are increasingly dominating the x86 market, we think that 64-bit solutions are most relevant.

7 Related Work

In the following, we will summarize existing code-reuse attacks and proposed defense mechanisms.

7.1 Code-Reuse Attacks: ROP / JIT-ROP

The most widespread defense against ROP is ASLR [32], which randomizes the base addresses of memory segments. Although it raises the bar, ASLR suffers from low entropy on 32-bit systems [30] and is not deployed in many libraries [28]. In addition, ALSR does not randomize within a memory segment, and thus leaves code at fixed offsets from the base address. Attackers can thus undermine ASLR by leaking a code pointer [18].

Researchers thus suggested fine-grained ASLR schemes that randomize code within segments. Fine-grained ASLR hides the exact code addresses from an attacker, even if a base pointer was leaked. For example, Pappas *et al*. [25] suggest diversifying code within basic blocks, such as by renaming and swapping registers, substituting instructions with semantically equivalent ones, or changing the order of register saving instructions. ASLP, proposed by Kil *et al.* [20], randomizes addresses of the functions as well as other data structures by statically rewriting an ELF executable. To increase the frequency of randomization, Wartell *et al.* propose STIR [33], which increases randomness by permuting basic blocks during program startup.

However, the invention of JIT-ROP undermined fine-grained ALSR schemes [31]. JIT-ROP assumes a memory disclosure vulnerability, which can be used by the attacker repeatedly. The attacker then follows the pointers to find executable memory, which she can read to find gadgets and build ROP chains on-the-fly.

Recently, Athanasakis *et al.* [1] proposed to extend JIT-ROP-like attacks by encoding gadgets in immediate values of JIT-compiled code. Despite being limited to two-byte constant emission by IE, the authors managed to use aligned `ret` instructions, located at the end of each function, as the part of their gadget. Note that, in their attack, the authors were able to emit *complete* two-byte gadgets in IE. Therefore, this attack will be further limited against the 32-bit version of Chakra, which is a default JIT compiler, even for 64-bit IE. In addition, there are by now known defenses, such as constant blinding, that protect against explicit constants.

7.2 Hidden or Non-Readable Code

In reaction to JIT-ROP, researchers started to propose a great number of defensive schemes that try to hide code or function pointers. In Oxymoron, Backes *et al.* [3] aim to defend against JIT-ROP by hiding code pointers from direct calls. However, the attacker can still find indirect code pointers (e.g., return addresses on the stack or code pointers on the heap), and follow them to read the code. Davi *et al.* [13] thus proposed Isomeron, an improved defense. They keep two versions of the code at the same time, one original and another diversified using fine-grained ASLR. At each function call, they flip a coin to decide which version of the code to execute. This gives a 50% chance of success for each gadget in the chain, making it unlikely to guess correctly for long gadgets.

Gionta *et al.* [15] proposed HideM, which utilizes a split TLB to serve read and execute accesses separately, thus forbidding the attacker to read code pages. Apart from requiring hardware support, HideM also has a limitation that it does not protect function pointers, allowing the attacker to use them in code reuse attacks.

Backes *et al.* [2] and Crane *et al.* [11] proposed two independent defense techniques, XnR and Readactor, respectively, based on the same principle: making executable regions of the memory non-readable. XnR does this in software, marking executable pages non-present and checking the validity of the accesses in a custom page-fault handler. This leaves only a small window of (currently executing) readable code pages, significantly reducing the surface of gadgets an attacker can learn. Readactor uses Extended Page Tables (EPT), hardware-assisted virtualization support for modern CPUs. EPTs allow keeping all executable pages non-readable throughout the entire program execution. In addition, Readactor diversifies the *static* code of the program and hides addresses of the functions by introducing call/jump trampolines, making it impossible to guess the address of any existing code. While being effective against ROP attacks, Readactor left some pointers, such as function addresses in import tables and vtable pointers, intact, thus leaving the programs vulnerable against

function-wise code reuse attacks like return-to-libc [22] or COOP [27]. The fixes to these problems have been proposed by Crane *et al.* in their followup work Readactor++ [12]. We have demonstrated how an adversary can undermine these proposals if code pointer in JIT-compiled code are not hidden. As an orthogonal defense to hiding code pointer in JIT code, we proposed to eliminate implicit constants from JIT-compiled code to preserve XnR's security guarantees.

Pereira *et al.* [26] designed a similar defense technique via a software-only approach for the ARM architecture. They propose Leakage-Resilient Layout Randomization (LR^2), which achieves non-readability of code in ARM by splitting the memory space in data and code pages and instrumenting load instructions to forbid code reading. Furthermore, LR^2 proposed to reduce the size overhead caused by trampolines by using a single trampoline for each callee and encoding the return address with secret per-function keys.

7.3 Defending JIT Against Attacks

Finally, we discuss research that aims to protect JIT compilers against exploitation. In JITDefender, Chen *et al.* [9] remove executable rights from the JIT-compiled code page until it is actually called by the compiler, and remove the rights when it is done executing. In this way they try to limit the time during which attackers can jump to JIT-sprayed shellcode. Although this was effective against some existing JIT-spraying attacks, JITDefender can be tricked by the attacker to keep the needed pages always executable, e.g., by keeping the executed code busy. Wu *et al.* [35] proposed RIM (Removing IMmediate), in which they rewrite instructions containing immediate values such that they cannot be used as a NOP sled. Later, Chen *et al.* [10] proposed to combine RIM and JITDefender, i.e., remove the executable rights from JIT-compiled code pages when not needed and also replace instructions containing immediate values.

In INSeRT, Wei *et al.* [34] propose fine-grained randomizations for JIT-compiled code. Their technique combines (i) removing immediate values via `XOR`ing them with random keys (i.e., constant blinding); (ii) register randomization; and (iii) displacement randomization (e.g., changing the order of parameters and local variables). Furthermore, INSeRT randomly inserts trapping instruction sequences, trying to catch attackers diverting the control flow. Still, its randomization neither affects call/jump displacements, nor would randomization without hiding code actually hinder our approach.

Most related to our attack are the defensive JIT randomization approaches proposed by Homescu *et al.* [17]. They propose librando, a library that uses NOP insertions to randomize the code offsets of JIT-compiled code. We have demonstrated that even browsers leveraging NOP insertion (like IE) are susceptible to our proposed attack and thus proposed a non-probabilistic defense.

8 Conclusion

We have shown that commodity browsers do not protect against code reuse in attacker-generated, JIT-compiled code. Our novel attack challenges the assumption of XnR schemes in that we demonstrate that an attacker can create predictable ROP gadgets *without* the need to read them prior to use. To close this gap, we suggested to extend XnR schemes with our proposed countermeasure that eliminates all critical implicit constants in JIT-compiled code, effectively defending against our attack. Our defense evaluation shows that such practical defenses impose little performance overhead.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. Moreover, we are grateful for the guidance from our shepherd, Ben Livshits, during the process of finalizing the paper. We also want to thank Stefan Nürnberger, Dennis Andriesse, and David Pfaff for their comments during the writing process of the paper.

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and for the BMBF project 13N13250.

References

[1] ATHANASAKIS, M., ATHANASOPOULOS, E., POLYCHRONAKIS, M., PORTOKALIDIS, G., AND IOANNIDIS, S. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium* (February 2015).

[2] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. 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* (New York, NY, USA, 2014), CCS '14, ACM, pp. 1342–1353.

[3] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2014), SEC'14, USENIX Association, pp. 433–447.

[4] BHATKAR, E., DUVARNEY, D. C., AND SEKAR, R. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium* (2003), pp. 105–120.

[5] BHATKAR, S., AND SEKAR, R. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA '08, Springer-Verlag, pp. 1–22.

[6] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM'05, USENIX Association, pp. 17–17.

[7] BLAZAKIS, D. Interpreter Exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–9.

[8] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 559–572.

[9] CHEN, P., FANG, Y., MAO, B., AND XIE, L. JITDefender: A Defense against JIT Spraying Attacks. In *Future Challenges in Security and Privacy for Academia and Industry*, J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, Eds., vol. 354 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2011, pp. 142–153.

[10] CHEN, P., WU, R., AND MAO, B. JITSafe: a Framework against Just-in-time Spraying Attacks. *IET Information Security 7*, 4 (2013), 283–292.

[11] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (Oakland)* (May 2015).

[12] CRANE, S., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., SUTTER, B. D., AND FRANZ, M. It's a TRAP: Table Randomization and Protection against Function Reuse Attacks. In *Proceedings of 22nd ACM Conference on Computer and Communications Security (CCS)* (2015).

[13] DAVI, L., LIEBCHEN, C., SADEGHI, A.-R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *22nd Annual Network & Distributed System Security Symposium (NDSS)* (Feb. 2015).

[14] DAVI, L. V., DMITRIENKO, A., NÜRNBERGER, S., AND SADEGHI, A.-R. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 299–310.

[15] GIONTA, J., ENCK, W., AND NING, P. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2015), CODASPY '15, ACM, pp. 325–336.

[16] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'D My Gadgets Go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 571–585.

[17] HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Librando: Transparent Code Randomization for Just-in-time Compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 993–1004.

[18] HUKU, A. Exploiting VLC. A Case Study on Jemalloc Heap Overflows. http://www.phrack.org/issues/68/13.html.

[19] JACKSON, T., SALAMAT, B., HOMESCU, A., MANIVANNAN, K., WAGNER, G., GAL, A., BRUNTHALER, S., WIMMER, C., AND FRANZ, M. Compiler-Generated Software Diversity. In *Moving Target Defense*, S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Eds., vol. 54 of *Advances in Information Security*. Springer New York, 2011, pp. 77–98.

[20] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22Nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), ACSAC '06, IEEE Computer Society, pp. 339–348.

[21] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 276–291.

[22] NERGAL. The Advanced Return-into-lib(c) Exploits. http://phrack.org/issues/58/4.html.

[23] NOVARK, G., AND BERGER, E. D. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.

[24] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 49–58.

[25] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 601–615.

[26] PEREIRA, O., STANDAERT, F.-X., AND VIVEK, S. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 96–108.

[27] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (Oakland)* (May 2015).

[28] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 25–25.

[29] SHACHAM, H. 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* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.

[30] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 298–307.

[31] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. 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* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 574–588.

[32] TEAM, P. Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[33] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 157–168.

[34] WEI, T., WANG, T., DUAN, L., AND LUO, J. INSeRT: Protect Dynamic Code Generation against Spraying. In *Information Science and Technology (ICIST), 2011 International Conference on* (March 2011), pp. 323–328.

[35] WU, R., CHEN, P., MAO, B., AND XIE, L. RIM: A Method to Defend from JIT Spraying Attack. In *Proceedings of the 2012 Seventh International Conference on Availability, Reliability and Security* (Washington, DC, USA, 2012), ARES '12, IEEE Computer Society, pp. 143–148.