A Call to ARMs: Understanding the Costs and Benefits of JIT Spraying Mitigations

Wilson Lian
UC San Diego
wlian@cs.ucsd.edu

Hovav Shacham
UC San Diego
hovav@cs.ucsd.edu

Stefan Savage
UC San Diego
savage@cs.ucsd.edu

***Abstract*—JIT spraying allows an attacker to subvert a Just-In-Time compiler, introducing instruction sequences useful to the attacker into executable regions of the victim program's address space as a side effect of compiling seemingly innocuous code in a safe language like JavaScript.**

We present new JIT spraying attacks against Google's V8 and Mozilla's SpiderMonkey JavaScript engines on ARM. The V8 attack is the first JIT spraying attack not to rely on instruction decoding ambiguity, and the SpiderMonkey attack uses the first ARM payload that executes unintended instructions derived from intended instruction bytes without resynchronizing to the intended instruction stream. We review the JIT spraying defenses proposed in the literature and their currently-deployed implementations and conclude that the current state of JIT spraying mitigation, which prioritizes low performance overhead, leaves many exploitable attacker options unchecked.

We perform an empirical evaluation of mitigations with low but non-zero overhead in a unified framework and find that full, robust defense implementations of diversification defenses can effectively mitigate JIT spraying attacks in the literature as well as our new attacks with a combined average overhead of 4.56% on x86-64 and 4.88% on ARM32.

I. Introduction

Web browsers are complex programs and continue to exhibit soundness errors in their memory access patterns that form the basis for a broad array of exploits. The combination of the large legacy software footprint and performance overhead concerns have limited the practical effect of proposals to rewrite browsers in memory-safe languages or with tight runtime control flow integrity checks. Instead, most industrial browser developers focus on the use of mitigations that prevent control flow violations from being reliably exploited (*e.g.,* stack cookies, W $\oplus$ X, Address Space Layout Randomization (ASLR), safe memory management functions, pointer encryption, Microsoft EMET, etc.) While none of these are fool proof, taken together they have been highly effective at complicating the exploitation of vulnerabilities.

However, one major loophole remains in the form of "Just-In-Time" (JIT) compilation. All of today's browsers make use of JIT compilation to improve JavaScript performance and thus require data pages be writable and executable (and for reasons we will explain, typically allow pages to be in both states simultaneously for extended periods of time). Thus, by combining this implicit ability to create new executable code (implicitly bypassing W $\oplus$ X) with the heap spraying technique commonly used to bypass ASLR defenses, attackers can still inject new code and blindly redirect to it. While a range of defenses against such "JIT spraying" attacks have ben proposed, modern browsers typically only implement versions of those mitigations that have extremely low overhead (e.g., occasionally blinding large constants).

In this paper we explore the practical import of these choices. Our first contribution is to demonstrate the feasibility of two new JIT spraying attacks affecting web browsers on Android phones, the first of which provides practical code injection against the Chrome V8 JIT on ARM and the second of which provides Turing-complete malicious computation against Mozilla's SpiderMonkey JIT on ARM. Thus, taken together with recent work by Lian et al. [17] showing JIT spraying vulnerabilities in Webkit (i.e., Apple phones), all major smartphone browsers (almost 2 billion computers) are vulnerable to this style of attack. Our second contribution is a collection of open source implementations of existing proposed JIT spraying mitigations for SpiderMonkey on both ARM32 and x86-64 and empirical evaluations of their performance overhead on a consistent testing platform. We find that enabling constant blinding—which incurs the highest overhead of any single mitigation that we implemented—can reduce the probability of landing a JIT spray exploit by a factor of 2.41×10^{462} with an overhead of just 1.39% and 3.99% on x86-64 and ARM32, respectively. We argue that the value of mitigation justifies the small performance penalty and that JIT developers should implement register randomization, constant blinding, call frame randomization, random NOP insertion, and base offset randomization (with combined average runtime overheads of 4.56% and 4.88% on x86-64 and ARM32, respectively) to close this remaining code reuse loophole.

II. Background

When a JIT compiler compiles code in a high level language into native instructions, the opcodes and operands it emits are heavily influenced by the potentially-untrusted high level code. Furthermore, the high level code can create new native code at-will by dynamically creating and evaluating new code. This grants the untrusted party who wrote the high level code unprecedented influence over large swaths of executable memory in the language runtime's address space. Blazakis [7]

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23108

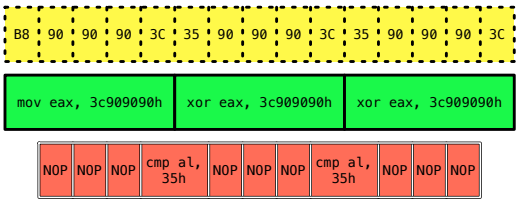


Fig. 1: Illustration of a NOP sled encoded in the bytes implementing the statement `x = 0x3c909090 ^0x3c909090 ^0x3c909090;`

was the first to publicize an attack which leveraged these properties of a JIT compiler to perform code injection on x86. In the attack, the adversary encodes a NOP sled and shellcode in a seemingly-innocuous sequence of bitwise XOR operations resembling the following:

```
x = 0x3c909090 ^ 0x3c909090 ^ 0x3c909090;
```

When compiling the above, the ActionScript JIT compiler produces the bytes shown in dashed boxes in Figure 1, which encode the x86 instructions shown in the solid-bordered boxes below them. However, since x86 instructions have variable lengths and can be decoded at any byte alignment, an alternate decoding of the bytes can be observed by disassembling from any unintended instruction boundary, as shown in the bottom row of Figure 1 in double-lined boxes. This alternate decoding functions as a NOP sled which lands at 4 out of 5 byte offsets and can be extended without resynchronizing to the intended instruction stream as long as the opcode bytes for XOR (the `0x35` bytes) continue to be consumed as instruction operands.

In an actual attack, the NOP sled would be lengthened by extending the chain of XORed `0x3c909090` constants, and eventually the `0x90` bytes would be replaced with the encodings for shellcode instructions, with the limitation that each shellcode instruction fit into three consecutive `0x90` byte slots. The attacker would place the XOR chain statement in a function then repeatedly declare and invoke it in order to cause the JIT compiler to fill as many pages as possible of executable memory with the hidden NOP sled and shellcode. By spraying NOP sleds that are much larger than the shellcode, execution beginning at a random address in sprayed code has nearly an 80% chance of successfully executing the shellcode.

More generally, the opportunities for exploitation introduced by JIT compilers are code reuse (e.g., the attack described above) and code corruption, wherein the attacker abuses the fact that JIT code memory must, at some point in time (or for many JITs, at all times), be writable. Writable code memory—once thought to be a relic of the bygone pre-W $\oplus$ X era—is necessary not only for the JIT to create and delete code, but also for the frequent patching many JIT implementations undertake to support inline caching, a performance optimization employed by nearly all JavaScript JIT compilers to ease the burden of the dynamic type system.

Since Blazakis first brought JIT spraying into the public eye, it has been extended to JavaScriptCore's non-optimizing JIT for x86 [26], Mozilla's JaegerMonkey and TraceMonkey JITs [25] for x86, the Tamarin ActionScript JIT for ARM [6], and JavaScriptCore's optimizing JIT for ARM's Thumb-2 instruction set [17]. JIT compilers have also been abused to construct ROP gadgets that are exploited with the aid of a memory disclosure vulnerability [4], [18]. With the exception of [18], all past incarnations of JIT spraying have relied on constant values supplied by the attacker in high level code; and without exception, all past JIT spraying attacks have made use of malicious instructions encoded at instruction boundaries not intended by the JIT compiler. These attacks would be thwarted by a robust implementation of constant blinding (cf. §IV). In §III-B, we will introduce a new JIT spraying attack against V8 on ARM which is the first JIT spraying attack that does not rely on the execution of JIT code at unintended instruction boundaries and, like [18], does not abuse the translation of untrusted constants into instruction operands in JIT code.

III. Novel JIT spraying threats against ARM

In this section, we present two novel JIT spraying threats against ARM. The first (§III-B) is a proof of concept end-to-end attack against Chrome's V8 JavaScript engine that applies Lian et al.'s gadget chaining technique [17]. Its novelty rests in the distinction that, unlike all prior JIT spraying attacks, this one does not rely on the improper disassembly of JIT code. Afterwards, we describe a method for encoding a "self-sustaining" JIT spraying payload which, in contrast to prior JIT spraying payloads for ARM, can execute an arbitrary number of malicious instructions without resynchronizing to the intended instruction stream (§III-C). We used this method to implement a proof of concept payload against Mozilla's SpiderMonkey JavaScript engine which interprets instructions for a One Instruction Set Computer.

A. Instruction sets on ARM

The discussion of our new threats hinge on an understanding of certain low-level details of the ARM architecture. Therefore, we briefly introduce relevant aspects of the instruction sets supported by 32-bit ARM chips[1]. Recent ARM chip designs (ARMv6T2 and later) have mandatory support for at least two instruction sets. These are the original "ARM" instruction set, and the newer "Thumb-2" instruction set. The ARM instruction set is composed of fixed-width 32-bit instructions stored in memory as 32-bit-aligned words. The Thumb-2 instruction set, on the other hand, is designed for improved code density and contains both 16-bit and 32-bit wide instructions stored in memory as a single 16-bit aligned halfword and two consecutive 16-bit aligned halfwords, respectively.

At any given time, a 32-bit ARM core is said to be executing in either "ARM mode" or "Thumb mode." Interworking between ARM code and Thumb-2 code is possible through the use of unprivileged interworking branch instructions, which either unconditionally toggle the processor between modes or derive the desired execution mode of the branch target from the least significant bit of the branch target address. Since both ARM and Thumb-2 instructions are aligned to at least 16-bit boundaries, the least significant bit of any valid instruction address is unused. Interworking branch instructions take advantage of this by repurposing the bit. If the bit is set, the target is executed in Thumb mode; otherwise it is executed in ARM mode.

[1] Support for a 64-bit instruction set called A64 was introduced in ARMv8-A. However, since support for A64 is not mandatory in ARMv8-A, and many 32-bit ARM chips remain on the market, we do not consider omission of A64 a significant limitation.

B. JIT spraying on ARM without improper disassembly

In this subsection, we describe a new proof of concept attack against Chrome's V8 JavaScript engine on ARM which demonstrates—for the first time against any architecture—the feasibility of carrying out a JIT spraying attack that uses JIT-emitted instruction bytes without exploiting ambiguity in the decoding of those instructions. In fact, this attack relies on neither untrusted constants appearing in JIT code as immediate operands nor execution of JIT code at unintended instruction boundaries. Since V8's JIT compiler emits fixed-width 32-bit ARM instructions, the latter non-dependency is trivial, provided that the JIT spraying payload is executed in ARM mode.

The V8 attack uses the *gadget chaining* technique introduced by Lian et al.[17]; gadget chaining is a technique in which an attacker's high level language (HLL) code (e.g., JavaScript) is able to treat unsafe computation performed by reused code as though it were a subroutine. The attacker's HLL code invokes a control flow vulnerability to branch to a reused code snippet, which performs unsafe computation then returns control flow back to the HLL code. Each reused snippet is referred to as a "gadget," and each gadget may or many not take arguments or return values to the HLL code. The use of gadget chaining gadgets differs from ROP gadgets, however, in that control flow after a gadget chaining gadget returns does not continue directly to another gadget, but rather back to the language runtime where the HLL code resumes execution.

The high level structure of the proof of concept attack is as follows. After JIT spraying a particular store instruction (the store gadget) into memory, the attacker clears the victim's i-cache of the sprayed store gadgets by calling numerous DOM functions. She then guesses the address of a store gadget and uses a hijacked virtual host function call[2] to simultaneously branch to that address and control the contents of the input registers used by the store gadget. The first invocation of the store gadget writes a return instruction (`bx lr`) into JIT code a short distance after the store instruction. Subsequent invocations are made in order to write 4 bytes at a time of shellcode into the memory following the injected return instruction. The victim's i-cache is cleared once more, and a final invocation of the store gadget overwrites the injected return instruction with a NOP instruction and the execution of the shellcode. The details of gadget layout and creation, the artificial control flow vulnerability, and failure-tolerant gadget invocation are described below.

1) Gadget layout and creation: The sprayed store gadget consists of an intended store instruction used to spill a live value onto the stack followed by at least one i-cache line (128 bytes on our test machine) of padding instructions that perform bitwise operations over caller-saved registers. The injected return instruction will be written after the one i-cache line padding so that it will be executed during the same gadget invocation that it is injected. The padding instructions must not access memory because they are likely to cause a segmentation fault or clobber critical machine state. They must also operate only on caller-saved registers because they will execute as intended and must preserve the values of callee-saved registers for when the gadget returns.

[2]This was a vulnerability which we artificially injected into V8.

It is necessary to inject a return instruction rather than allowing control flow fall through into the enclosing function's epilogue and return instruction because the epilogue performs stack cleanup and loads the return address from the stack, both of which would prevent a proper return to high level language (HLL) code given our decision to use an injected control flow vulnerability in the form of a hijacked virtual host function call. When control flow arrives at the gadget under those circumstances, the stack is not setup properly for a JIT function epilogue to clean it up, and the return address resides in the link register (`LR`) rather than on the stack. However, once the store instruction has written its return instruction during its first invocation, the gadget is a reusable primitive that can be called repeatedly to overwrite arbitrary words in memory.

The sprayed store instruction is `str r2, [r11, #-20]`, where `r11` is used as a frame pointer register in V8's JIT code. The JavaScript function whose JIT compilation results in the emission of a spray gadget defines numerous variables which are used in the computation of the return value. By defining more such variables than there are allocatable registers, V8's optimizing JIT will begin spilling values onto the stack. The sprayed store instruction is one such spilling instruction. We `eval()` the definition and repeated invocation of the sprayed function to trigger optimized compilation and the spraying of a store gadget. Optimized compilation is necessary because only the optimized compiler allocates and spills registers, which are necessary to create the store instruction and the subsequent memory-access-free padding instructions.

2) Artificial control flow vulnerability: For our proof of concept attack, we simulated a memory corruption vulnerability that could be used to hijack the virtual function table pointer of a DOM object. We added a JavaScript host function `hijackVTable` into V8 that performs the desired corruption. Hijacked virtual functions are especially useful for a gadget chaining because they can serve two purposes: subverting control flow and controlling the gadget's operands, which in the case of the store gadget are two registers. We make use of the DOM's `Blob` class and its `slice()` method, which is implemented as a C++ virtual function and accepts two `long`s as arguments that can be controlled by a JavaScript caller. We were fortunate that both arguments eventually reside in the registers used by the store gadget (`R2` and `R11`), despite the fact that one of the `long` arguments is actually passed on the stack. This occurs because the various trampolines executed to shuffle values between the JavaScript calling convention and the architecture ABI calling convention happen to leave a copy of the stack-passed argument in `R11`.

3) Failure-tolerant invocation: In order to use the store gadget, our control flow vulnerability must be able to precisely target the gadget's store instruction; if execution begins before the store instruction, the intended instructions before it could clobber the source register operand. If execution begins after it, the new return sequence cannot be patched in during the gadget's first invocation, most likely leading to a crash. To solve this problem, we place the gadget at a (semi-)predictable offset within each coarse-grained memory allocation chunk.

V8's code memory allocator maps a new 1MB chunk of RWX memory if it can't fulfill an allocation request from the current pools of free JIT code memory. Allocation requests are then satisfied starting at the low-addressed end of the

new chunk. If we could coerce V8 into placing a copy of the optimized function containing the store gadget as the first unit of code compilation in each fresh 1MB code chunk, we would need to guess only which 1MB chunk contains a sprayed gadget (i.e., the most significant 12 bits of a 32-bit address).

Unfortunately, due to the nature of V8's JIT compilation pipeline, it is not possible to guarantee that the store gadget will be the first unit of code compilation placed in each 1MB chunk. During a single function instance's lifetime from declaration to optimized compilation, V8 produces four different pieces of code which contend for the coveted first slot. They are the anonymous function that declares the function being sprayed, the unoptimized JIT code for the function being sprayed, a second copy of the unoptimized JIT code (which is produced once more after V8 decides to compile the function with the optimized compiler), and the optimized JIT code for the function being sprayed.

For reasons which will become apparent, it is essential that these four pieces of code are emitted in that exact order, with no interleaving between parts of consecutively-sprayed instances of the function. Our spraying procedure ensures this by invoking each instance of the sprayed function in a loop a sufficiently-large number of times in order to cause V8 to consider the function "hot" and optimize it. The number of loop iterations was tuned to be large enough that the invocation loop for a particular instance of the function would still be running when the optimized code (which is compiled asynchronously) is finally emitted.

If the first piece of code in each 1MB chunk were chosen uniformly at random from the four possibilities, 25% of the time it would be the anonymous declaration function, over whose size and contents we exert very little control. However, due to the various space requirements of the different pieces of code—384 bytes for the declaration, 2912 bytes for each copy of the unoptimized code, and 672 bytes for the optimized code—a new 1MB chunk is most likely to be allocated for the large unoptimized spray code. Indeed, measurements of V8 embedded in Chrome show that the probabilities that the first copy of optimized spray code in a 1MB chunk will be preceded by 0, 1, and 2 copies of the unoptimized spray function are 0.391%, 49.2%, and 48.4%, respectively; and the probability that the anonymous declaration function will take the first slot is only 1.17%.

Although the optimized spray function is not likely to be sprayed at any single location near the beginning of all 1MB chunks, in over 98% of chunks, the only code preceding it in a chunk are unoptimized spray functions, whose size and contents we control. We take advantage of this fact and craft the spray function in such a way that an intended return instruction is emitted at the same offset (Δ) from the beginning of the function in unoptimized code as the store gadget in optimized code. This makes it safe to accidentally branch into an unoptimized spray function with a hijacked function call since execution will immediately return rather than crashing. Figure 2 illustrates how we accomplished this by placing a conditional return early in the sprayed function to take advantage of the fact that V8's unoptimized JIT code is less dense than its corresponding optimized code.

With the spray function's unoptimized and optimized code laid out as described, there is >98% probability that the store

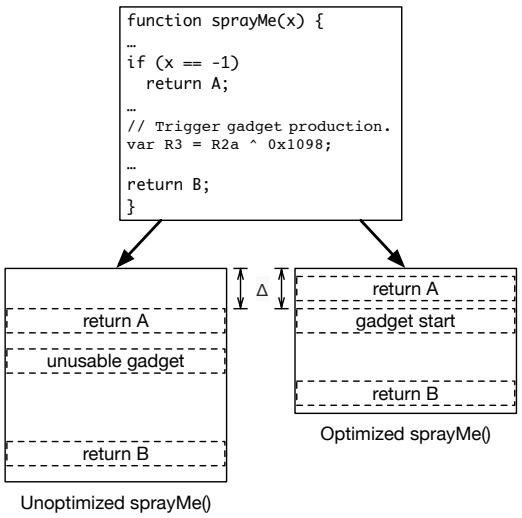


Fig. 2: Illustration of how a return instruction in unoptimized JIT code is aligned to the same function offset Δ as a gadget in optimized JIT code.

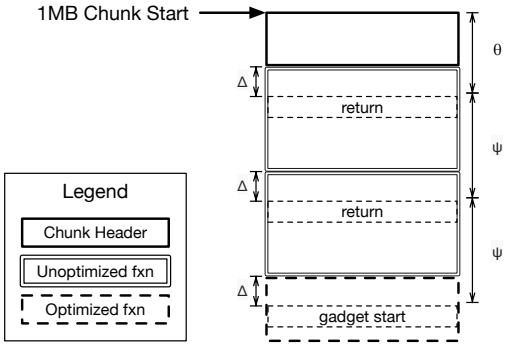


Fig. 3: Illustration of the beginning of a 1MB chunk that can be probed for the location of a gadget in a failure-tolerant manner. An incorrect guess of θ or $\theta + \psi$ will only execute a harmless return instruction. Δ is the common offset of both the return instruction and the gadget in both the unoptimized and optimized code.

gadget will reside at one of the following offsets in a given 1MB chunk: θ, $\theta + \psi$, or $\theta + 2\psi$. The values of both θ and ψ are deterministic and known. The value of θ is the size of the fixed-sized header at the start of each 1MB chunk plus Δ; and ψ is the size of the unoptimized spray function. Figure 3 illustrates an example memory layout at the beginning of a 1MB allocation chunk in which two copies of the sprayed function produced by the non-optimizing JIT precede a copy of the sprayed function produced by the optimizing JIT. The optimized copy contains the store gadget, which resides at the offset $\theta + 2\psi$ from the start of the chunk. Observe that if there were zero or one copies of the unoptimized function code before the optimized copy, the gadget's offset from the chunk's start would be θ and $\theta + \psi$, respectively.

This meticulously-crafted memory layout enables us to probe for the gadget's address in a failure-tolerant manner. The first time the attacker triggers the control flow vulnerability, she guesses a 1MB chunk and targets the common offset in the first function in the chunk (θ). In the unlikely event that the first

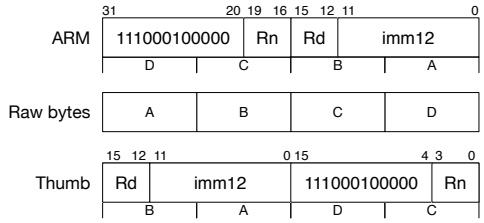


Fig. 4: Illustration of how the immediate-operand bitwise AND instruction from the ARM instruction set can be decoded as two 16-bit Thumb-2 instructions.

function in the 1MB chunk is a declaration (1.17% probability, assuming spray code is monopolizing JIT code memory), the attack will fail. However, with high probability, it will be a copy of the sprayed function's optimized or unoptimized code. In those cases, either the gadget or a return instruction will execute. If it is the former, the attack succeeds; otherwise, the hijacked virtual function call will immediately return. Eventually, the attacker's script will expect an invocation of the store gadget to result in shellcode execution, and when that fails to occur, it can be concluded that the control flow vulnerability was targeting a return instruction rather than a store gadget. The script can then increase the target address of the control flow vulnerability by the size of the unoptimized spray function (ψ) and try again.

V8 limits the amount of JIT code memory at 256MB. If the attacker is able to monopolize these 256MB, her odds of success depend mostly on her ability to guess which 1MB chunks contain JIT code. On a 32-bit system, a conservative estimate is 6.125% ($256/4096 \times 0.98$); however, a more realistic estimate might take into account that the location of JIT code regions can be narrowed down to half of the available address space, giving a probability of 12.25%.

C. Constructing a self-sustaining ARM JIT spraying payload

Prior JIT spraying attacks against ARM failed to repurpose JIT code to form a "self-sustaining" JIT spraying payload that executes in its entirety once execution branches to it.[3] Instead, Lian et al. [17] introduced the gadget chaining technique, which we borrowed in the attack we presented in §III-B. Lian et al. studied JavaScriptCore, which emits Thumb-2 instructions, and considered the viability of self-sustaining payloads that encode an unintended instruction stream—be it a stream of ARM or Thumb-2 instructions—using intended Thumb-2 instructions; they found such payloads to be infeasible.

In this section, we describe a method for encoding an unintended Thumb-2 instruction stream using ARM instructions emitted by the Mozilla SpiderMonkey JavaScript engine's optimizing JIT compiler IonMonkey. Our technique can be used by an attacker to achieve arbitrary Turing-complete computation while executing only instructions decoded from intended instruction bytes. Whereas the proof of concept JIT spraying attacks against ARM described in [17] and §III-B use the JIT spraying payload to launch a JIT code corruption attack, the technique described in this section is effective even in the presence of non-writable JIT code.

[3]Beck [6] demonstrated a spraying technique against the Tamarin ActionScript JIT on ARM that would enable the encoding of a self-sustaining payload, but it leverages constant pools—data values inlined with JIT code—rather than maliciously-repurposed instructions.

```
function sbnz(a, b, c, d)
  Mem[c] = Mem[b] - Mem[a]
  if (Mem[c] != 0)
    // branch to instruction at address d
  else
    // fallthrough to next instruction
```

Listing 1: Pseudocode for the `sbnz` instruction.

We implemented a proof of concept self-sustaining payload which executes an interpreter loop for a One Instruction Set Computer (OISC) [19], an abstract universal machine that has only one instruction. There are many options for the single instruction; we implemented *Subtract and Branch if Non-Zero (SBNZ)*. Listing 1 shows the pseudocode for the `sbnz` instruction. In the remainder of this subsection, we describe in detail our technique for constructing a self-sustaining JIT spraying payload capable of Turing-complete computation, a NOP-sled construction method, the means by which a self-sustaining payload may be invoked, and the limitations of the technique and our proof of concept payload.

1) Payload-building technique: We first describe our method for encoding an unintended Thumb-2 instruction stream among the intended ARM instructions emitted by Mozilla SpiderMonkey's optimizing JIT compiler IonMonkey and orchestrating the proper flow of control through the unintended instructions. Our technique makes use of the bitwise AND instruction, which computes the bitwise AND of a 12-bit immediate value (`imm12`) and the contents of a register (`Rn`) and stores the result into an arbitrary register (`Rd`). By carefully structuring our JavaScript code, we are able to control both 4-bit register operands and the 12-bit immediate for a total of 20 out of 32 bits. The bytes in the encoding of ARM's immediate-operand bitwise AND instruction form two consecutive 16-bit Thumb instructions, as shown in Figure 4. From top to bottom, the rows show the layout of the ARM AND instruction, the in-memory layout of those bytes, and the layout of those same bytes when decoded as Thumb-2 instructions.

The observant reader may be curious as to why the unintended Thumb-2 instruction stream will decode to 16-bit instructions rather than 32-bit Thumb-2 instructions. The reason is that 32-bit Thumb-2 instructions must begin with the bit prefix 11101_2 or 1111_2, but IonMonkey only allocates live values to registers in the range 0000_2–1011_2, inclusive. Neither byte B nor byte D in this particular instruction can contain this prefix, and therefore Thumb-mode decoding beginning at either halfword can only yield 16-bit Thumb-2 instructions. Why not choose an ARM instruction whose byte B or byte D can include 32-bit Thumb-2 instruction prefixes? The reason is that it is difficult for an adversary to coerce the JIT compiler into producing such instructions; and very few bits within those instructions are easily influenced by the attacker.

In addition to the constraints on the `Rd` register, the first Thumb-2 instruction is also constrained by the set of valid 12-bit immediate operands to the ARM AND instruction. The 12-bit immediate is meant to be interpreted as an 8-bit value with a 4-bit rotation field, but valid encodings must use the smallest possible rotation value. Therefore, it is impossible to induce the JIT compiler into emitting certain bit patterns in the `imm12` field. Taking these constraints into account, the halfword formed by bytes C and D can still encode a broad range of 16-bit Thumb-2 instructions.

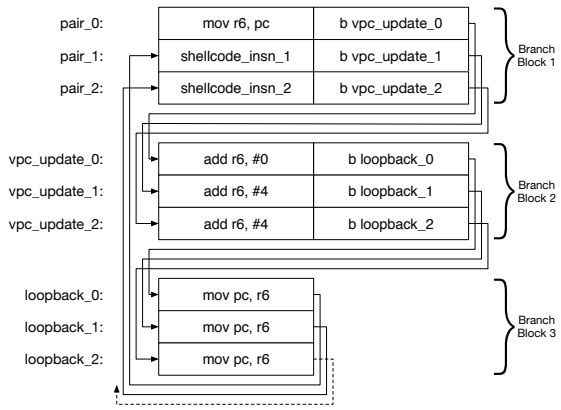


Fig. 5: Illustration of using a virtual PC (in this case `R6`) to more efficiently utilize the space skipped over by branches.

The second Thumb-2 instruction must be an unconditional PC-relative forward branch of at least 512 halfwords. The `Rn` field forms the least-significant 4 bits of the branch distance in units of halfwords. The self-sustaining payload works by chaining together pairs of unintended 16-bit Thumb-2 instructions with these unconditional branches. The first Thumb-2 instruction performs useful work for the adversary; the second branches to the first Thumb-2 instruction in a subsequent pair. For this branch to target the first instruction in a pair, the branch offset must be an odd number of halfwords, so `Rn` must be an odd-numbered register. The value of the PC in Thumb mode is the address of the current instruction plus 4 (i.e., 2 halfwords). Consequently, the closest we can place the next pair of unintended Thumb-2 instructions is $(512+1+2)\times 2 = 1030$ bytes after the start of the unintended branch instruction.

Naïvely chaining 1030-byte forward branches would require an exorbitant amount of memory to encode even a simple payload. To reduce the space requirements of our self-sustaining payload, we designate a general purpose register as a virtual PC which we use to loop execution back into the branched-over space, where another unintended instruction pair has been placed. We define a *branch block* as the largest block of unintended instruction pairs whose first unintended instruction pair skips over all subsequent unintended instructions pairs in that branch block. Figure 5 shows the virtual PC method with 3 branch blocks under simplified conditions. Note how execution flows through each unintended instruction pair in each branch block (with the exception of branch block 3, which only executes the first unintended instruction in the pair), then through the second instruction pair in each branch block, etc. In our proof of concept payload, 12-bit immediate encoding rules require us to populate a register with the virtual PC advancement amount and perform register-register addition rather than register-immediate addition. Furthermore, in order to prevent dead store elimination, the JavaScript statements that produce unintended instruction pairs reside in separate mutually-exclusive conditional blocks, resulting in a larger virtual PC advancement amount of 36 bytes (Listing 2). Note that although Figure 5 shows only three branch blocks, longer payloads can be encoded by inserting branch blocks before the "vpc_update" block. Another option is to increase the size of branch blocks by using an intended instruction other than bitwise AND; for example, bitwise XOR and OR would result in branch blocks that are 64 and 768 bytes longer, respectively.

TABLE I: Table of instructions implementing the SBNZ OISC abstract machine as a self-sustaining payload. Horizontal rules indicate branch block boundaries where padding is inserted.

#	Label	Unintended Thumb instruction	Intended ARM instruction
1	vpc_init	add r6, pc, #36	and r10, r1, #9437184
2		add, r7, #1	and r3, r1, #262144
3	oisc_pc_init	mov r5, sp	and r4, r1, #114294784
4	interpreter_loop_top	ldr r1, [r5, #0]	and r6, r1, #2686976
5		ldr r2, [r5, #4]	and r6, r1, #6946816
6		ldr r3, [r5, #8]	and r6, r1, #11206656
7		ldr r1, [r1, #0]	and r6, r1, #589824
8		ldr r2, [r2, #0]	and r6, r1, #1179648
9		sub r2, r2, r1	and r1, r1, #335872
10		str r2, [r3, #0]	and r6, r1, #26
11		cbz r2, #104 (zero)	and r11, r1, #-2013265918
12	non_zero	ldr r5, [r5, #12]	and r6, r1, #15532032
13		subs r6, #162	and r3, r1, #2592
14	zero	adds r5, r5, #13	and r3, r1, #54525952
15		adds r5, r5, #3	and r3, r1, #12582912
16		subs r6, #215	and r3, r1, #3440
1	incr_init	movs r7, #35	and r2, r1, #9175040
2–12	vpc_advance ($\times$ 11)	adds r6, r7	and r4, r1, #1040187392
13	non_zero_loopback	subs r6, #162	and r3, r1, #2592
14–15	vpc_advance ($\times$ 2)	adds r6, r7	and r4, r1, #1040187392
16	zero_loopback	subs r6, #217	and r3, r1, #3472
1–16	branch_vpc ($\times$ 16)	mov pc, r6	and r4, r1, #191889408

```
function sprayMe(r0, R10, FP, r8, R7, R5) {
  // Statements to define additional variables and
  // populate their values into registers go here
  if (R10 == 0) {
    R10 = R1 & 9437184;  // and r10, r1, #9437184
  } else if (R10 == 1) {
    R3 = R1 & 262144;    // and r3, r1, #262144
  } else if (R10 == 2) {
  ...
  }
  // Return statement using all variables goes here
}
```

Listing 2: The structure of the JavaScript function sprayed to produce the self-sustaining SBNZ OISC payload.

Using the virtual PC method, we implemented the interpreter loop for an SBNZ OISC abstract machine. We designate a general-purpose register as the OISC PC and use unintended instructions to perform the subtraction and update the OISC PC according to its outcome. Our implementation expects the first instruction, composed of four consecutive 32-bit addresses corresponding to the four instruction operands, to reside at the top of the stack when it begins executing. The instructions used to build the interpreter are shown in Table I. Note that instructions 1 and 2 within each branch block are 40 bytes apart, whereas all other inter-instruction spacing within branch blocks is 36 bytes. When tracing control flow through the table, remember that instructions sharing a common number in the first column will be executed consecutively with one another with the exception of instruction 11 in the first branch block (`cbz`), which may branch to the `zero` label. This proof of concept demonstrates the feasibility of performing Turing-complete computation with unintended Thumb-2 instructions decoded from intended ARM instruction bytes that are executed as a self-sustaining payload.

2) Encoding a NOP sled: It is possible to construct a NOP sled by placing $n+k$ branch blocks prior to the branch blocks containing shellcode. The unintended instruction pairs

in the initial n branch blocks exist only to direct control flow forward to the final k branch blocks. The unintended instructions in the final k branch blocks of the NOP sled use their statically-predetermined offset within the branch blocks to construct a branch to the first unintended instruction in the first shellcode branch block. For example, the first unintended instruction in the final NOP sled branch block must effect a large forward branch to skip the entire branch block, but the last unintended instruction in the same branch block need only skip any remaining tail in its own branch block and whatever short head exists at the beginning of the next branch block.

The success rate of correctly landing in the NOP sled depends on how densely unintended instruction pairs can be packed in the final k branch blocks. The only way to both achieve a high density of unintended instruction pairs and avoid dead store elimination is to use the the destination register of the intended AND instruction as one of the input operands. For example, `and r1, r1, #10; and r1, r1, #10` is okay, but `and r4, r1, #10; and r4, r1, #10` is not because the first instruction can—and will—be eliminated. Recall that the register must also be odd-numbered in order for the unintended branch instruction in each pair to correctly target the beginning of the next unintended instruction pair. We were unable to devise a NOP sled whose unintended instructions are derived from only intended instructions operating on odd-numbered registers. We must therefore sacrifice density in the same manner as described in §III-C1 and place unintended instruction pairs in the NOP sled 36 bytes (9 ARM instructions) apart. The success rate of correctly landing in such a NOP sled is therefore $1/9$.

3) Executing the payload: Static code for the ARM architecture is compiled with the expectation that callees and callers might need to be executed in a different instruction set mode; therefore, interworking branches abound. An attacker will most likely only need to ensure that the control flow vulnerability she exploits targets an address whose least significant bit is set, which will cause her payload to be executed in Thumb mode.

4) Limitations: A major limitation of the payload encoding method described in this section is that we are unable to encode an unintended system call instruction. In order to do so, we would need to be able to control an intended instruction whose destination register (or source register in the case of a store instruction) is the stack pointer; we are not aware of such a capability against IonMonkey. However, the Turing-complete computation that this type of payload is able to construct can be used to orchestrate a code reuse attack against static code which contains system call instructions.

Our proof of concept SBNZ OISC implementation requires the operands to the `sbnz` instruction to contain absolute addresses. This requires the attacker either to learn where her SBNZ instructions will reside via an information leak or to heap spray them. Unfortunately, heap spraying SNBZ instructions competes with JIT spraying. A more practical SBNZ OISC implementation might use stack pointer offsets rather than absolute addresses for `sbnz` operands. The attacker could even devise an SBNZ NOP sled to place on the stack before her SBNZ shellcode to mitigate an unpredictable stack layout.

As we mentioned above, the set of unintended Thumb-2 instructions that we were able to encode was constrained by the set of registers that could be chosen as the destination register in the intended ARM instruction. We were fortunate that IonMonkey's allocates live values to `R11` because it enabled us to encode the "Compare and Branch on Zero" (`cbz`) instruction, which is crucial for Turing-completeness. V8, on the other hand, does not allocate live values to register `R11`, significantly complicating the encoding of a Turing-complete payload. However, even if it were not possible to encode Turing-complete computation in the self-sustaining payload, the technique allows for the construction of more sophisticated gadgets for use in gadget chaining.

IV. JIT SPRAYING MITIGATIONS

Researchers and practitioners have proposed numerous mitigations against the security risks introduced by JIT compilers. The mitigations vary widely in peformance overhead, difficulty of integration into an existing JIT compiler, and defensive effectiveness. We organize these efforts into the following three categories: capability confinement, memory protection, and diversification. In order to place our own defense implementations and the performance evaluations thereof (§V) into context, we provide an overview of JIT spraying mitigation proposals in these three classes and their adaptations (when applicable) in real world JIT implementations.

A. Capability confinement

The objective of capability confinement defenses is to make JIT code an unattractive reuse target in general by reducing the set of capabilities that JIT code can possess. More sophisticated capability confinement defenses also prevent JIT code from corrupting itself. The least sophisticated capability confinement defenses employ simple heuristics to detect system calls from JIT code [11] or the emission of long sequences of consecutive instructions taking 32-bit immediate operands, where those sequences begin with `mov reg, imm32` [5], which was the hallmark of Blazakis' original attack. Though the performance overhead of [11] is not prohibitive at 1.84% on the SPEC CPU2000 Integer benchmark ([5] does not report on its performance overhead), the two heuristic defenses can be easily circumvented by reusing statically-compiled system calls and avoiding the creation of the initial `mov` instruction,[4] respectively. The next two capability confinement defenses offer much stronger security benefits through sandboxing.

1) NaCl-JIT: NaCl-JIT [3] is a system that extends the Native Client (NaCl) sandbox [34] so that a language runtime running in the sandbox can dynamically install, invoke, modify, and delete code within the sandbox on the fly. The sandbox provides the following three high level guarantees about sandboxed code execution: (1) it cannot read or write outside of a contiguous region of data memory; (2) it contains only instructions drawn from a whitelist, and they reside in a contiguous region of sandboxed code memory; and (3) aside from API calls into the NaCl runtime, its control flow only executes instructions decoded at intended instruction boundaries within the aforementioned code memory region. Thus, even a malicious JIT-compiled program that is able to completely commandeer the sandboxed language runtime and

[4]Rather than writing high-level code of the form `var x = (imm32^...^imm32);`, one can write `var x = (y^...^imm32);` where `y` is not defined as a constant, which will not require moving a constant into a register.

issue arbitrary NaCl-JIT API calls still cannot access memory outside of the sandbox or directly execute non-whitelisted instructions such as system calls or cache flushes. While this may allow a malicious JIT-compiled program to access sandbox memory in ways that the sandboxed language runtime did not intend for it to access, the underlying system outside the sandbox remains safe.

The NaCl sandbox is implemented via code verification, instruction bundling, and (for most architectures) software guards, which incur a high overhead. The runtime overhead of the NaCl-JIT port of the V8 JavaScript engine on the V8 JavaScript benchmark suite ranges from 28%-60% on x86-32 and x86-64. While NaCl-JIT's security properties are indeed alluring, performance regressions on this order are unlikely to be considered acceptable for a mainstream JavaScript engine actively engaged in the benchmark wars.

2) RockJIT: RockJIT [23] offers a platform similar to NaCl-JIT with considerable performance improvements due to more efficient JIT code validation and a more efficient mechanism for protecting control flow (fine-grained control flow integrity (CFI) [2] (without a shadow stack) in the language runtime and coarse-grained CFI on JIT code). RockJIT's authors ported the V8 JavaScript engine to use RockJIT; and on the same set of benchmarks over which NaCl-JIT showed a 51% overhead, RockJIT had just 9.0% overhead. Over the entire Octane 2 JavaScript benchmark suite, RockJIT incurred a 14.6% average overhead.

However, these performance gains come with security risks. Notably, RockJIT allows the (untrusted) language runtime to directly make system calls (excluding memory reprotection), and it trusts the language runtime to announce indirect branch targets within JIT code so that RockJIT can update the CFI policy. A control flow bending [8] attack would enable an attacker to abuse the lack of a shadow stack and issue any system call present in the language runtime.

B. Memory protection

A vulnerability commonly introduced by JavaScript JIT compilers is permanent RWX permissions on JIT code pages, even on systems that enforce W $\oplus$ X on normal code and data pages. This practice is done in the name of performance; inline caching (IC) [12]—a performance optimization popular [30], [20], [1] among JavaScript implementations—often requires the frequent compilation of short code stubs during JIT code execution as well as the patching of calls to those stubs into existing code. Leaving JIT code pages RWX at all times eliminates memory reprotection system calls during inline cache installation and maintenance as well as those that would be necessary during the compilation of normal program code.

RWX memory poses a significant threat to JIT security. Lian et al.'s attack against the JavaScriptCore JIT on ARM [17] as well as the new attack we introduced in § III-B rely on always-RWX JIT code pages in order to perform self-modification of JIT code. Memory protection defenses seek to rid JIT code of its always-RWX memory protection status by adapting W $\oplus$ X to dynamically-generated code. Two popular approaches to memory protection for JIT code are transient protection and dual mapping.

1) Transient protection: A recent addition to the SpiderMonkey JavaScript engine [21] is W $\oplus$ X protection for JIT code with the aim of preventing corruption of JIT code. When the defense is enabled, JIT code is RX by default, and when the runtime needs to modify a unit of code compilation, that unit is temporarily reprotected RW. The performance overhead of this change is less than 1% on the Kraken and Octane JavaScript benchmark suites and no more than 4% overhead on the SunSpider benchmark suite. SpiderMonkey's implementation of inline caches inherently favors low-overhead W $\oplus$ X protection, as it requires far less-frequent patching (and therefore fewer memory reprotection system calls) and stub compilation than other compilers such as V8. Reducing the need for patching in V8 is a non-trivial matter. In fact, the V8 team has explained to us that although they are undergoing efforts to eliminate patching for some types of ICs, others are still considered too performance-sensitive. Unfortunately, this means that transient W $\oplus$ X protection will likely remain too expensive for V8.

JITScope [35] is a proposed defense that provides normally-RX protection similar to SpiderMonkey's W $\oplus$ X defense, but it reprotects memory more frequently and therefore incurs a higher overhead (1.6%-6.0%). Although these defenses offer protection against code corruption in single-threaded environments, Song et al. [28] demonstrated that Web Workers [31]—which enable JavaScript programs to spawn multiple threads that can communicate with one another—could be used to circumvent such transient memory protections by creating a race condition. While one thread coerces the language runtime into making a particular memory region writable (e.g., by triggering an inline cache patch), another thread exploits a vulnerability that corrupts the writable JIT code memory. This attack does, however, require that the attacker be able to predict the address of the JIT code memory that will be made writable by the former thread, which is an assumption that does not go hand in hand with a spraying attack.

JITDefender [9] and JITSafe [10] take a different approach and seek to prevent malicious code reuse; they allow JIT code to be writable at all times and enable executability only when the language runtime enters it. JITSafe and JITDefender incur $<$1% overhead, but they do not protect against JIT spraying when the control flow vulnerability is triggered by JavaScript. For example, each JIT sprayed function containing the JIT spraying payload could call another instance of the function in order to make all copies simultaneously executable; the last callee in the chain would then trigger the control flow vulnerability.

2) Dual mapping: As we saw in §IV-B1, varying JIT code's writability and executability over time can be vulnerable to race condition attacks. Furthermore, toggling between RW and RX protections is not even possible on all systems. SELinux prohibits processes from adding executability to memory if it has ever been writable in the past [13] in order to thwart code injection attacks that attempt to grant executability to shellcode injected into a data buffer.

A proposed alternative to transient memory protection is creating separate virtual memory mappings with immutable RW and RX protection status for physical JIT code pages. These mappings are either contained within a single process or split between two processes. The SpiderMonkey team experimented with a single-process dual mapping scheme [22] and found that it incurred about a 1% slowdown, but it

was not deployed to production because an information leak vulnerability could reveal the locations of both mappings and eliminate the defense's effectiveness. Other research efforts address the shortcomings of single-process dual mapping by separating the RW and RX mappings of JIT code across process boundaries. A trusted process holds either the RW or RX mapping, and an untrusted process holds the complementary mapping. However, introducing this separation into an existing language runtime requires invasive process re-architecting as well as runtime overhead in the form of inter-process communication. Lobotomy [16] and SDCG [28] take opposite approaches and allow the untrusted process to handle the RW and RX mappings, respectively. Overhead for these dual mapping defenses is high. Lobotomy suffers a mean overhead of 387%, and SDCG takes a 16.6% performance hit.

C. Diversification

The cornerstone of all attacks that reuse JIT-emitted code for malicious purposes [7], [26], [27], [25], [17], [4] is the assumption that, for a given input program, a particular JIT compiler will always emit the same sequence of machine instructions (modulo memory addresses embedded in the code). Diversification defenses seek to invalidate this assumption, thereby undermining the utility of JIT-emitted code for malicious reuse. Diversification defenses can be organized into the following two categories: intra-instruction randomization and code layout randomization. Because most diversification defense proposals in the literature combine various flavors of diversification defenses and evaluate them on varying platforms and benchmarks, rather than report the overheads found in the literature, we refer the reader to §V-F wherein we report the overheads of our own implementations of diversification defenses, evaluated under a unified testing framework.

1) Intra-instruction randomization: Intra-instruction randomization defenses invalidate an attacker's assumption that particular instruction encodings will appear in JIT code memory. The three types of intra-instruction randomization that appear in the literature are register randomization, constant blinding, and call frame randomization.

a) Register randomization: Register randomization permutes register assignment during compilation, resulting in instructions whose register operands are unpredictable [32], [33] and can often be performed with nominal overhead at compile time, though some runtime overhead may be added due to instructions that require longer encodings for certain register operand values.

b) Constant blinding: Immediate operands can consume a large percentage of an instruction's encoding and are often derived from untrusted values provided in the code being compiled. This grants attackers a large amount of control over the code produced by the JIT compiler, enhancing its utility for malicious code reuse. Indeed, predictable immediate operands have been a cornerstone of many JIT spraying attacks. Constant blinding [32], [25], [33], [10], [14] seeks to eliminate this predictability. A typical implementation of constant blinding splits each instruction that contains an untrusted immediate operand into two instructions whose respective immediate operands are functions of the untrusted immediate and a random secret value. The side effect of the composition of the two new instructions is the same as that of the original instruction, but neither of the new immediate operands are predictable. The overhead of constant blinding comes in the form of both increased code footprint and increased execution time. Athanasakis et al. [4] estimate that constant blinding can result in up to 80% additional instructions.

Lian et al. [17, §VI.A] found that JavaScriptCore employs a very weak form of constant blinding; it only blinds a subset of constants and each with only 1/64 probability, which was insufficient to prevent a JIT spraying attack on ARM. In our study of V8, we found that it only blinds constants >0x1ffff that are being moved into a register or pushed onto the stack. This implementation fails to protect against even Blazakis' originally-publicized JIT spraying attack and only applies to x86-32 and x86-64. Athanasakis et al.'s [4] investigation into Internet Explorer's Chakra JIT uncovered that it always blinds constants >0xffff, but this was not enough to prevent them from using untrusted constants to encode ROP gadgets.

c) Call frame randomization: Call frame randomization [32] scrambles the instructions that are used to access values such as arguments, local variables, spilled temporary values, etc. in a function's call frame. These instructions usually access memory at predictable immediate offsets relative to the stack pointer or a call frame register, which as we show in §III-B, can provide an attacker with a convenient predictable memory access instruction for reuse.

2) Code layout randomization: Predictable are not only the contents of JIT code, but also the layout of its instructions relative to one another and the boundaries of coarser-grained memory allocation units. Nearly all JIT spraying attacks we have seen so far rely on predictable code layout either to prevent an unintended instruction stream from resynchronizing to the intended stream or to predict the relative or absolute locations of instructions. Two strategies have been proposed to diversify JIT code layout at both fine and coarse granularity: random NOP insertion and base offset randomization, respectively.

a) Random NOP insertion: Random NOP insertion [14], [15], [32], [25], [33] involves injecting semantic NOP instructions at random in JIT code. Its effect is both to randomize the offset of any given instruction from the beginning of the unit of code compilation and, more generally, to randomize the relative offset between any given pair of instructions. Like constant blinding, the overhead of random NOP insertion comes from increased code footprint (leading to increased i-cache pressure) and wasted cycles at runtime; however the overhead for random NOP insertion scales with code size rather than the number of untrusted constants compiled. Lian et al.'s [17] study of JavaScriptCore revealed that JSC's non-optimizing JIT avoids the scaling overhead problem by only randomly inserting a single NOP instruction at a fixed location at the beginning of certain units of code compilation, which provides very little security benefit. Athanasakis et al. [4] report that Internet Explorer's Chakra JIT employs random NOP insertion, but they omit details of its implementation.

b) Base offset randomization: Base offset randomization [25] places a random amount of "dead" space before the beginning of each unit of code compilation, either in the form of NOP instructions or free space. This perturbs both the offset of the first unit of code compilation

when the JIT compiler maps a fresh region of executable memory and the relative offsets between consecutively-compiled units of code compilation. The absence of base offset randomization is critical to the heap feng shui [29] used to pinpoint gadget addresses in gadget chaining attacks such as [17] and our V8 attack (§ III-B). Base offset randomization would have drastically reduced the reliability of these attacks with less overhead than random NOP insertion.

V. Evaluation of diversification mitigations

As we saw in § IV, very few JIT spraying mitigations that have been proposed have been deployed in production browser releases, and those like constant blinding and random NOP insertion that are deployed have been severely limited to the point that they have lost their effectiveness. We argue that JIT code reuse can be effectively mitigated via fully-functional diversification defenses with only modest, but ultimately worthwhile, overhead. However, the answers to the questions of how much performance overhead diversification defenses incur and to what extent they improve security are muddled. Various incarnations of the diversification mitigations described in § IV-C are mixed and matched to compose many different defense systems mentioned in the literature [3], [4], [10], [14], [15], [32], [33]. Many of these implementations are not fully specified, and what descriptions exist often vary considerably. Moreover, performance evaluations of diversification mechanisms are often reported as aggregates with each other and other unrelated mitigations; and the hardware and benchmarking suites on which the implementations are evaluated vary by author.

Thus, there has been no clear source in the literature providing detailed implementation descriptions and measurements of their associated runtime overheads on consistent hardware and benchmarks. The purpose of this section is to provide that information so that JIT compiler authors considering adopting these defenses can more comfortably weigh the costs and benefits of diversification defenses. To better understand the benefits of each defense, we also analyze each defense with respect to concrete JIT spraying attacks to quantify the factor by which the probability of a successful attack is reduced.

To this end, we implemented all five diversification defense techniques described in § IV-C on the SpiderMonkey JavaScript engine for both its non-optimizing (Baseline) and optimizing (IonMonkey) JIT compilers on the ARM32 and x86-64 architectures.[5] Our implementations are by no means highly optimized; instead, our priority is to avoid creating corner cases that can be exploited by a wily attacker to improve her chances of defeating the mitigation. During development, we found that random design decisions in the JIT backend greatly impacted the difficulty of integrating defenses into the existing system. That is not to say that these decisions were made carelessly, but rather that they were perhaps not made with thought towards the generality necessary to support mitigations. The source code for our mitigations is available as a public fork of Mozilla's GitHub repository; our work is based on top of commit `ce31ad5`.[6]

[5]We did not implement register randomization for x86-64's non-optimizing compiler for reasons described later.

[6]The fork can be found at https://github.com/wwlian/gecko-dev.

A. Register randomization

Implementing register randomization for IonMonkey is extraordinarily non-invasive. IonMonkey compiles scripts to an intermediate representation (IR) and performs analyses over the IR in order to run a register allocator. We simply permute the order in which the allocator considers physical registers to satisfy allocation requirements. The changes for our implementation span 6 lines of code and randomize both floating point and general purpose register allocations. Some IR instructions are assigned fixed source or destination registers which cannot be randomized at the level of the register allocator; however, these fixed assignments do not bind to actual physical registers, but rather "abstract registers" which are mapped to physical registers. Fortunately, randomizing registers for the Baseline JIT involves randomizing the mappings from these same abstract registers to physical registers.

SpiderMonkey's Baseline JIT does not use a register allocator; instead it emits statically-defined instruction sequences for bytecode instructions for a stack-based virtual machine. The instruction sequences implementing bytecodes are defined by C++ code that allocates abstract registers as source, destination, and temporary registers used by each bytecode's implementation. To the C++ programmer, using an abstract register "feels" like using a physical register, but they are simply variables named after the physical registers that hold an integer value identifying an actual physical register.

Randomizing registers for the Baseline JIT (and indirectly, IonMonkey) involves permuting the underlying values that are assigned to the abstract register variables named after physical registers. Any uses of these variables will propagate the randomized physical register mapping. However, additional complexity must be introduced at the call and return control flow edges between statically-compiled native code and JIT code since certain values are expected to be passed between native and JIT code in specific physical registers in accordance with the architecture's ABI. To ensure that JIT code—which is defined in C++ under the assumption that abstract registers named after physical registers actually refer to those physical registers—is able to conform to the architecture ABI, we introduce a sequence of pushes and pops into the trampolines that execute at the boundaries between native and JIT code; the pushes and pops have the effect of permuting registers or inverting the permutation as needed.

In addition to the interoperability issues with native code, there were other cases where assumptions regarding the bindings between abstract registers and specific physical registers caused no small number of headaches. In these corner cases, violating these assumptions via randomization leads to incorrect behavior and data corruption that eventually causes a hard-to-debug crash much later than the misbehavior itself. These corner cases were very difficult to track down because the assumptions relied on very low level details that were not documented in any central location. For example, ARM is able to load two 32-bit values from memory into two consecutively-numbered general purpose registers as long as the lower-numbered register is even. If C++ code used abstract registers named after qualifying registers for such a load, randomization can easily invalidate the consecutivity, parity, and ordering assumptions.

Floating point register randomization is unnecessary for the Baseline JIT because it does not generate code that operates on floating point registers (with the exception of IC stubs, which are shared and cannot be sprayed). Instead, floating point values in Baseline JIT code are stored in general purpose registers and passed to IC stubs or host functions which perform the desired floating point operations.

Special care must also be taken to maintain abstract registers' volatility; in other words, we only permute volatile (a.k.a. caller-saved) registers with other volatile registers and likewise for non-volatile (callee-saved) registers. This is necessary because there are instances where code using an abstract register assumes that it maps to a non-volatile register and does not save that register's value prior to calling a subroutine. This limitation only applies to the abstract-to-physical remapping; in IonMonkey, values that are not bound to an abstract register are free to be allocated to any register.

Because of the many intricacies of permuting the mapping from abstract registers to physical registers, we limited our remapping implementation to the ARM architecture. We also limit randomization to registers that SpiderMonkey considers "allocatable," which excludes the program counter, stack pointer, link register, and a register used internally by the compiler for very short-lived scratch values. Although it presents a significant weakness to our implementation, we do not randomize the abstract register mapping that refers to the architecture ABI's integer return register, as a considerable amount of code assumes that it is not randomized. It should absolutely be randomized, but this is presently left for future work.

The probability that an attacker's payload will be emitted as expected if it requires k out of n randomized registers to be correctly mapped is $(n-k)!/n!$.

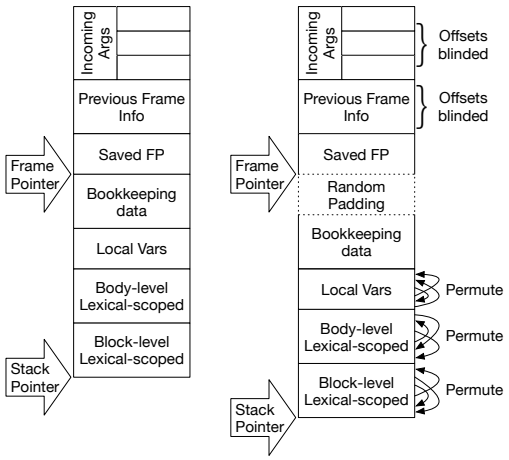
B. Constant blinding

Most constant blinding implementations observed in real-world code only blind a limited subset of instructions or untrusted constants, presumably under the rationale that some instructions and constants are harmless and therefore represent unnecessary overhead if blinded. Our implementation blinds every untrusted (meaning that it appears in the JavaScript code) integer and floating point constant. Untrusted constants only appear in Baseline JIT code as values that are loaded into registers; we protect Baseline JIT code by blinding those load instructions using the canonical `mov reg, blinded_val; xor reg, secret` instruction sequence, where `secret` is an immediate with a unique[7] 32-bit random value, and `blinded_val` is an immediate whose value is `secret` $\oplus$ `untrusted_constant`.

We implement constant blinding for IonMonkey by injecting blinding instructions into the architecture-independent intermediate representation (IR) called MIR. MIR is compiled from SpiderMonkey's interpreter bytecode, optimized, lowered to an architecture-dependent IR, then finally compiled to native code. During construction of the MIR code, we tag constants found in the bytecode as untrusted so that we avoid blinding constants generated by the compiler that were not present in the JavaScript. After the MIR has been optimized, we replace untrusted constants that remain with a sequence of MIR instructions that implement constant blinding in the same XOR-based manner as the Baseline JIT's constant blinding. It is important to apply constant blinding only after optimizations have completed to avoid potential constant folding optimizations, which would undo the blinding.

In order to predict the code sequences that will be generated for any given untrusted k-bit constant, the attacker must guess each corresponding secret bit, for which she has a success probability of 2^{-k}. Since all constants are blinded with unique secrets, the attacker's probability of predicting the code for the n untrusted constants, each k bits long, needed for her JIT spraying payload is 2^{-kn}.

[7]Unlike some constant blinding implementations (e.g., V8) which share the same secret value among constants within the same compilation unit, ours generates a fresh secret for each constant.



(a) Unrandomized (b) Randomized

Fig. 6: Baseline JIT call stacks with and without call frame randomization. Stacks grow downward.

C. Call frame randomization

SpiderMonkey's Baseline JIT and IonMonkey JIT use very different call frame conventions requiring different treatment to randomize. The Baseline JIT uses a frame pointer relative to which all call frame elements are accessed and pushes outgoing function arguments onto the stack as part of its implementation of a stack-based virtual machine. IonMonkey, on the other hand, performs frame pointer omission (i.e., call frame elements are accessed relative to the stack pointer) and pre-allocates enough stack space during each function prologue to fit the maximum number of outgoing function call arguments across all calls within that function. For both Baseline and IonMonkey call frames, we randomly shift the frame pointer and stack pointer, respectively, relative to the call frame elements. Whenever possible, we also permuted call frame elements of the same type, and when neither of the above was possible, we performed a process similar to constant blinding to the load/store instructions accessing the call frame.

1) Baseline JIT: Figure 6a illustrates the layout of an unrandomized Baseline JIT call frame. All elements shown are accessed at statically-computed frame pointer-relative offsets. Baseline JIT code stores three types of local variables on the stack above the frame pointer, and we randomize their offsets by permuting their orders within each type and randomly increasing the size of the bookkeeping data structure pushed onto the stack before them, shown as the dotted box in

Figure 6b labeled "Random Padding." The size of this padding is determined for each function at JIT compile time and is between 0 and 15, inclusive, units of stack alignment (the size of which is 8 bytes for ARM32 and MIPS32 and 16 bytes otherwise). Because that does not change the frame pointer's relative offset to the incoming function arguments and "Previous Frame Info," we modify instructions that access them in a manner similar to constant blinding. Rather than accessing those elements at fixed offsets relative to the frame pointer, each access site populates a scratch register with the value of the frame pointer minus a unique random multiple of 4 in the range $[0, 64)$[8] and performs the access using a new offset that corrects for the shifted base register value.

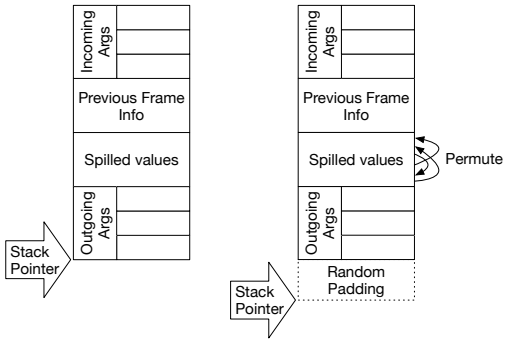
We do not shift the incoming arguments by injecting stack padding between them and the frame pointer location because the great complexity of the code that traverses and unwinds the call stack makes doing so very difficult. Similarly, we do not permute incoming arguments on the stack because of the complex interactions between their stack positions and SpiderMonkey's implementaion of JavaScript features like rest parameters, default parameters, and the `arguments` object. However, permuting call frame elements is secondary to shifting their offsets since permutation without shifting is vulnerable to the corner case where there is only one element to permute.

An attacker is only able to predict the blinding offset of an incoming argument access site with probabiity $\frac{1}{16}$. Therefore, if an attack relies on reusing n incoming argument access sites, there is only a $\frac{1}{16^n}$ chance that she will be able to predict all necessary access site offsets. The deviation of the frame pointer offset of a local variable or body/block-level lexically-scoped variable from its unrandomized value can be interpreted as the linear combination of independent discrete uniform random variables. In particular, if there are n variables of a certain type (local, body-level lexical, or block-level lexical), the offset shift of the ith $(0 \leq i < n)$ variable of that type, is given by $Z_i = -(X_i + Y)$, where X_i is the shift due to permutation and is distributed as $X_i \sim 8 \cdot (U\{0, n-1\} - i)$; and Y is the shift due to padding the bookkeeping data structure and is distributed as $Y \sim a \cdot U\{0, m-1\}$, where a is the size of a unit of stack alignment in bytes (which varies by architecture), and m is the number of possible padding amounts ($m = 16$ for our implementation).

2) IonMonkey JIT: Figure 7a shows the IonMonkey call frame layout. Since IonMonkey performs frame pointer omission, and all call frame elements are below the stack pointer, we can shift all call frame accesses by pushing a random amount of padding—whose size is determined once for each compilation unit at JIT compile time—on top of the call frame. The size of the padding is between 0 and 15, inclusive, units of stack alignment (the size of which is 8 bytes for ARM32 and MIPS32 and 16 bytes otherwise). IonMonkey does not store local values on the stack unless they must be spilled due to register contention; we permute the order that these spilled values are allocated to stack slots. Figure 7b illustrates the application of these randomizations to an IonMonkey call frame.

Since the stack pointer-relative offset of all non-spill values is shifted by a common value, an attacker's probability of guessing any number of non-spill value offsets is reduced by a factor of 16. Spilled values have their values shifted according to a distribution similar to the one described above for Baseline JIT locals and lexically-scoped variables. One difference is that spilled values may have varying sizes that are a multiple of 4 bytes, so the permutation distribution is not uniform.

[8]We use a multiple of 4 because the ARM code generation backend can compile an optimized instruction sequence for certain cases where the offset is a multiple of 4.



(a) Unrandomized (b) Randomized

Fig. 7: IonMonkey call stacks with and without call frame randomization. Stacks grow downward.

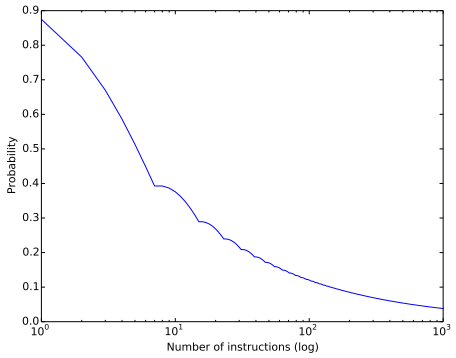


Fig. 8: Probability that attacker's best guess correctly predicts the number of random NOPs inserted in n instructions.

D. Random NOP insertion

Our implementation of random NOP insertion places a single NOP instruction before each intended instruction with probability $p = \frac{1}{8}$. There is a small handful of exceptions where random NOP insertion must be disabled due to assumptions in the JIT's implementation regarding the precise layout of a section of emitted code.

Random NOP insertion's security benefit depends on the attacker's needs. If the attacker needs to predict the offset of the nth instruction from the beginning of its unit of code compilation, she must predict the number of NOPs that will be inserted before it, given by the random variable $X \sim B(n, \frac{1}{8})$. The attacker's best guess is the mode of X; the probability that this best guess will be correct as a function of n is shown in Figure 8.

Alternately, an attacker might require n consecutive instructions with no random NOPs inserted between them (e.g., when the payload uses instructions decoded at unintended instruction boundaries as in [7]), which occurs with probability $(1-p)^{n-1} = \left(\frac{7}{8}\right)^{n-1}$.

TABLE III: Geometric mean of code size increases incurred by diversification defenses when executing benchmark suites

	x86-64	32-bit ARM
Register randomization	-0.008%	1.01%
Constant blinding	0.433%	1.56%
Call frame randomization	2.79%	1.31%
Random NOP insertion	2.67%	12.58%
Base offset randomization	2.39%	2.52%
All	8.57%	18.15%

E. Base offset randomization

Whenever SpiderMonkey needs to find space to place a unit of code compilation, it rounds up the size of the JIT code to an architecture-specific alignment granularity G and searches memory pools of allocated code memory for free space to accomodate the rounded size. We randomize the base offsets of each unit of code compilation by randomly increasing the size of its allocation request by rG where $r \in \{0, 1, 2, ..., 15\}$ and shifting the code rG bytes further into the allocation. The attacker's probability of guessing the base offset shifts for n consecutive units of code compilation is $\left(\frac{1}{16}\right)^n$.

F. Benchmark results

We evaluated the performance overheads of our implementations on x86-64 and 32-bit ARM using the SunSpider 1.0.1, Kraken 1.1, and Octane v.2 benchmarks suites. Results for x86-64 were gathered on a quad-core Intel Core i7-870 2.93GHz processor with 16GB RAM running Ubuntu 14.04.2 LTS with kernel version 3.13.0-49-generic. Results for 32-bit ARM were gathered on an octa-core AppliedMicro APM883208-X1 ARMv8 2.4GHz processor with 16GB RAM running 32-bit Debian 8.0 in a chroot jail on APM Linux with kernel version 3.12.0-mustang_sw_1.12.09-beta.

To evaluate our implementations, we built an unmodified version of SpiderMonkey and a separate binary for each diversification mechanism. We also built a binary that deploys all implemented defenses. We executed each benchmark suite 100 times for each binary and computed the arithmetic means for each group of 100 benchmark scores. We computed the overhead imposed on the results of "smaller-is-better" benchmarks (SunSpider and Kraken) as $\bar{v}/\bar{u} - 1$, where $\bar{v}$ is the arithmetic mean of the 100 benchmark scores for a particular modified binary, and $\bar{u}$ is the arithmetic mean of the 100 benchmark scores for the unmodified binary. Octane is a "bigger-is-better" benchmark whose scores are derived from a "smaller-is-better" measurement by dividing a constant value by the measurement, so we compute the overhead on its results as $\bar{u}/\bar{v} - 1$.

The measured overheads of our implementations are shown in Table II. We used Welch's unequal variances t-test to test the mean benchmark score from each variant's 100 execution sample against the mean benchmark score from the unmodified binary's 100 execution sample and indicate statistically-significant ($p < 0.05$) changes to the mean by printing the corresponding overhead with boldface type. Note that overheads are not independent and cannot necessarily be added.

G. Code size increase

To measure the impact of our mitigation implementations on the memory demands of the JIT, we instrumented SpiderMonkey to emit the file name, line number, and number of JIT code bytes used each time unit of code compilation is compiled and executed the three benchmark suites once on each binary variant. Let $\bar{v_i}$ be the average code size of the ith file name-line number pair, as emitted by a binary variant implementing a one or more diversification defenses; let $\bar{u_i}$ represent the average code size for the ith file name-line number pair, as emitted by an unmodified SpiderMonkey binary. We compute the increased memory usage for each variant as $\sqrt[n]{\prod_{i=1}^{n}(\bar{v_i}/\bar{u_i})} - 1$. Table III shows the code size increases for each mitigation, broken down by architecture.

H. Concrete security analysis

In order to give concreteness to the security benefits offered by our diversification defense implementations, we report on our analysis of the estimated relative success probability of four concrete JIT spraying attacks when launched against individual diversification mitigations. The results are shown in Table IV. Remember that since some defenses may interact with one another, these relative probabilities may not necessarily be combined by multiplication.

Our estimates are conservative; in order to make the relative probabilities concrete, we have made assumptions in the attacker's favor when necessary. For example, random NOP insertion can disrupt Blazakis' JIT spray by causing its NOP sled to resynchronize to the intended instruction stream, but the probability of at least one NOP interrupting execution depends on how many uninterrupted instructions the attacker requires, including the NOP sled. This in turn depends on where in the NOP sled the attacker's control flow vulnerability happens to divert control. Therefore, we conservatively assume that the attacker's control flow vulnerability directs execution to the head of the shellcode and only compute the probability that a NOP will not be inserted into the sequence of instructions needed to encode a very short 10-instruction shellcode. We consider minor adaptations to the existing attacks that allow them to use the most likely diversification outcome when possible, but we do not claim to have considered the most optimized versions of each attack. Lastly, the values shown in Table IV only reflect the relative success of a single instance of the sprayed function.

Constant blinding, which incurs the greatest runtime overhead among the surveyed diversification defenses, performs tremendously well to mitigate [7], [17], and the self-sustaining ARM payload (§III-C), which rely on attacker-chosen constants appearing in JIT code. We also observe that register randomization and call frame randomization complement constant blinding by diversifying compiler-chosen operands, relied upon heavily by our V8 attack.

We were surprised to find that register randomization provided little defense against [7]. Blazakis' attack is able to fare well against register randomization because the attacker's payload is an unintended instruction stream (cf. § II) that skips over intended instruction opcodes (the XOR opcode, in the case of Blazakis' attack) as long as the number of bytes to be skipped is correctly predicted. The x86 XOR instruction's opcode is either 1 byte long when XORing against `%eax` or

TABLE II: Diversification performance overheads. Bold typeface in non-geometric mean columns denotes statistically-significant impact on the mean benchmark score of each 100-execution sample (Welch's unequal variances t-test, p <0.05); and negative overheads indicate improved performance. *Register randomization overhead for x86-64 only includes IonMonkey randomization.

	x86-64				32-bit ARM			
	SunSpider	Kraken	Octane	G. Mean	SunSpider	Kraken	Octane	G. Mean
Register randomization*	-1.94%	**-0.829%**	-0.404%	-1.06%	**1.62%**	**0.456%**	0.265%	0.777%
Constant blinding	-1.36%	**2.65%**	**2.93%**	1.39%	**1.62%**	**6.02%**	**4.39%**	3.99%
Call frame frandomization	-1.68%	-0.199%	0.324%	-0.523%	0.138%	**-2.26%**	**-1.05%**	-1.06%
Random NOP insertion	-0.762%	**2.12%**	**1.44%**	0.922%	**1.67%**	**1.76%**	**1.35%**	1.59%
Base offset randomization	-2.38%	-0.207%	**-0.846%**	-1.15%	0.498%	**0.302%**	0.223%	0.341%
All	1.71%	**5.70%**	**6.33%**	4.56%	**4.44%**	**5.48%**	**4.71%**	4.88%

TABLE IV: Estimated relative success probabilities for concrete attacks against single diversification defenses. Lower values indicate better mitigation.

	Register randomization	Constant blinding	Call frame randomization	Random NOP insertion	Base offset randomization
Blazakis 2010 [7] (x86-32)	92.9%	6.84×10^{-47}%	100%	30.1%	100%
Lian et al. 2015 [17] (ARM)	1.79%	1.53×10^{-3}%	100%	3.74%	0.391%
ARM V8 gadget chaining (§III-B)	0.909%	100%	6.25%	3.70%	0.229%
SpiderMonkey self-sustaining ARM payload (§III-C)	2.51×10^{-5}%	4.15×10^{-461}%	100%	7.95×10^{-21}%	100%

2 bytes long when XORing against any other register. The attacker can therefore assume with highest probability that the XOR opcode will be 2 bytes long and adjust her payload to skip over a 2-byte intended instruction opcode.

Random NOP insertion provides good protection across the board by both disrupting the unintended instruction streams used by Blazakis' attack and the self-sustaining ARM payload as well as diversifying the location of useful code gadgets used by [17] and our new V8 attack. Base offset randomization, on the other hand, only defends against the latter pair of attacks. Base offset randomization offers better defense than NOP insertion against a hypothetical attack in which the attacker needs to predict the offset of an instruction near the beginning of a unit of code compilation. If the instruction is one of the first 372[9] in a unit of code compilation, the attacker has a greater probability of predicting the number of random NOPs inserted before it (and by proxy its offset) than predicting the base offset randomization of a unit of code compilation.

VI. Discussion

The opportunities for both code corruption and code reuse made possible by JIT compilers have been shown to be exploitable against real JIT implementations time and time again [7], [27], [26], [6], [25], [24], [28], [4], [17]. Unfortunately, the community developing arguably the most prominent and vulnerable JITs—those found in web browsers—has been slow to react to effectively mitigate these threats in their JavaScript JITs. SpiderMonkey has taken the lead in code corruption mitigation by deploying W $\oplus$ X JIT memory, but it still lacks effective code reuse mitigations. Similarly, although Internet Explorer's Chakra deploys a form of random NOP insertion, its constant blinding implementation, which does not blind constants $\leq$0xffff, can still be bypassed to create ROP payloads [4]. It is apparent to these authors that crafty attackers can and will discover and exploit any corners cut in the implementations of JIT spraying defenses. For this reason, we urge JIT developers to consider deploying defenses at full strength, even at the expense of non-trivial performance overhead.

By reducing the predictability of instruction operands and code layout, JIT code reuse can be mitigated. Rather than selecting a subset of the diversification defenses evaluated in § V, we recommend all 5 be deployed simultaneously in order to leave no stone unturned. If a JIT were to omit a particular defense, that undiversified area would eventually become a valuable piece of a future attack. This observation is merely a corollary to our assertion in the previous paragraph that partial implementations of defenses do not go unpunished.

Our measurements in §V indicate that most diversification defenses can be deployed at full strength with only modest performance and memory overhead. A noteworthy exception is random NOP insertion for the ARM instruction set. Since our implementation inserts a NOP between each instruction with 12.5% probability, and all ARM instructions have the same size, we observe a ≈12.5% code size increase. The x86-64 architecture, in contrast, is able to achieve a much lower memory overhead thanks to its 1-byte NOP instruction encoding and variable-length instructions. JIT developers may wish to consider carefully lowering the probability of random NOP insertion on platforms with limited memory and fixed-width instructions. To establish a lower bound on memory overhead when random NOP insertion is dialed back, we measured the memory overhead for ARM32 when all diversification defenses except random NOP insertion are enabled and found it to be 6.15%.

[9]This figure is a function of the probability of random NOP insertion and the number of potential base offsets.

A potential objection to diversification defenses is that an attacker with arbitrary memory read capabilities can avoid failure by spraying until her payload is emitted as desired. While this is true, we argue that diversification defenses mitigate the entire class of blind JIT spraying attacks and raise the bar for malicious reuse of JIT code by mandating that a memory disclosure accompany the control flow vulnerability.

Very recently-published work [18] demonstrates the creation of unintended instructions from the offset field in x86's PC-relative branch and call instructions. To mitigate this threat; the authors of [18] implemented blinding of the implicit constants in relative branches. At submission time, our diversification implementations do not include this new mitigation, and we recognize that future work will need to correct this flaw.

VII. Conclusion

As JIT compilers continue to blanket the landscape of language runtimes, so do attacks that abuse their predictability and unique memory protection model, as evidenced by the new threats against ARM JITs introduced in this paper. Implementations of mitigations have appeared in production JITs, but their potential has been artificially limited in order to boost performance, resulting in a failure to provide satisfactory protection. The empirical evaluations and analyses in this paper demonstrate that diversification defenses are effective and can be implemented in full with only modest overhead. We encourage JIT developers to take our experiences implementing diversification defenses and use them to guide implementations on their own systems.

Acknowledgment

We would like to thank Eric Rescorla for thoughtful feedback. We also thank Jan de Mooij, Eric Faust, JF Bastien, and Michael Stanton for helpful discussions. This material is based in part upon work supported by the U.S. National Science Foundation under Grants No. 1228967, 1410031, and 1514435, and by gifts from Mozilla and Google.

References

[1] Property cache. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Property_cache. [Online; accessed 2 November 2015].

[2] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of CCS 2005* (2005), ACM, pp. 340–353.

[3] Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., and Yee, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of PLDI 2011* (New York, NY, USA, 2011).

[4] 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 NDSS 2015* (Feb. 2015).

[5] Bania, P. JIT spraying and mitigations. *arXiv preprint arXiv:1009.1038* (2010).

[6] Beck, P. JIT Spraying on ARM. https://prezi.com/ih3ypfivoeeq/jit-spraying-on-arm/, 2011.

[7] Blazakis, D. Interpreter exploitation: Pointer inference and JIT spraying. Presented at BlackHat DC 2010, Feb. 2010.

[8] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *In Proceedings of USENIX Security 2015* (Aug. 2015).

[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*. Springer, 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] De Groef, W., Nikiforakis, N., Younan, Y., and Piessens, F. JITSec: Just-In-Time security for code injection attacks. In *Proceedings of WISSEC 2010* (Nov. 2010), pp. 1–15.

[12] Deutsch, L. P., and Schiffman, A. M. Efficient implementation of the smalltalk-80 system. In *Proceedings of POPL 1984* (1984).

[13] Drepper, U. SELinux memory protection tests. http://www.akkadia.org/drepper/selinux-mem.html, Apr. 2009. [Online; accessed 3 November 2015].

[14] Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of CCS 2013* (2013), ACM, pp. 993–1004.

[15] Jangda, A., Mishra, M., and De Sutter, B. Adaptive just-in-time code diversification. In *Proceedings of ACM MTD 2015* (2015).

[16] Jauernig, M., Neugschwandtner, M., Platzer, C., and Comparetti, P. M. Lobotomy: An architecture for jit spraying mitigation. In *Proceedings of ARES 2014* (2014), IEEE, pp. 50–58.

[17] Lian, W., Shacham, H., and Savage, S. Too LeJIT to Quit: Extending JIT Spraying to ARM. In *Proceedings of NDSS 2015* (2015).

[18] Maisuradze, G., Backes, M., and Rossow, C. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *Proceedings of USENIX Security 2016* (2016).

[19] Mavaddat, F., and Parhami, B. Urisc: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education* (1988).

[20] Millikin, K. V8: High Performance JavaScript in Google Chrome. https://www.youtube.com/watch?v=lZnaaUoHPhs, 2008. [Online; accessed 2 November 2015].

[21] Mooij, J. D. W^X JIT-code enabled in Firefox. http://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox/, Dec. 2015.

[22] Mozilla. Bug 506693 - SELinux is preventing JIT from changing memory segment access. https://bugzilla.mozilla.org/show_bug.cgi?id=506693, 2009. [Online; accessed 3 November 2015].

[23] Niu, B., and Tan, G. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of CCS 2014* (2014).

[24] Pie, P. Mobile Pwn2Own Autumn 2013 - Chrome on Android - Exploit Writeup, 2013.

[25] Rohlf, C., and Ivnitskiy, Y. Attacking Clientside JIT Compilers. http://www.matasano.com/research/Attacking_Clientside_JIT_Compilers_Paper.pdf, 2011.

[26] Sintsov, A. JIT-Spray Attacks & Advanced Shellcode. Presented at HITBSecConf Amsterdam 2010. Online: http://dsecrg.com/files/pub/pdf/HITB%20-%20JIT-Spray%20Attacks%20and%20Advanced%20Shellcode.pdf, July 2010.

[27] Sintsov, A. Writing JIT Shellcode for fun and profit. Online: http://dsecrg.com/files/pub/pdf/Writing%20JIT-Spray%20Shellcode%20for%20fun%20and%20profit.pdf, Mar. 2010.

[28] Song, C., Zhang, C., Wang, T., Lee, W., and Melski, D. Exploiting and protecting dynamic code generation. In *Proceedings of NDSS 2015* (2015).

[29] Sotirov, A. Heap feng shui in javascript. *Black Hat Europe* (2007).

[30] Stachowiak, M. Introducing SquirrelFish Extreme. https://www.webkit.org/blog/214/introducing-squirrelfish-extreme/, 2008. [Online; accessed 2 November 2015].

[31] W3C. Web workers. http://www.w3.org/TR/workers/, 2015. [Online; accessed 3 November 2015].

[32] Wei, T., Wang, T., Duan, L., and Luo, J. Secure dynamic code generation against spraying. In *Proceedings of CCS 2010* (2010).

[33] Wu, R., Chen, P., Mao, B., and Xie, L. Rim: A method to defend from jit spraying attack. In *Proceedings of ARES 2012* (2012).

[34] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of IEEE S&P (Oakland) 2009* (2009).

[35] Zhang, C., Niknami, M., Chen, K. Z., Song, C., Chen, Z., and Song, D. JITScope: Protecting web users from control-flow hijacking attacks. In *Proceedings of INFOCOM 2015* (2015).