UNIVERSITY OF CALIFORNIA, SAN DIEGO

JIT Spraying Threats on ARM and Defense by Diversification

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Wing-Soon Wilson Lian

2016

The Dissertation of Wing-Soon Wilson Lian is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2016

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisors Hovav Shacham and Stefan Savage. It has been a long road, and without their support, encouragement, patience, and guidance, I would never have found my way to the end.

During these six years, I've had the joy of meeting many exceptional climbers and people who helped shape this experience into an enjoyable and memorable one. Thank you for all the belays, spots, and good times: Gabe Cavin, Rick Delhommer, Susan Delhommer, Kevin Fliflet, Mariko Kellogg, Brian Lewis, Amanda Li, Richard Lie, Garren Melton, Perry Naughton, Davis Ngo, Jason Oberg, Gunnar Poplawski, Sonia Rackelmann, Jarita Ta, and Max Takano.

Next, I'd like to acknowledge my close friend Daniel Ricketts. Thanks for the stimulating conversations, the numerous batches of beer brewed, the kilos of Kinder Schoko-Bons smuggled, and everything else. Your friendship and support have been indispensable.

I'd also like to thank my family for their support, especially my parents for supporting me throughout my life and long career as a student.

Thanks as well to Ariana Mirian for her help proofreading this dissertation.

A big thanks goes out to Brian Kantor for his help purchasing and configuring domains for the DNSSEC measurement paper.

Lastly, I'd like to thank my co-author Eric Rescorla for his patience, insight, and hard work on the DNSSEC paper.

Chapters 1, 3, 4, 5, and 8, in part, are a reprint of the material as it appears in *Proceedings of the 2015 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2015. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 4, 6, 7, and 8, in part, have been submitted for publication of the

material as it may appear in appear in *Proceedings of the 2017 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2017. The dissertation author was the primary investigator and author of this paper.

VITA

2009	Bachelor of Science in Computer Science University of North Carolina at Chapel Hill
2010–2016	Research Assistant University of California, San Diego
2013	Master of Science in Computer Science University of California, San Diego
2016	Doctor of Philosophy in Computer Science University of California, San Diego

PUBLICATIONS

Wilson Lian, Hovav Shacham, Stefan Savage. “Too LeJIT to Quit: Extending JIT Spraying to ARM.” In Proceedings of the Network and Distributed System Security Symposium, 2015.

Wilson Lian, Eric Rescorla, Hovav Shacham, Stefan Savage. “Measuring the Practical Impact of DNSSEC Deployment.” In Proceedings of the USENIX Security Symposium, Washington D.C.,

ABSTRACT OF THE DISSERTATION

JIT Spraying Threats on ARM and Defense by Diversification

by

Wing-Soon Wilson Lian

Doctor of Philosophy in Computer Science

University of California, San Diego, 2016

Professor Stefan Savage, Co-Chair
Professor Hovav Shacham, Co-Chair

Just-in-Time compilers offer substantial runtime performance benefits over traditional execution methods like interpretation; and they have enjoyed widespread deployment in the JavaScript engines found in nearly all modern web browsers. Unfortunately, security has taken the back seat to performance in many JIT compilers, despite the often untrusted nature of their inputs and the tremendous privilege that they have been granted to generate machine code on the fly. While the concerns regarding performance are understandable, the threat posed by blind JIT spraying has been underestimated.

In this dissertation, we demonstrate the feasibility of blind JIT spraying on the

ARM architecture against three modern JavaScript engines, despite many restrictions imposed by ARM such as coarse-grained instruction boundaries and limited space for encoding immediate operands. We find that useful instruction decoding ambiguity can be leveraged to create a blind JIT spraying payload using either intended ARM or Thumb instructions. Furthermore, we demonstrate that instruction decoding ambiguity is not necessary in the construction of a blind JIT spraying payload. We also introduce a technique for abusing JIT sprayed code called gadget chaining, which enables an attacker to exploit even limited control over JIT code.

To better understand the state of JIT spraying mitigation research and deployment, we survey the literature and examine four open source JavaScript engines several years after the debut of JIT spraying on x86. We find that all four engines cut corners in their implementations—often quite egregiously—in the name of reducing performance overhead.

In order to form a consolidated picture of the costs and benefits of diversification defenses, we implement five diversification defenses, without cutting corners, on the SpiderMonkey JavaScript engine for 32-bit ARM and x86-64 and empirically evaluate their overheads across a consistent set of benchmarks and hardware. We find that all five diversification defenses can be deployed in tandem with reasonable security parameters at a runtime overhead cost of $<5\%$. Our analysis of the diversification defenses indicates that, in combination, they provide effective mitigation of the blind JIT spraying threat for less overhead than other effective options.

Introduction

The common ground shared by all computers is software that is—either directly or indirectly—written by human developers who inevitably make mistakes and introduce bugs into their software. In most cases, the consequences of these bugs are not severe, but in others, they can create an opportunity for a malicious party interacting with the software to effect undesirable and, sometimes, even harmful computation.

The classic example of a security-relevant bug is the buffer overflow vulnerability, in which the vulnerable program copies a user input into a fixed-size buffer without first checking whether the input's length exceeds that of the buffer. If the input is too large, part of its contents can overflow the buffer and overwrite memory adjacent to the buffer. By carefully crafting an input used to overflow a buffer allocated on the stack, an attacker can corrupt the saved function return address and divert control flow to an address of her choosing. To exploit this "control flow vulnerability," the attacker could inject a malicious program into the victim's address space by injecting it as part of an input to the vulnerable program [4], or she could reuse pre-existing code by directing control flow to a shared library function's entry point [25] or even a short snippet of code in the middle of a library function or the program's own binary [46].

Defenses against corruption of the saved return address [1, 20, 28] and buffer overflows [45] have been developed, and some have even been widely deployed. However, they fail to provide comprehensive prevention of control flow vulnerabilities. Other defenses such as control flow integrity [3] and software-based fault isolation [57] take more generalized approaches to preventing control flow vulnerabilities, but a combination of runtime overhead and difficulty of integration with legacy binaries has prevented their widespread use. Consequently, control flow vulnerabilites remain a threat, and much effort is still poured into mitigating an attacker's ability to successfully exploit them.

In an effort to prevent exploitation of control flow vulnerabilities with code injection and code reuse attacks, the default behavior of nearly all modern operating systems is to prohibit data from being executed and to randomize the base addresses of the program's stack, heap, binary image, and shared libraries. With these defenses, known as W $\oplus$ X and ASLR, respectively, in place, the threat posed by control flow vulnerabilites superficially appears to be mitigated, but unfortunately, this is not the case. With the aid of a vulnerability that leaks the address of any shared library function, the randomized addresses of all other library functions can be computed [47]; similarly, a bug which allows an attacker to perform arbitrary memory reads can completely undermine randomization of the address space layout. Such additional vulnerabilities do exist in real programs, but even in their absence, a unique attack surface remains in the form of JIT-compiled code.

The growing popularity of the Internet has driven the growth of the web browser as a platform for client-side software, much of which is distributed as high level language code or portable bytecode to support a diverse ecosystem of machines. However, interpreted code is nowhere near as performant as its native code counterpart. To reap the performance benefits of native code, language runtimes both inside and outside web browsers began including components to compile bytecode and source code to native code on the fly. These Just-In-Time (JIT) compilers drastically improved performance but opened the door to a new class of attack—the JIT spraying attack.

The role of a JIT compiler is to take arbitrary bytecode or high level language code as input and produce new, executable native code derived from it. Therefore, by carefully crafting an input to a JIT compiler, an attacker can trick it into unwittingly emitting native code that is useful to some malicious intent of the attacker. Moreover, the attacker has the ability to cause the JIT compiler to generate as much of this code as she desires, and she can leverage this to “spray” useful native code throughout the victim’s address space.

The ability to abuse a JIT compiler to dynamically allocate large swaths of executable memory and fill them with malicious code allows an attacker to defeat both W $\oplus$ X and ASLR. Even if the attacker does not know the precise randomized address of code emitted by the JIT compiler, a random address chosen *blindly* (i.e., without the aid of an information leak vulnerability) is likely to point to malicious JIT code after enough of it has been generated. Consequently, triggering a control flow vulnerability in the JIT compiler or any program that embeds it (e.g., a web browser) to branch to a random address is likely to execute JIT sprayed code, which will perform malicious computation on behalf of the attacker.

Prior work has primarily focused on JIT spraying against the x86 architecture, and many defenses against it have been proposed, but few have been deployed with the gusto required to effectively mitigate the threat of JIT spraying. Instead, many defense implementations favor performance over covering all corner cases, leaving opportunities for exploitation as a consequence. The contributions of this dissertation are as follows:

- The extension of JIT spraying to the ARM architecture
- A comprehensive survey and analysis of JIT spraying mitigations in the literature
- Open source implementations of full-fledged diversification defenses for 32-bit ARM and x86-64 on a single JavaScript engine
- Empirical evaluations of the runtime and memory overheads of the above implementations on a consistent set of benchmark suites and hardware testing platforms.

These evaluations indicate that fully-implemented diversification defenses are practical and effective.

Outline

Chapters 1, 2, and 3 provide context to our work. In Chapter 1, we give a more technical description of the attack and defense landscape leading into the debut of JIT spraying and the JIT spraying attack against x86 itself. Chapter 2 outlines our assumptions regarding a blind JIT spraying attacker's capabilities and the defense mechanisms against which she contends. Chapter 3 describes key features of the ARM architecture.

In Chapter 4, we consider an attacker's options for controlling the output of a JIT compiler and discuss the challenges of creating different types of blind JIT spraying payloads for the ARM architecture. We find that the design of ARM's instruction set encodings leads introduces no small amount of complexity to this task.

In Chapters 5 and 6, we describe proof of concept blind JIT spraying attacks against the JavaScriptCore JavaScript engine for ARM—which is used by all major web browsers on Apple's iOS mobile operating system—and the V8 JavaScript engine for ARM found in the Chrome web browser on all operating systems except iOS. In Chapter 7, we present a proof of concept blind JIT spraying payload, but not a full end-to-end attack, against the SpiderMonkey JavaScript engine for ARM. SpiderMonkey is the JavaScript engine that ships with the Mozilla Firefox web browser on all operating systems except iOS.

In Chapter 8, we turn our attention to JIT spraying mitigations, beginning with a survey of those proposed in the literature and an investigation of those deployed into actual JavaScript engines. We find that JIT spraying mitigations are in short supply in most real world JIT compilers, and almost all of those that exist do so only in enfeebled forms that minimize their performance impact at the cost of any substantive protection. We then describe a suite of diversification defenses that we implemented for the SpiderMonkey JavaScript engine which seeks to prioritize security over performance and present the results of an empirical evaluation of the performance and memory overheads thereof. We find that in exchange for only modest runtime overhead ($<5\%$), diversification defenses are capable of mitigating all known blind JIT spraying attacks.

Finally, we conclude in Chapter 9.

Chapter 1

Background

History holds no shortage of avenues through which an adversary can exploit a control flow vulnerability to perform arbitrary malicious computation on a victim's machine. Roughly speaking, they can be broken down into the following two categories: code injection attacks and code reuse attacks.

A code injection attack such as Aleph One's infamous stack smashing attack [4] introduces new code masquerading as data into a vulnerable process's address space and exploits a control flow vulnerability to divert execution to it. W $\oplus$ X (a.k.a. Data Execution Prevention (DEP), Exec Shield, etc.) [5] has become the standard defense against code injection. W $\oplus$ X is a defense mechanism that enforces the separation between data pages such as the stack and heap—which should be writable but not executable—and code pages—which should be executable but not writable. W $\oplus$ X is widely supported in hardware in modern Intel, AMD, and ARM chips; if a W $\oplus$ X violation occurs (e.g., branching to a non-executable page), a fault is raised.

Enter code reuse attacks, which circumvent W $\oplus$ X by repurposing instructions found within the vulnerable process's own executable memory as the building blocks for malicious computation. Canonical examples of code reuse attacks are the return-to-libc attack [25] and return-oriented programming (ROP) [46]. The most widely deployed defense against code reuse attacks is Address Space Layout Randomization (ASLR).

Code reuse attacks require the attacker to guess the addresses of the instructions they intend to repurpose for their malicious computation with pinpoint accuracy. ASLR makes that task more difficult by randomizing the locations of objects in a process's virtual address space. Typically these objects are the stack, the heap, shared libraries, and the process's binary image.

In the absence of W $\oplus$ X, an attacker can reduce ASLR's effectiveness with a heap spraying attack (e.g., [50]). Heap spraying is a code injection technique in which a buffer containing the attacker's malicious code is programmatically generated and copied (i.e., "sprayed") to multiple locations throughout the heap. Each instance of the malicious code—often called *shellcode*—is preceded by a much longer sequence of NOP instructions called a *NOP sled*. The purpose of a NOP sled is to increase the surface area of addresses that can be branched to in order to execute the shellcode from start to finish. Without NOP sleds, the only way to ensure that an instance of shellcode executes in its entirety is to branch precisely to its first instruction. ASLR prevents the attacker from accurately predicting where injected instances of shellcode will reside in memory; but if enough copies of NOP sled-prefixed shellcode are sprayed into memory, a randomly-chosen address is likely to point into a NOP sled. If an attacker can successfully target such an address with a control flow vulnerability, execution beginning in the NOP sled will "slide down" through the NOP sled into the first instruction of the shellcode.

Shacham et al. [47] demonstrated another technique for circumventing ASLR in a 32-bit address space to bootstrap a code reuse attack, even with W $\oplus$ X enabled. The attack learns the randomization offset of the victim process's shared libraries by brute force search, but it has the disadvantage of crashing the victim process once for each incorrect guess. This method is acceptable against a web server process, which automatically replaces the crashed request handling child processes with forked copies with identical address space layouts; but it is not suitable for attacking a user-facing process such as a web browser.

JIT spraying brings together code injection and code reuse into a hybrid attack that defeats both DEP and ASLR with a much lower probability of crashing the victim process than [47]. First introduced by Blazakis [13], JIT spraying makes use of the Just-In-Time compilers (a.k.a. JIT compilers, or simply JITs) found in the implementations of many languages that are not compiled to native code ahead of time. JIT compilers allow these language implementations to enjoy vast performance improvements over interpretation by emitting native code at runtime. They are deployed widely and can be found in the JavaScript engines of nearly all mainstream web browsers. JIT spraying abuses the observation that when a JIT compiler compiles code in a high level language down to native code, 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 address space of the language runtime and any program with which it shares its address space.

Blazakis' JIT spraying attack targeted the ActionScript (Flash) JIT on x86. In the attack, the adversary encodes a NOP sled and shellcode in the JIT code produced by a seemingly-innocuous sequence of bitwise XOR operations resembling the following:

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

When compiling the above statement, the ActionScript JIT compiler produces the bytes shown in dashed boxes in Figure 1.1, 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

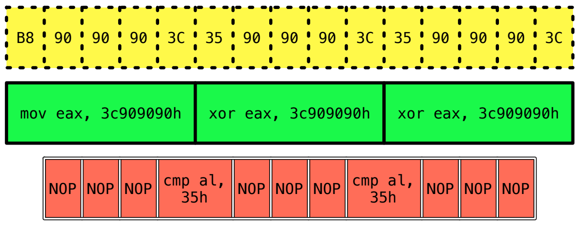


Figure 1.1. Illustration of a NOP sled encoded in the bytes implementing the statement `x = 0x3c909090 ^0x3c909090 ^0x3c909090;`

in the bottom row of Figure 1.1. This alternate decoding contains one-byte x86 NOP instructions (`0x90`) and compare instructions that are semantic NOPs so long as the state of the processor flags need not be preserved. Therefore, execution that begins at 4 out of 5 byte offsets in the JIT code will execute a NOP sled rather than the intended instruction stream. The NOP sled can be extended in length 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 adversary would extend the NOP sled by XORing more `0x3c909090` constants and eventually encode shellcode instructions up to three bytes in length in place of the `0x90` (NOP) bytes. The XOR chain statement could be placed in a function that is repeatedly declared and invoked 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.

After Blazakis brought JIT spraying into the public eye, Sintsov explored the art of writing ActionScript JIT spray payloads for the x86 in greater depth by demonstrating the construction of a stage-0 JIT spray shellcode that locates and calls `VirtualProtect` to enable executability on a shellcode buffer [49]. JIT spraying has also been extended beyond the ActionScript JIT to other x86 JITs. Sintsov demonstrated [48] a JIT spraying attack against the JavaScriptCore Baseline JIT on x86. Rohlf and Ivnitskiy pointed out the presence of attacker-provided constants in JIT code emitted by Mozilla's JaegerMonkey and TraceMonkey JavaScript engines for x86; they also introduced the idea of ROP gaJITs, short instruction sequences ending in a return that can be sprayed multiple times into memory and cobbled together into a ROP attack [44]. In 2011, Beck [12] presented findings that the Tamarin ActionScript JIT for ARM can be abused to inject attacker-provided code verbatim via constant pools, which are regions of data that are placed inline with JIT code and contain constant operands too large to encode as immediate operands within an instruction.

Chapter 1, in part, is a reprint of the material as it appears in *Proceedings of the 2015 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, has been submitted for publication of the material as it may appear in appear in *Proceedings of the 2017 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Assumptions and Threat model

In this chapter, we lay out the assumptions we make with regard to the capabilities possessed by an attacker as well as the defense mechanisms in place on systems under attack. The most important capability that we assume is that the attacker is able to leverage a control flow vulnerability in the process under attack on-demand to divert control flow to an address of her choosing. Despite defensive efforts to preserve the integrity of control flow (e.g., stack canaries [1, 28, 20], CFI [3], etc.), control flow vulnerabilities continue to plague modern systems. Therefore, we consider this assumption to be within reason.

We assume that the process under attack is protected by security mechanisms such as W $\oplus$ X and fine-grained ASLR, which enable it to resist traditional code injection and code reuse attacks such as stack smashing [4] and return-oriented programming [46].

We grant the adversary no further capabilities beyond a control flow vulnerability and access to a scripting environment which performs JIT compilation of attacker-provided code on the victim's machine (e.g., a JavaScript engine in a web browser). In particular, we do not assume that the attacker is able to write arbitrary memory locations in the victim process's address space. We argue that an arbitrary memory write capability obviates malicious reuse of JIT code and merely amounts to abuse of the readable-writable-executable memory protections often assigned to memory regions allocated for

JIT code, and while such a capability would pose a severe threat, it ignores the threat of attacker influence on the instructions emitted by JIT compilers.

Furthermore, we assume that the attacker is operating “blindly” without the aid of a vulnerability allowing her to read arbitrary memory locations. We believe that the threat model in which an attacker does not possess a memory disclosure vulnerability is reasonable. The majority of attacks in the literature that maliciously reuse JIT code can be performed blindly. Furthermore, if we are able to reach a state where deployed JIT spraying defenses completely mitigate blind JIT spraying attacks, we will have raised the bar on malicious reuse of JIT code to the extent that control flow vulnerabilities will not be useful against JIT code unless accompanied by other vulnerabilities.

Chapter 3

ARM Architecture

The ARM architecture is a reduced instruction set computer (RISC) architecture that has enjoyed widespread deployment in computing environments where low power draw is important such as smartphones, tablets, and laptops. ARM Holdings' 2014 Strategic Report [8] proclaims a dominating 95% ARM market share in mobile handsets and an 86% market share in the broader category of smartphones, tablets, and laptops. In the global processor market, ARM is a growing presence; ARM chip production increased by 20% from 2013 to 2014, making up 37% of all chips manufactured in the latter year.

In this dissertation, we will focus on ARM chips implementing the ARMv7-A architecture, which supports a 32-bit address space with 32-bit arithmetic. Chips implementing the newer 64-bit ARMv8-A architecture, introduced in 2011, are beginning to appear but are out of the scope of this dissertation.

3.1 Instruction sets

Prior to ARMv4T, the ARM architecture supported just one instruction set known simply as "ARM." ARM instructions are stored as fixed-width 4-byte words aligned to 4-byte boundaries. In 1994, ARM Holdings released the ARM7TDMI core implementing the ARMv4T architecture, which introduced the "Thumb" instruction set, composed of 2-byte fixed-width instructions stored as halfwords. Like ARM instructions, Thumb instructions must be aligned, but rather than being 4-byte aligned, Thumb instructions must be 2-byte aligned.

In 2003, ARM introduced the Thumb-2 enhancements to the Thumb instruction set, which added 4-byte instructions (separate from those found in the ARM instruction set) that can be intermixed with 2-byte Thumb instructions. Unlike ARM instructions, which are encoded as 4-byte words, 4-byte Thumb-2 instructions are encoded as two consecutive 2-byte halfwords. Thumb-2 support was introduced in the ARMv6T2 architecture and is mandatory in all cores implementing ARMv7 and above. We use the names *Thumb* and *Thumb-2* interchangeably to refer to the Thumb-2 instruction set containing mixed 2- and 4-byte instructions.

The ARM architecture includes support for two other instruction sets, Jazelle and Thumb Execution Environment (ThumbEE). Jazelle was intended to allow Java bytecode to be executed directly on hardware but is almost never implemented, and ThumbEE has been deprecated. Therefore, both Jazelle and ThumbEE are outside the scope of this dissertation.

Whether a particular ARM core will interpret an instruction stream as ARM, Thumb, ThumbEE, or JVM bytecode is determined by its current execution mode, which is stored in the instruction set state register (`ISETSTATE`). The `ISETSTATE` register can be modified through the use of interworking instructions. Since the ThumbEE and Jazelle execution modes are rarely used, we constrain our discussion of instruction set interworking to ARM-Thumb interworking.

ARM processors allow ARM code to call into and return from Thumb code and vice versa. The current execution mode can be changed via a handful of branch instructions that always toggle the instruction set and interworking branch instructions that use the least significant bit of the branch target address as a signal for whether to execute in ARM or Thumb mode. When branching to an address using an interworking branch instruction, the processor inspects the least significant bit of the branch target address. If it is set, the processor clears the least significant bit and branches execution to the resulting address in Thumb mode; if it is not set, the processor branches execution to the target address in ARM mode. This clever use of the least significant bit is made possible by the fact that ARM and Thumb instructions are aligned to 4- and 2-byte boundaries, respectively. Consequently, the least significant bit of every instruction's address is never needed to identify the branch target and is free to be repurposed for interworking.

3.2 Core registers

The ARM architecture has 13 32-bit general purpose registers (`R0`-`R12`) and three 32-bit special-purpose registers (`R13`-`R15`). The usage convention for the general purpose core registers is defined by the Procedure Call Standard for the ARM Architecture (AAPCS) [7] and is summarized in Table 3.1.

The special-purpose registers have roles defined by the instruction set and implemented in hardware. `R13` serves as the stack pointer register (`SP`). Special variants of the `add` and `sub` instructions are hardwired to use `SP` as an operand.

The link register (`LR`/`R14`) is used to hold subroutine return addresses. The ARM analogs of x86's `call` instruction are the branch with link (`bl`) and branch with link and exchange (`blx`) instructions. When either of these instructions is executed, it not only causes execution to branch to the provided branch target, but also saves the address of the instruction following the branch instruction into `LR`. To support ARM-Thumb interworking, the saved return address has its least significant bit set if and only if the branch was executed in Thumb mode. Whether the callee is ARM code or Thumb code, it will be able to return to its caller in the proper execution mode because the return

Table 3.1. ARM general-purpose registers

Register	Argument	Return value	Scratch	Local Var.	Platform-specific
R0	✓	✓	✓		
R1	✓	✓	✓		
R2	✓	✓	✓		
R3	✓	✓	✓		
R4				✓	
R5				✓	
R6				✓	
R7				✓	
R8				✓	
R9					✓
R10				✓	
R11				✓	
R12			✓		

address's least significant bit encodes the return mode. If the callee makes any subroutine calls of its own, it must save `LR` before it gets overwritten by the call. For this reason, it is common practice to store all callee-saved registers along with `LR` onto the call stack at the beginning of each function and restore them prior to returning.

The program counter register (`PC`/`R15`) holds the address of the instruction currently being executed plus 8 while in ARM mode or the address of the currently-executing instruction plus 4 while in Thumb mode. Most data processing and memory instructions can write their results into the `PC`. The `PC` overwrite has the effect of branching to the address written to the register and, in certain circumstances, can cause the processor to switch from ARM to Thumb mode or vice versa. A common convention at subroutine return sites is to restore the callee-saved registers from the stack, and then to restore the saved `LR` value (which held the return address) directly into the `PC`, effectively causing the subroutine to return to its caller.

3.3 Endianness

ARM is a bi-endian architecture, meaning that it can interpret words and half-words as either big or little endian. The `ENDIANSTATE` execution state register stores a bit determining data memory endianness, and the ARM ISA provides the `setend` instruction to modify its value. Prior to ARMv7, ARM supported both big and little endian instruction memory, but big endian instruction support was dropped in ARMv7. The endianness of data memory can still be toggled.

3.4 Conditional execution

Unlike the x86 instruction set, which only supports conditional execution of branch instructions, most ARM instructions can be predicated through a 4-bit condition code. This allows for the construction of some `if-then-else` blocks without the use of a branch instruction. The condition code field is located in the most significant nibble of the ARM instruction and comprises three bits which specify a positive condition (e.g., negative result, carry bit set, overflow) and a fourth bit which allows for negation of the condition (e.g., non-negative result, carry bit clear, no overflow). The condition code 1110_2 is the Always (AL) condition code and cannot be negated; 1111_2 is an illegal condition code that will cause a processor exception if encountered. Some ARM instructions do not treat the most significant nibble as a condition code. These instructions may only be executed unconditionally, and their most significant nibble must be 1111_2. They are composed of coprocessor instructions, SIMD instructions, hint instructions, and an unconditional PC-relative branch with a forced instruction set mode change to Thumb mode.

Since the Thumb-2 instruction set is designed to improve code density, most of its instructions do not contain a condition code field. The only exception is the PC-relative branch instruction. Conditional execution in Thumb mode can also be achieved through the “compare and branch on zero/non-zero” (`cbz`/`cbnz`) instruction, which compares a register’s value against zero and conditionally performs a PC-relative branch based on the outcome. Thumb-2 also provides the ability to create a short conditionally-executed block of code without a branch instruction via the “if-then” (`it`) instruction. Up to four Thumb-2 instructions following an `it` instruction are conditionally executed based on a condition code provided in the `it` instruction.

Chapter 3, in part, is a reprint of the material as it appears in *Proceedings of the 2015 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 4

JIT Spraying Payloads on ARM

4.1 Introduction

There is a pattern in security research whereby new attacks are initially designed to target the x86 platform. This is not without good reason. The x86 is undoubtedly the most prevalent architecture on the market. Eventually, however, researchers discover that the architectural features that were thought to be lynchpins of an attack are in fact merely implementation details. For example, Shacham's seminal work on return-oriented programming [46] was thought to hinge on specific properties of the x86 architecture such as its variable-length, unaligned instructions and small register file. However, since then, there has been an explosion of work extending ROP to architectures very different from the x86 such as SPARC [15], ARM [34], and the Zilog Z80 [17].

In similar fashion, the x86 architecture lends itself particularly well to JIT spraying. In particular, the x86's variable-length, byte-aligned instructions and support for 32-bit immediate operands are conducive to hiding malicious instructions within benign JIT-compiled instructions. The absence of these features in other architectures greatly complicates the construction of a JIT spraying payload. In this chapter and the three that follow it, we continue the tradition of extending attacks from the x86 to new architectures by demonstrating JIT spraying on the ARM architecture. We begin by examining the attacker's options with regard to encoding a JIT spraying payload on the ARM architecture.

After discussing vectors through which an attacker can exert influence over JIT code memory, we consider three types of JIT spraying payloads that one might try to construct on ARM. They are same-instruction set self-sustaining payloads, cross-instruction set self-sustaining payloads, and gadget chaining payloads. Of these three types, we found that cross-instruction set self-sustaining payloads and gadget chaining payloads are the most feasible payloads for blind JIT spraying on ARM. Specifically, they allow us to make the following contributions:

1. We show for the first time that RISC architectures are not immune to instruction decoding ambiguity in JIT-compiled code demonstrate a payload that hides unintended Thumb-2 instructions among intended Thumb-2 instructions. In Chapter 5, we use this payload in a proof of concept blind JIT spraying attack in conjunction with a new model we discovered for JIT spraying called *gadget chaining*, wherein a high level language is able to treat short snippets of unsafe computation called *gadgets* as callable primitives.

2. We demonstrate that JIT spraying attacks are not dependent on abusing instruction decoding ambiguity (i.e., unintended instructions). We present a proof of concept attack against Chrome's V8 JavaScript engine on ARM which uses gadget chaning and relies only on intended ARM instructions to bootstrap arbitrary malicious code execution (Chapter 6).

3. In Chapter 7, we present a proof of concept JIT spraying payload against Mozilla's SpiderMonkey JavaScript engine on ARM which takes advantage of ARM-Thumb interworking to hide an unintended Thumb-2 instruction stream that does not resynchronize to the intended instruction stream among intended ARM instructions.

Taken together, the proof of concept attacks against JavaScriptCore, V8, and SpiderMonkey on ARM cover major browsers on both the Android and iOS smartphone platforms and show that almost 2 billion computers are vulnerable to JIT spraying attacks.

4.2 Controlling JIT compiler output

The ease with which an attacker can devise a high level language input to a JIT compiler that results in the emission of JIT instructions that can be reused for malicious computation (i.e., the JIT spraying payload) is a function of both the JIT compiler's code generation practices and the target architecture. The JIT compiler's practices with respect to register allocation policy, call/return conventions, management of runtime type information, instruction selection, and code layout dictate the set of useful JIT code emissions that the attacker will be able to induce and the organization thereof. Architectural design choices such as instruction encoding layout, instruction length, and instruction alignment decide what, if any, ambiguous instruction decoding is possible.

In this section, we focus our attention primarily on features of the ARM architecture that give the attacker control over JIT code and allow her to create malicious code from the encodings of benign code. For the sake of concreteness, our discussion makes reference to the specific code emission practices of certain JITs when necessary, but our intention is to highlight concepts may be generalized to other JIT compilers.

4.2.1 Attacker-controlled bits

One of the attacker's primary goals when designing a JIT spraying payload is maximizing the percentage of attacker-controlled bits in JIT-emitted code. This is especially important when the JIT spraying attack relies on improper disassembly of JIT code. We define an attacker-controlled bit as a bit in an instruction's immediate or register field whose value an attacker can predictably manipulate by varying her input to

Table 4.1. ARM and Thumb immediate encoding limits

	Immediate bits	Immediate bits (arithmetic)
ARM	25	16
Thumb-2	24	16

the JIT.

Maximizing attacker-controlled bits involves both maximizing the concentration of attacker-controlled bits in instructions that do contain them as well as minimizing the emission of instructions bytes that do not contain attacker-controlled bits. Blazakis' original JIT spray payload against x86 is a prime example of JIT code with high attacker-controlled bit concentration. It contains numerous back-to-back sequences of one non-attacker- controlled byte (the instruction opcode)[1] followed by four attacker-controlled bytes (an immediate operand).

4.2.2 Immediate bits

Unfortunately, ARM and Thumb instruction encodings cannot contain as many immediate bits as x86 instructions. In contrast with the 32-bit immediates that can be encoded in the x86 bitwise XOR instruction, both the ARM and Thumb instruction sets' immediate-operand bitwise operation instructions (AND, OR, and exclusive OR) only dedicate 12 bits to the immediate operand. Compiling the bitwise XOR of a 32-bit immediate on ARM is usually handled by loading the constant piecewise into a register 16 bits at a time, then computing the bitwise XOR over the register operand. Table 4.1 lists the maximum number of immediate bits that can be encoded in any one ARM or Thumb instruction as well as in only arithmetic/bitwise instructions, whose immediate

[1]The attacker can manipulate the opcode to a small degree by choosing different operations and gaming the register allocator, but this control is relatively constrained compared to direct control over the immediate operand.

bits are more readily and precisely influenced by the JIT's input than other instructions such as load/store or branch instructions.

As mentioned above, there are instructions for moving a 16-bit immediate half of a register; these instructions exist in both the ARM and Thumb instruction sets and are usually emitted as a pair. The `movw` instruction moves a halfword into the bottom half of a register and clears the top half; the `movt` instruction moves a h alfword into the top half of a register while preserving the bottom half. Since `movw`/`movt` pairs are simply compensating for the lack of 32-bit immediate operands on the ARM architecture, we have never observed them emitted in long back-to-back chains; they are always punctuated with instructions that operate on the register being populated.

Aside from the immediates in arithmetic/bitwise instructions, an attacker could control the immediate bytes in a PC-relative branch by creating branches of varying sizes. Maisuradze et al. [36] demonstrate an application of this control vector by using the offsets in PC-relative calls and branches in x86 JIT code to encode ROP gadgets whose locations are revealed with the help of a memory disclosure vulnerability. If control flow between unintended instructions encoded by branch offsets can be orchestrated with PC-relative branches, a non-ROP attack could be launched. To conserve space when a payload makes use of multiple branch offsets, branches can be nested by creating a common convergence point for all of the branches. Such an optimization is crucial for reusing branch instructions with large branch offset fields. For example, the ARM instruction set's PC-relative branch contains a 24-bit branch offset, allowing it to skip up to 64MB of code (2^24 4-byte ARM instructions). Listing 4.1 shows how an attacker might use mutually-exclusive `if` blocks to generate branch instructions with strictly decreasing immediate offsets. A drawback to to this method is that not all branch offsets will be available due to the auxiliary code that must be produced between each branch instruction. Second, since the branch distances are organized in strictly decreasing

```
if (condition_1)
  ;
else if (condition_2)
  ;
else if (condition_3)
  ;
...
// All ‘then’ blocks converge here.
```

Listing 4.1. Mutually-exclusive conditionals enable an attacker to spray PC-relative branch instructions with varying immediate offsets.

order, it may be non-trivial to execute consecutive unintended instructions encoded by increasing branch distances, as this would require branching backwards. Rather than branching backwards, one could create multiple sets of mutually-exclusive `if` blocks and optimize the ordering and choice of unintended instructions to minimize the number of such blocks that need to be compiled.

An attacker might also attempt to influence the immediate offset field in a load or store instruction by tuning the parameter, variable, or object property access patterns in her high level language code. This is possible because many values are stored sequentially in memory in a predictable order. For example, SpiderMonkey's non-optimizing JIT places formal function arguments in an array 28 bytes above the location pointed to by its frame pointer register. An attacker can therefore predict that, since SpiderMonkey stores JavaScript values as 64-bit NaN-boxes, read accesses of the first argument will result in a load instruction whose immediate offset is 28; accesses of the second argument will have an immediate offset of 36; and so on.

4.2.3 Register fields

In addition to immediate bits, an attacker can control the choice of operand and destination registers used in certain instructions by carefully crafting her input to the JIT compiler. The JavaScriptCore, V8, and SpiderMonkey JavaScript engines employ

```
function (R0, R2, R8, R10) {
  var R1 = R0 ^ 0x1234;
  var R4 = R2 ^ 0x2345;
  var R9 = R8 ^ 0x3456;
  var R11 = R10 ^ 0x4567;
  // At this point all registers have been populated
  return (R1^R2) | (R4^R8) | (R10^R9) | (R11^R0);
}
```

Listing 4.2. Variables in this function are named for the registers storing the variables' values at the point in execution just prior to the return statement.

```
4051            eors    r1, r2
ea84 0408       eor.w   r4, r4, r8
4321            orrs    r1, r4
ea8a 0a09       eor.w   r10, r10, r9
ea41 010a       orr.w   r1, r1, r10
ea8b 0b00       eor.w   r11, r11, r0
ea41 010b       orr.w   r1, r1, r11
```

Listing 4.3. Raw bytes and disassembly of the computation of the return value in Listing 4.2, as generated by JavaScriptCore's optimizing JIT compiler.

multiple levels of JIT code optimization; and at the lowest level, where a register allocator is not used, the registers chosen for a particular high level operation are always the same. Values are conveyed from one high level operation to the next via the stack. JIT code at the lowest level of optimization is therefore immune to this form of manipulation. All three of the aforementioned JavaScript engines do however have an optimization level which performs register allocation, enabling an attacker to predict the registers which will hold the values corresponding to high level language variables. The parameters and variables in the anonymous function shown in Listing 4.2 are named to correspond with the registers into which their values will be stored. The function body's first four lines cause its arguments to be loaded into registers in a predictable order. The result of XORing each argument with an immediate is stored into other registers, again in predictable order. At this point, the attacker can predict which registers will be used in combination with one another and in what manner, as the return statement demonstrates.

Listing 4.3 shows the raw bytes and disassembly of the return value computation generated by JavaScriptCore's optimizing JIT compiler. Notice how the attacker has complete control over which registers are XORed together: `R1` with `R2`, `R4` with `R8`, etc. Furthermore, the attacker has control over which registers are chosen as accumulator registers based on the order of the operands. The result of (`R4 ^ R8`) is stored into `R4` because `R4` is the left-hand operand of the XOR operator. Likewise, the result of (`R10 ^ R9`) is saved into `R10`, which was written to the left. The capability for the attacker to control the accumulator register varies by JIT compiler; if similar code were compiled by SpiderMonkey's optimizing JIT, the lower-numbered register would always be chosen as the accumulator.

Influence over what registers are used as operands can translate to influence over the lengths of instructions if the JIT emits Thumb-2 instructions. Register fields in most 16-bit Thumb instructions are only three bits wide, restricting them to accessing only the lower 8 registers. Using `R8`–`R15` as an operand usually requires the use of a 32-bit instruction. This can be observed in the first two lines of Listing 4.3 which shows that the JIT chooses to use the 16-bit XOR encoding to XOR `R1` with `R2`, but it must use the 32-bit encoding to XOR `R4` with `R8` since the 16-bit encoding only provides 3-bit register fields which cannot encode `R8`.

4.2.4 Arithmetic woes

If an attacker opts to chain together arithmetic operations such as addition, subtraction, multiplication, and division rather than bitwise operations in an attempt to induce the JIT to emit densely-packed instructions with attacker-controlled immediates a la Blazakis' original payload [13], she will most likely discover that the arithmetic instructions will not be emitted back-to-back. This is because it is common for JIT compilers to inject runtime checks after these operations to test for overflows, underflows, and other exceptional conditions. These checks introduce several consecutive instructions that do not contain attacker-controlled bytes and should therefore be avoided. For this reason, we consider bitwise operations to be the best method for generating long runs of tightly-packed instructions with a relatively high concentration of attacker-controlled bits.

As an aside, runtime checks are necessary for a JavaScript JIT because although JavaScript numbers are all technically floating point numbers, optimized code can be generated that assumes (based on observations of prior values) that certain numbers are 32-bit integers. This enables optimized JIT code to take advantage of native integer arithmetic, which can be faster than its floating point counterpart. However, if the result of 32-bit integer arithmetic were to under- or overflow, the runtime would need to intervene to convert the value to a floating point representation, which would be large enough to hold the result. JIT code generated by a non-optimizing JIT compiler would not necessarily contain these runtime checks, but none of the three non-optimizing JavaScript compilers we studied are useful for generating instructions with densely-packed attacker-controlled bits because they produce "canned" instruction sequences for high-level operations and do not incorporate untrusted constants into the sequences for bitwise and arithmetic operations. Therefore, it is only necessary to consider the usefulness of arithmetic instructions emitted in optimized JIT code.

4.3 Same-instruction set self-sustaining payloads on ARM

In this subsection, we analyze the prospect of same-instruction set self-sustaining JIT spraying payloads on ARM. This is the type of payload used in Blazakis' [13] original JIT spraying attack. It is "self-sustaining" because once the payload is branched to via a control flow vulnerability, an arbitrary number of unintended instructions can be executed without resynchronization to the intended instruction stream. The notion of a self-sustaining JIT spraying payload therefore only pertains to payloads which abuse instruction decoding ambiguity to create unintended instructions from intended JIT-emitted instructions. A self-sustaining payload is designated as being “same-instruction set” when the intended instruction stream and unintended instruction streams are executed under the same instruction set mode.

On the ARM architecture, same-instruction set decoding ambiguity can be brought about by endianness confusion or execution from an unintended instruction boundary. Endianness confusion can be created by toggling the processor’s endian state to cause it to decode JIT code with a reversed byte order from the one under which it was generated. However, since recent (ARMv7 and later) ARM chips only support little endian instruction memory, we omit endianness confusion from the scope of our work, and it remains an open problem.

Same-instruction set execution of JIT code at unintended instruction boundaries on ARM is limited to Thumb-mode execution. Specifically, the CPU under attack must support the extended Thumb-2 instruction set. The reason for this is quite self-evident. In ARM mode, the processor ensures that instruction fetching and decoding occurs along 4-byte aligned boundaries; it is simply impossible to divert control flow into the middle of an ARM instruction. Likewise, on a processor that lacks support for 32-bit Thumb-2 instructions, all 2-byte aligned branch targets are intended instructions. Thumb-2, however, allows for ambiguity in the decoding of instruction memory. While it is still necessary to fetch and decode instructions from 2-byte aligned boundaries, the mixing of 16- and 32-bit Thumb-2 instructions allows for the second halfword of a 32-bit Thumb-2 instruction to be interpreted as the first halfword of an unintended instruction.

In order for the unintended instruction stream to avoid resynchronizing with the intended instruction stream, all unintended instructions must be 32-bit Thumb-2

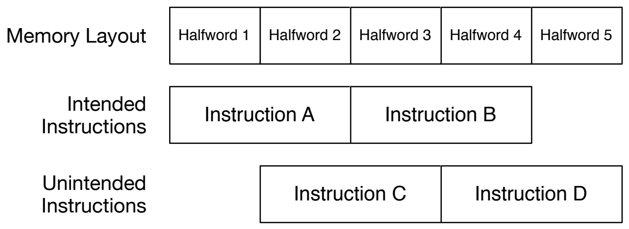


Figure 4.1. Example of two possible Thumb-mode decodings of a sequence of halfwords.

instructions, and the intended instructions encoding them must also be 32-bit Thumb-2 instructions. In other words, the attacker's goal is to execute a chain of intended 32-bit Thumb-2 instructions out of phase by one halfword, as illustrated in Figure 4.1. Notice that if any of the intended or unintended instructions were shorter than 32-bits wide, the instruction stream would resynchronize. For example, if intended Instruction B were a 16-bit instruction, unintended execution would resynchronize to the intended instruction beginning with Halfword 4 immediately after executing unintended Instruction C. Similarly, if unintended Instruction C were a 16-bit instruction, execution would resynchronize immediately after it, resuming the intended instruction stream at Instruction B.

Inducing a JIT compiler to generate a chain of intended 32-bit Thumb instructions that encode a out-of-phase chain of 32-bit Thumb instructions is challenging. The intended instructions used must have the property that their second halfword is a valid first halfword for a 32-bit Thumb instruction. All 32-bit Thumb-2 instructions must begin with either the 1111_2 or 11101_2 bit pattern, but the second halfword of all 32-bit Thumb encodings of the bitwise AND, OR, and XOR operations begin with a 0-bit. Therefore the intended instruction stream cannot comprise chained bitwise operations. Moreover, we are not aware of any 32-bit Thumb-2 instructions that are both easily controlled and generated back-to-back using SpiderMonkey, JavaScriptCore, or V8 and encoded using a second halfword that can be interpreted as the first halfword of a 32-bit Thumb-2 instruction.

Thus, we conclude that same-instruction set self-sustaining payloads are infeasible on ARM. At best, we can execute a single unintended 16-bit Thumb instruction before execution resynchronizes to the intended instruction stream. In Section 4.5, we describe how an attacker can exploit this limited decoding ambiguity to build a gadget chaining payload.

4.4 Cross-instruction set self-sustaining payloads

A cross-instruction set self-sustaining payload takes advantage of the ARM architecture's support for interworking branches. An obvious consequence of the interworking feature is that corrupted control data (e.g., function pointers, saved return addresses, etc.) can cause the processor to execute Thumb code in ARM mode or vice versa. This capability could be exploited by crafting an input to a JIT compiler that leads to the emission of code that, when executed in an instruction set mode not intended by the JIT compiler, performs malicious computation. Such a payload can avoid resynchronization to the intended instruction stream as long as the unintended instruction stream does not perform an interworking branch back to the intended instruction set mode at an intended instruction boundary. In Sections 4.4.1 and 4.4.2, we analyze the feasibility of creating useful Thumb-to-ARM self-sustaining payloads and ARM-to-Thumb self-sustaining payloads, respectively.

4.4.1 Thumb-to-ARM self-sustaining payloads

Encoding a Thumb-to-ARM self-sustaining payload poses a perplexing practical problem of nightmarish complexity due to the fact that intended Thumb-2 instructions are at most as long as the unintended ARM instructions that they are used to encode. Consequently, no two consecutive halfwords in a Thumb-2 instruction stream can be used to encode an unintended ARM instruction comprising exclusively attacker-controlled bits. In other words, every unintended ARM instruction derived from a Thumb-2 instruction stream must contain fixed opcode bits, and since it is easier for an attacker to exert fine control over operand bits than opcode bits in the intended instruction stream, the set of encodable unintended ARM instructions is constrained by the quirks of the Thumb-2 opcode encoding scheme. To make matters more complicated, each of the two consecutive Thumb-2 halfwords that are used to build an ARM instruction could be a 16-bit Thumb instruction, the first halfword of a 32-bit Thumb instruction, or the second halfword of a 32-bit Thumb instruction (with the exception of two combinations—16-bit instruction followed by second halfword of 32-bit instruction and first halfword of 32-bit instruction followed by 16-bit instruction). In this subsection, we scratch the surface of this payload type; truly understanding its feasibility requires exhaustive cataloguing of Thumb instructions that can be generated consecutively with respect to a specific JIT compiler.

In particular, we study the consequences of using each of the three types of Thumb halfword to encode the most significant half of an unintended ARM instruction. The most significant half of the ARM instruction is important because most of the instruction's opcode is located therein. We will pay particular attention to two fields found in ARM instructions that complicate the task of encoding useful ARM instructions with intended Thumb-2 instructions. They are the condition code and the result destination register (denoted `Rd`) for ALU instructions (e.g., `mov`, `and`, `sub`). Figure 4.2 illustrates the positions of these fields in ARM instructions and their corresponding positions in a Thumb instruction stream. Notice that the most significant 4 bits of every consecutive Thumb halfword will correspond to either the condition code or `Rd` (if the unintended instruction contains the `Rd` field). Below we will see the impact of these two fields on building ARM instructions out of Thumb-2 instructions. We structure our discussion by

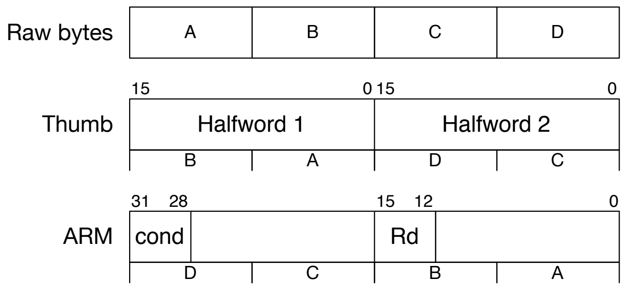


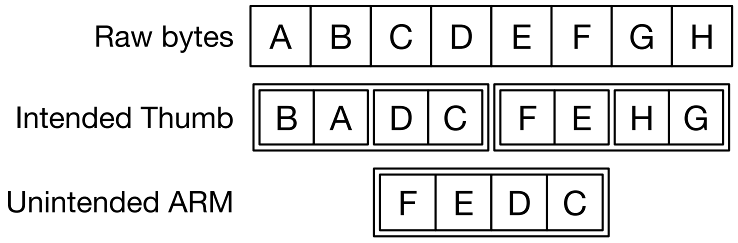
Figure 4.2. Example decoding of four consecutive bytes of little endian instruction memory into two Thumb halfwords and an ARM instruction with both a condition flag and an ALU destination register. Bytes are denoted A-D. Note the correspondence between the first 4 bits of each Thumb halfword with the condition code and `Rd`.

referring to the three types of Thumb-2 halfwords used to encode an ARM instruction's most significant half by the following three class numbers:

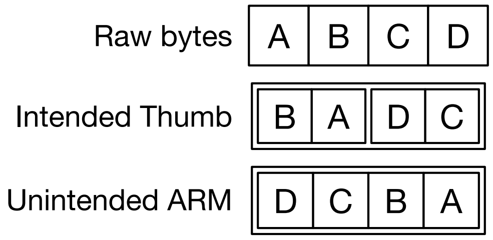
1. The first halfword of a 32-bit Thumb instruction (Figure 4.3a)
2. The second halfword of a 32-bit Thumb instruction (Figure 4.3b)
3. The sole halfword of a 16-bit Thumb instruction (Figure 4.3c)

Class 1

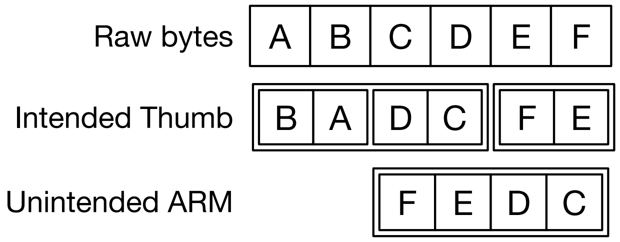
We first look at using the first halfwords of 32-bit bitwise instructions—which, as we described in Section 4.2, are the easiest instructions to generate back-to-back and control—as the most significant half of an unintended ARM instruction. Thumb's 32-bit register-operand bitwise instructions can only be used to from PC-relative branches because they all begin with the bit pattern 1110101_2, which is the prefix for ARM's various PC-relative branch instructions. The immediate-operand variants of the bitwise operations offer little more than their register-operand counterparts; they can only encode variants of the `vst` instruction, which store the contents of floating point registers into memory.



(a) Class 1: First halfword of 32-bit Thumb instruction



(b) Class 2: Second halfword of 32-bit Thumb instruction



(c) Class 3: Sole halfword of 16-bit Thumb instruction

Figure 4.3. Illustrations of the three classes of halfwords from which unintended ARM instructions can draw their most significant half.

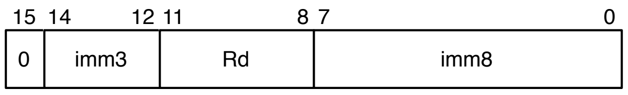


Figure 4.4. Diagram of the second halfword encoding found in many Thumb ALU instructions with an immediate operand.

The first halfword of Thumb-2 load and store instructions, whose immediate offsets can be influenced by an attacker, do not encode valid ARM instructions. The `movw` and `movt` instructions can only be used to encode various SIMD instructions, and the 32-bit Thumb PC-relative branch instruction can only encode memory barriers, SIMD instructions, hint instructions, and a handful of coprocessor instructions. In general, it is difficult to use class 1 halfwords to encode a broad array of ARM instructions because the first halfword of a 32-bit Thumb instruction is composed primarily of fixed opcode bits rather than register or immediate operands. The condition code field, which occupies the most significant nibble of most ARM instructions, further hinders unintended ARM instruction diversity since variations in the most significant nibble of the Thumb-2 instruction will only affect it and not the ARM opcode.

Class 2

The second class of Thumb halfwords can generate a broader range of unintended ARM instructions since the second Thumb halfword, which typically contains more attacker-controlled bits than the first halfword, will coincide with the most significant bits of the unintended ARM instruction, where the fixed opcode bits usually reside. Figure 4.4 illustrates the typical layout of the second halfword in a 32-bit Thumb ALU instruction with an immediate operand, which contains 11 more-easily controlled immediate bits and a 4-bit register field. The attacker cannot create the Always condition code with this class using any of the easily-controlled instructions described in Section 4.2 (except in the unlikely case that the JIT under attack uses the link register as an allocatable general purpose register for live values, in which case the attacker could use a load or store instruction). Without the Always condition code, the attacker must duplicate each unintended instruction with complementary condition codes in order to ensure that it executes. This is not a major setback, however, because the encodings of most of Thumb's immediate-operand ALU instructions allow the attacker to control the bits that correspond to the last three bits of the unintended instruction's condition code. In fact, this may be a boon to the attacker, as it allows her to write shellcode instructions that can be conditionally executed.

The second class of halfwords does, however, suffer from a drawback stemming from the the destination register (`Rd`) field found in nearly all ARM instructions that produce an ALU result. As shown in Figure 4.2, if the second halfword of a Thumb instruction is decoded as the most significant half of an ARM instruction, the first four bits of the first halfword of that same instruction will be decoded as the `Rd` field (for ARM instructions that have one, of course). Since the Thumb instruction is a 32-bit instruction, we know that its first four bits must be either 1110_2 or 1111_2. Therefore, the unintended ARM instruction's `Rd` field can only take the values `R14/LR` and `R15/PC`. Although the second class provides yet another way for an attacker to encode a branch instruction (since writing to the `PC` results in a branch to the address written), it is difficult to perform useful computation without being able to store ALU results to registers other than `R14` and `PC`. For example, invoking a system call on the ARM Linux GNU EABI requires placing the system call number into `R7` and its arguments into `R0-R6`.

Some ARM instructions do not expect the `Rd` field in the location shown in Figure 4.2 and do not conflict with 32-bit Thumb instruction prefixes (1111_2 and 11101_2). They comprise the following types of instructions:

- multiplication

- Advanced SIMD & floating point (VFP)
- hint instructions
- memory barriers
- coprocessor instructions
- branches
- system instructions (e.g., system call)
- miscellaneous addition & subtraction
- memory loads & stores

It may be possible to extend the number of registers available to unintended instructions encoded with second class halfwords by exploiting Advanced SIMD and VFP instructions and the "extension registers"—a set of registers separate from the core registers—on which they operate. The high-level idea is to populate `R14/LR` with available standard ALU instructions and use the `vmov` instruction (which copies values between from core registers to the extension registers and vice versa) to shuttle values into and out of various extension registers via `R14/LR` and to use Advanced SIMD/VFP instructions to process those values. Unfortunately, `vmov` cannot be used to copy values into abitrary core registers; the only core registers into which `vmov` is permitted to move values are `R14/LR` and `R15/PC` due to the 32-bit Thumb instruction prefix.

A simple application of this technique is a stack-based virtual machine. Operands are popped into `R14/LR` then moved into extension registers. An Advanced SIMD/VFP instruction performs an operation on them, and the result is moved back into `R14/LR`, from whence it is pushed onto the stack. We did not build this payload; our analysis of the second class of halfwords has been simplified by constraining only the bits that

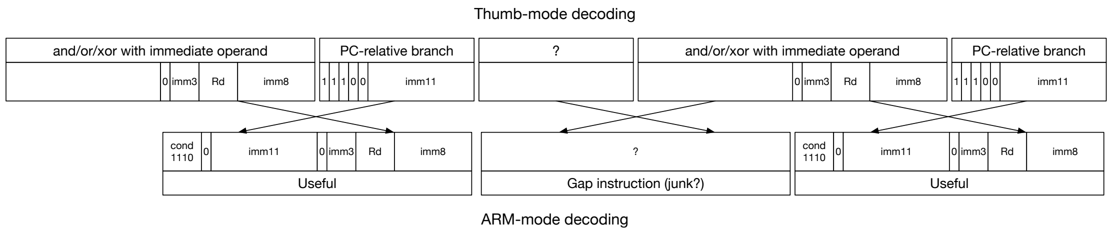


Figure 4.5. Diagram illustrating the use of 16-bit Thumb branch instructions as the most significant half of unintended ARM instructions. Thumb halfwords appear to swap order in the ARM decodings due to the little endian decoding of words in memory.

form the prefixes of all 32-bit Thumb instructions. The actual set of unintended ARM instructions that can be encoded with the second class of halfwords depends on the set of intended Thumb instructions that the attacker is able to induce a specific JIT to emit.

Class 3

The third class of halfwords provides the most potential flexibility for unintended ARM instruction encoding, but it poses the greatest logistical challenge. The only 16-bit Thumb instruction whose prefix can encode the Always condition code is the unconditional PC-relative branch instruction. Figure 4.5 illustrates the structure of a payload using the branch instruction as the first halfword of each useful unintended ARM instruction. By creating large `if-then` blocks as described in Section 4.2, the attacker can enumerate a large portion of possible ARM instruction first halves. By placing an immediate-operand instruction immediately before the branch instruction, the attacker can control most of the bits in the second half of the unintended ARM instruction as well.

A challenge to using the third class of halfwords is filling the gaps between useful unintended ARM instructions. In the example in Figure 4.5, the second half of each useful unintended ARM instruction comes from the second half of a 32-bit Thumb instruction. The first half of each of those 32-bit Thumb instructions forms the first half of an ARM instruction containing very few attacker-controlled bits; these are the instructions in the gaps. Even if the second half of the unintended ARM instructions were to be derived from 16-bit Thumb instructions, it is unlikely that an attacker can induce any JIT compiler to emit an unconditional branch instruction as every other instruction. There are bound to be gap instructions. The attacker must ensure that these gap instructions are semantic NOPs with respect to the useful instructions in her payload.

Using 16-bit Thumb instructions aside from the PC-relative branch provide the benefit of allowing the attacker to conditionally execute unintended ARM instructions, but unconditionally executing ARM instructions using those instructions is much more difficult. This is because no 16-bit Thumb instructions place operands in the first four bits of the instruction. Thus, if the attacker needs to duplicate an ARM instruction with complementary condition codes, she cannot simply duplicate the operation in her high level language code with different operands. She must find two separate 16-bit Thumb instructions that encode the top half of the same ARM instruction, but with complementary fourth bits.

By now, we hope to have convinced the reader that Thumb-to-ARM self-sustaining payloads—while possible in theory—are a nightmare to implement in practice.

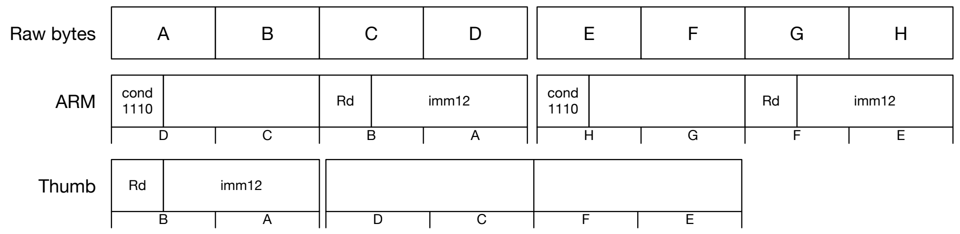
4.4.2 ARM-to-Thumb self-sustaining payloads

At first glance, the task of encoding a useful Thumb-2 payload with ARM instructions appears insurmountable. On one hand, Thumb-mode execution has the advantage of supporting 16-bit instructions, which can be formed from a strict subset of a single intended ARM instruction's bits. Ideally, unintended 16-bit Thumb instructions would be formed from the attacker-controlled-bit-rich least-significant halves of intended ARM instructions. This allows for a wide range of unintended Thumb instructions to be encoded without being constrained by the fixed opcode bits of an ARM instruction.

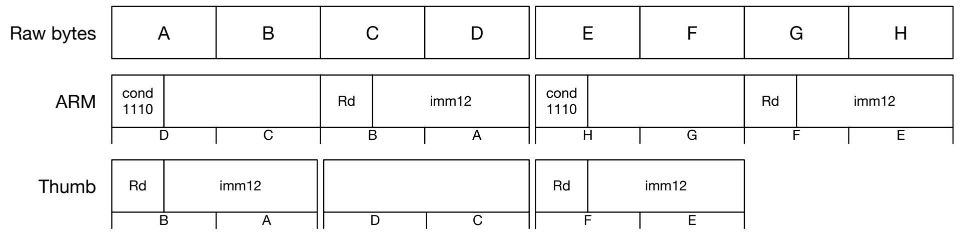
On the other hand, the vast majority of ARM instructions contain the Always condition code, meaning that the Thumb instructions formed from the most significant halves of such ARM instructions must contain the 1110_2 prefix. The set of Thumb instructions with this prefix is limited; the only 16-bit Thumb instruction starting with 1110_2 is an unconditional branch. The 32-bit Thumb instructions that allow that prefix are load/store instructions, register-operand data processing instructions, coprocessor instructions, SIMD, and floating point instructions.

Furthermore, whenever the least-significant half of an intended ARM instruction is used as a 16-bit Thumb instruction, at least one 16-bit Thumb instruction encoded from the most significant half of an intended ARM instruction must be executed before another 16-bit Thumb instruction from the least significant half of an ARM instruction can be executed.

Figure 4.6 illustrates this point. Suppose that the least significant half of each ARM instruction can be influenced by an attacker (halfwords BA and FE); in both subfigures, the attacker has used the least-significant half of the first ARM instruction to encode a useful 16-bit Thumb instruction. Notice in Figure 4.6a that when halfword DC initiates a 32-bit Thumb instruction, halfword FE cannot be a 16-bit Thumb instruction; instead, it must be consumed as the second halfword of a 32-bit Thumb instruction. Extrapolating from this, if halfword HG were to initiate a 32-bit Thumb instruction, the least significant half of the next ARM instruction (not pictured) would be forced to form the second halfword of the 32-bit Thumb instruction. However, if halfword DC were a 16-bit Thumb instruction as shown in Figure 4.6b, halfword FE is free to become a 16-bit Thumb instruction. More generally, once the most significant half of an intended ARM instruction encodes the first half of a 16-bit unintended Thumb instruction, the least significant half of the next ARM instruction is free to encode a useful 16-bit unintended Thumb instruction.



(a) Most significant half of ARM instruction encodes start of 32-bit Thumb instruction



(b) Most significant half of ARM instruction encodes 16-bit Thumb instruction

Figure 4.6. Illustration of ARM-to-Thumb payloads resulting from the most significant half of an ARM instruction (halfword DC) encoding the first halfword of a 32- and 16-bit Thumb instruction. Subsequent least-significant halves cannot encode 16-bit Thumb instructions until after a most-significant half encodes a 16-bit Thumb instruction.

The consequence of the above observation is that if an attacker wishes to encode a 16-bit Thumb instruction using the least-significant half of an ARM instruction, she must eventually execute an unconditional branch instruction. Superficially, it seems that such a payload encoding will waste a large quantity of JIT code, but in the remainder of this subsubsection, we look at a concrete ARM instruction and discuss how it can be used to construct a relatively space-efficient ARM-to-Thumb self-sustaining payload. In particular, we will analyze ARM's immediate-operand bitwise AND instruction, which computes the bitwise AND of a 12-bit immediate value (`imm12`) and the contents of a register (`Rn`) then stores the result into an arbitrary register (`Rd`). By carefully structuring the JavaScript code used to generate the payload, 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 4.7. From top to bottom, the rows show

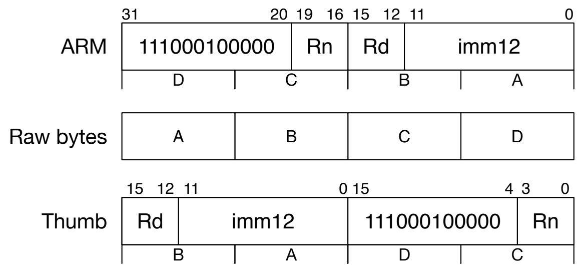


Figure 4.7. Illustration of how the immediate-operand bitwise AND instruction from the ARM instruction set (top row) can be decoded as two 16-bit Thumb-2 instructions (bottom row).

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 ARM JITs are unlikely to to perform the bitwise AND operation with a destination register (`Rd`) of `R14/LR` or `R15/PC`. Therefore, neither byte B nor byte D in this particular instruction will contain this prefix, and 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 immediate-operand data processing instructions are easy to generate and furnish the attacker with the most controllable bits in the instruction's least-significant half; and all such instructions begin with 16-bit Thumb-2 instruction prefixes.

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 prefix, 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 A and B can still encode a broad range of 16-bit Thumb-2 instructions.

The second Thumb-2 instruction must be an unconditional PC-relative forward branch of at least 512 halfwords (this minimum exists because one of the high-order bits in the branch instruction's offset must be set due to the intended ARM instruction's opcode). 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 in each pair performs useful work for the attacker; the second branches to the first Thumb-2 instruction in a subsequent pair. In order for this branch to target the first instruction in a pair, the branch offset must be an odd number of halfwords; therefore `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 $(512+1+2)*2 = 1030$ 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 can designate a general purpose register as a virtual PC which we use to loop execution back into the space skipped by unintended Thumb branches, 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 4.8 shows the virtual PC method with 3 branch blocks under simplified conditions. Note how execution flows through the first 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 proof of concept payload we describe in Section 7.2, 12-bit immediate encoding

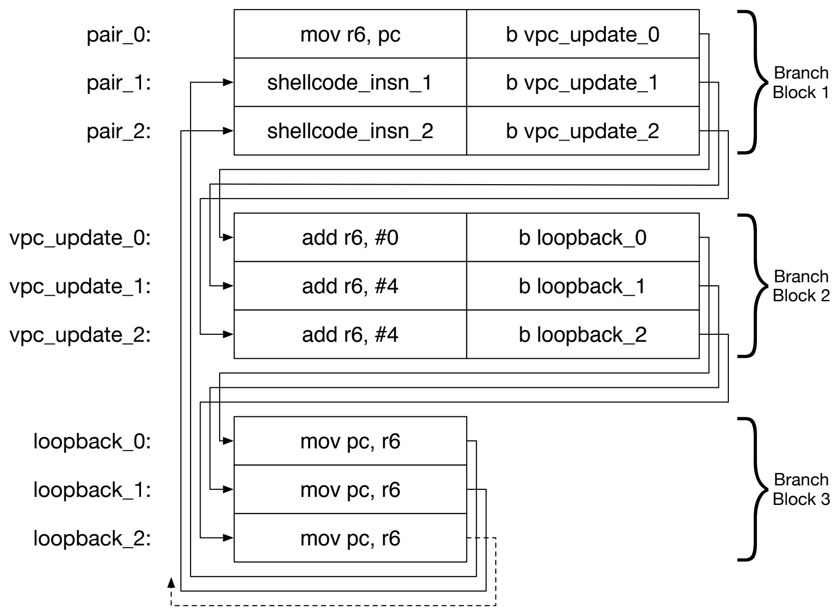


Figure 4.8. Illustration of using a virtual PC (in this case R6) to more efficiently utilize the space skipped over by branches.

rules require us to populate a register with the virtual PC advancement amount and perform register-register addition rather than register-immediate addition to advance the virtual PC. 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 interval of 36 bytes rather than 4 bytes. Since these details are SpiderMonkey-specific, we present them in Section 7.2.

Note that although Figure 4.8 shows only three branch blocks, longer payloads can be encoded by inserting an arbitrary number of 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, due to the high-order bits in the unintended branch instruction’s offset field that correspond to set bits in their opcodes.

4.5 Gadget chaining payloads

In this subsection, we introduce *gadget chaining*, a novel technique for utilizing JIT sprayed payloads. Unlike an attack which diverts control flow to a self-sustaining payload, in which control flow is retained throughout the execution of the entirety of the attacker's malicious computation, a gadget chaining attack uses the high level language already available to the attacker to perform Turing-complete computation and treats short sequences of JIT-sprayed code as callable primitives used to augment the program with capabilities beyond those prescribed by the language's designers.

We refer to these short sequences of JIT sprayed code as "gadgets," and they can be thought of as short subroutines that are called by the high level language (e.g., JavaScript) via a control flow vulnerability. Once execution branches to a gadget, it performs its malicious computation (e.g., storing a value to memory or moving a value into a register) and returns control flow back to the high level language.

We borrow the term "gadget" from return-oriented programming parlance, but gadget chaining's gadgets function quite differently. Whereas the return at the end of a ROP gadget serves to divert control flow to the next gadget, a gadget chaining gadget's return is more akin to a normal function return.

As we observed in Sections 4.3 and 4.4, it can be difficult on ARM to produce a payload in which every instruction performs useful computation for the attacker. For this reason, most of the instructions executed in each gadget are actually filler instructions that do not perform useful computation. For example, in the case of the Thumb-mode gadget we describe below, the first instruction in the gadget is an unintended 16-bit Thumb instruction. After that, execution resynchronizes to the intended instruction stream, and eventually execution reaches the end of the gadget, where a function return sequence is

```
function readGadget(x) {
  return x ^ 0x11111610;
}
```

Listing 4.4. JavaScript function that produces a memory-read gadget when JIT compiled.

```
0x0:  mov    r2, lr
0x2:  str.w  r2, [r5, #-16]     ; save return address
...
0x32: ldr.w  r0, [r5, #-64]     ; load argument
0x36: movw   r12, #5648         ; 0x1610
0x3a: movt   r12, #4369         ; 0x1111
0x3e: eor.w  r0, r0, r12
0x42: mov.w  r1, #4294967295    ; 0xffffffff
0x46: ldr.w  r2, [r5, #-16]     ; load return address
0x4a: ldr.w  r5, [r5, #-40]     ; restore frame ptr
0x4e: mov    lr, r2
0x50: bx     lr                 ; return
...
```

Listing 4.5. Selected instructions from the DFG JIT compilation of the `readGadget` function given in Listing 4.4.

found. The return sequence enables the gadget to return control flow to the high level language.

Consider the following example: a memory read gadget. When compiled by JavaScriptCore's optimizing DFG JIT, the JavaScript function `readGadget` shown in Listing 4.4 will contain a gadget that will enable an attacker to read bytes from a specific address in memory into a JavaScript value. Listing 4.5 shows an excerpt from the JIT code emitted when `readGadget` is compiled to Thumb-2 instructions by JavaScriptCore's DFG JIT, which shows behavior that one might expect. Note the instruction at offset 0x42, which places 0xffffffff into `R1` before the return sequence. This instruction sets a register responsible for conveying to the caller the type of the return value; in this case, it indicates a 32-bit integer value.

Listing 4.6 shows the instruction stream when execution begins at the second halfword of the 32-bit instruction at offset 0x36. The halfword at offset 0x38 is a 16-

```
...
0x38: ldr    r0, [r2, #64]
0x3a: movt   r12, #4369         ; 0x1111
0x3e: eor.w  r0, r0, r12
0x42: mov.w  r1, #4294967295    ; 0xffffffff
0x46: ldr.w  r2, [r5, #-16]     ; load return address
0x4a: ldr.w  r5, [r5, #-40]     ; restore frame ptr
0x4e: mov    lr, r2
0x50: bx     lr                 ; return
...
```

Listing 4.6. Disassembly of the `readGadget` function's DFG JIT code starting from the middle of the intended instruction at offset `0x36`.

bit load instruction that adds 64 to the value in `R2` and loads a word of memory from that address into `R0`. Immediately thereafter, execution resynchronizes to the intended instruction stream. The upper halfword of `R12` is loaded with the value `0x1111`,[2] and the value that was read from memory is XORed against the entire contents of `R12`. Because the attacker may not be able to predict `R12`'s value at the time the gadget is invoked, it is necessary to make its contents predictable by writing `0x1111` into its upper halfword. The result of the XOR operation is returned to the JavaScript execution context as a 32-bit integer.

The attacker can recover 2 bytes of memory by XORing the upper halfword of the returned value against `0x1111`, and by repeatedly invoking the read gadget, the attacker can read out all but the first two bytes of a readable memory region. If the attacker can assume that `R12` will contain the same value at the beginning of every invocation of the read gadget, then it is possible to ascertain the value of `R12`'s lower halfword after the second invocation of the read gadget, at which point any memory marked readable can be read and un-XORed.

Figure 4.9 depicts a simplified view of calling and returning from the read gadget. A JavaScript control program calls a JIT-compiled wrapper function, which places the

[2]This pollution of `R12` is not harmful because JSC uses it as a scratch register.

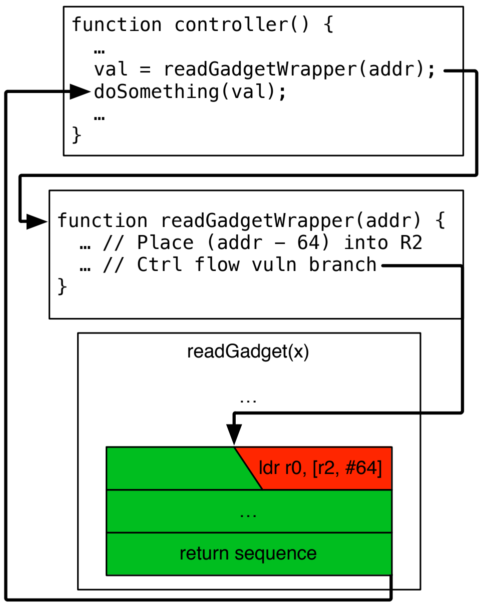

function controller() {
…
val = readGadgetWrapper(addr);
doSomething(val);
…
}
function readGadgetWrapper(addr) {
… // Place (addr - 64) into R2
… // Ctrl flow vuln branch
}
readGadget(x)
…
ldr r0, [r2, #64]
…
return sequence

Figure 4.9. Diagram of the invocation of the read gadget with arrows showing control flow.

address to be loaded (adjusted by the offset of 64) into a register and exploits a control flow vulnerability to branch to the read gadget. The read gadget computes the return value as described above, and its return sequence serves as a return from the wrapper function back into the control program. Certain details are abstracted away in Figure 4.9. Most notably, the wrapper function must populate certain registers needed as input to the gadget and must furthermore branch to the read gadget without perturbing either the JavaScript call frame pointer or the native stack pointer (the `SP` register). Otherwise, the gadget's return sequence will not work properly, and subsequent execution is at risk of crashing. Furthermore, the control flow vulnerability used is presumed to preserve the value placed in `R2` so that it can be used by the gadget. These conditions can be satisfied by taking advantage of the fact that JIT-compiled functions can be called directly from within another JIT-compiled function. If an attacker were to exploit a bug allowing her to trick the high level language runtime into writing a gadget's address in place of a function entry point, the read gadget could be called with the call frame register intact and without growing the native call stack. Consequently, the return from the gadget to the controller depicted in Figure 4.9 would succeed. Even the gadget's return value would be passed correctly to the controller and interpreted as the wrapper function's return value. Such a bug is plausible since a reference to each JIT code block's entry point is typically held in a heap object which could potentially be corrupted by the attacker to form the control flow vulnerability.

Giving an attacker the capability to perform arbitrary memory reads on top of a repeatedly-invokable control flow vulnerability is sufficient for her to perform arbitrary malicious computation using an attack dubbed Just-In-Time code reuse [51]. Just-In-Time code reuse allows an attacker to harvest the addresses of useful code sequences from the address space of a process protected by fine-grained ASLR. The code sequences could subsequently be used to launch a code reuse attack such as ROP. One of the requirements for Just-In-Time code reuse attack is an existing memory disclosure vulnerability: a `ReadByte(address)` function. A callable read gadget as described above provides exactly this functionality.

A register-to-register move instruction can be used to construct a gadget that leaks the contents of a register (even the stack pointer or link register) as a JavaScript numerical value. Since an attacker can influence the contents of one or more registers at the time a gadget is invoked, it is also possible to disclose the address of a JavaScript object by "casting" it to a number.

When compiled by JavaScriptCore's optimizing compiler, the JavaScript statement `return arg0 ^ 0x10` produces an `R2`-disclosure gadget that can be used to cast a pointer to a JavaScript object to a JavaScript numeric value. The gadget begins with an unintended `movs r0, r2` followed by `mov.w r1, #4294967295` (i.e., 0xffffffff, the JavaScriptCore tag for 32-bit integers) and a return sequence. If the gadget is invoked with a JavaScript object's payload field in `R2`, the object's payload, which is a pointer to the object itself, will be returned as a 32-bit integer. When used in conjunction with subsequent read gadget calls, the disclosure of an object's address could lead to the disclosure of a non-JIT code pointer. A non-JIT code pointer is desirable as a seed for code sequence harvesting in a code reuse attack; JITed code tends to branch directly only to other JITed code.

As we alluded to in the previous paragraph, a single JIT spraying attack using gadget chaining can make use of more than one gadget. This can be accomplished by spraying all necessary gadgets in the same payload at known offsets from one another. If the attacker is able to correctly guess the address of one gadget, she can use the known offsets to correctly deduce the addresses of the other gadgets. Gadgets—especially those leveraging unintended instructions decoded from unintended instruction boundaries—must be branched to at their exact addresses; otherwise, a useful, unsafe instruction might be skipped or improperly executed (e.g., an input register for an unsafe instruction might be clobbered by a preceding instruction). Placing gadgets at predictable offsets requires heap feng shui [53] tailored to the JIT under attack. For concrete examples of this, see Sections 5.2 and 6.2, in which we describe concrete heap feng shui for gadget chaining against JavaScriptCore and V8, respectively.

The unsafe instructions in a gadget need not be unintended instructions. The gadget chaining attack against V8 (Section 6.2) utilizes a store gadget containing only intended ARM instructions.

Chapter 4, in part, is a reprint of the material as it appears in *Proceedings of the 2015 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, has been submitted for publication of the material as it may appear in appear in *Proceedings of the 2017 Network and Distributed System Security*

Symposium. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Thumb gadget chaining against JavaScriptCore

5.1 The JavaScriptCore JavaScript Engine

JavaScriptCore (JSC) is the JavaScript engine used by the WebKit layout engine. It is an attractive platform to target because of iOS's code signing policy, which requires all executable code running on iOS to be cryptographically signed, with one exception—JIT code emitted by JSC. Hence, JSC is used by Chrome, Safari, and Firefox—the three major browsers—on iOS. Since the code signing policy complicates traditional code injection attacks, JIT spraying against JSC is an especially compelling attack path. In this section, we describe JSC for ARMv7-A as it appears in the WebkitGTK version 2.2.2-1 port for Debian; this is the version of JSC that we target in our proof of concept JIT spraying attack in Section 5.2.

JSC is a multi-tier JavaScript engine, meaning it will recompile a piece of JavaScript code with increasing levels of optimization as that code's execution count grows. When JSC is given a piece of JavaScript source code, it is first compiled down to bytecode. Initially, the bytcode is interpreted, but once it has been executed several times (6 times for functions or 100 times for loops), the bytecode is compiled down to unoptimized native code[1] by the Baseline JIT. Once the Baseline JIT's code has executed many times (60 times for functions or 1000 times for loops), the Data Flow Graph (DFG) JIT kicks in and emits optimized Thumb-2 code. An adversary can induce JSC into compiling a piece of JavaScript with any tier by selecting an appropriate number of times the script is invoked.

Both the Baseline JIT and DFG JIT produce Thumb-2 code on 32-bit ARM machines, and the code resides in memory pages that are marked readable-writable-executable (RWX). The write flag remains throughout the lifetime of the JIT code because JSC occassionally modifies the native code in situ.

5.1.1 Low Level Interpreter

At the bottom tier lies the Low Level Interpreter (LLInt), which interprets bytecode. Bytecode consists of 32-bit opcodes followed by as many 32-bit operands as are required by that opcode. Bytecode opcodes are pointers to pre-compiled code snippets in the interpreter's text section implementing the bytecode operations. During bytecode execution, a virtual program counter (vPC) register points to the currently-executing opcode in the bytecode while the real `PC` is in the code snippet pointed to by the opcode. The snippet accesses the opcode's operands via vPC-relative memory loads, performs the desired computation (optionally storing results onto a special JavaScript stack), advances the vPC, and finally branches to the next opcode's snippet via a register-indirect jump through the vPC.

5.1.2 Baseline JIT

Cold code that has become "warm" gets compiled to native code by the non-optimizing Baseline JIT. The instruction stream produced by the Baseline JIT differs

[1]On ARMv7-A, all JSC JIT tiers produce Thumb-2 code.

slightly from the one executed by the LLInt since it does not need to manage the vPC; but they are functionally equivalent. Baseline JIT code has clear boundaries where the execution of one bytecode instruction ends and the next begins, and it does not flow scratch values in registers across those boundaries. Instead, scratch values are stored onto the JavaScript stack and read back out by subsequent bytecode operations.

5.1.3 Data Flow Graph (DFG) JIT

During execution in the LLInt and Baseline JIT, JSC collects type profiling information in order to try to predict the types of operands found in the code. The DFG JIT uses this type profiling information to aggressively optimize "hot" code for what it perceives to be the common case. When a piece of DFG JITed code is executed, and the runtime data types match those predicted by the DFG JIT, execution continues on the fast path through the optimized DFG JITed code. Otherwise, execution will fall back to the Baseline JIT code via a process known as an on-stack-replacement (OSR) exit. Among the DFG JIT's features are dead code elimination analysis, function inlining, and a basic register allocator.

5.1.4 Fourth Tier LLVM (FTL) JIT

In May 2014, the WebKit developers enabled what is known as the Fourth Tier LLVM (FTL) JIT as an additional compilation tier. The FTL JIT utilizes the LLVM compiler infrastructure to provide a higher-performance alternative to the DFG JIT [43]. Since the FTL JIT was still in the experimental development phase in the version of JSC under study, its functionality is out of the scope of this chapter.

5.1.5 JavaScript value representation

All values in JavaScript are stored as 64-bit IEEE 754 double precision floating-point numbers. Non-float 32-bit data types are encoded in the floats using NaN values. A

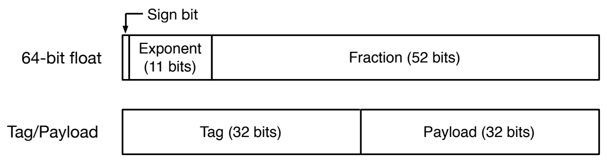


Figure 5.1. Illustration mapping the bits of an IEEE 754 double precision floating-point number to the tag and payload portions of a 32-bit JSC JS value.

NaN (Not a Number) value is a floating point value wherein all the bits in the exponent component are set, and the fraction component is non-zero. The low-order 32-bits of the float store the non-float's value; this is called the *payload* field. The upper half of the float is a 32-bit *tag* field, whose value indicates the value's type. If the 64-bit value can be interpreted as a non-NaN float, then the value is interpreted as a 64-bit float; otherwise, its type is given by the tag field. Figure 5.1 shows the mapping from a 64-bit floating point value to a tag/payload pair. Tags for non-floats are assigned in such a way that the value will be a NaN float. Some tag examples are `0xffffffff` for 32-bit integers and `0xfffffffb` for pointers to objects.

5.1.6 JavaScript call stack and calling convention

JavaScriptCore allocates a stack separate from the one used by native host functions. Rather than maintaining a pointer to the head of the stack, a register holds a pointer to a fixed location in the current function's call frame; the value of the caller's call frame pointer is stored in the callee's stack frame and explicitly restored by the callee before returning. The JS call frame also stores function arguments, local variables, the return address, and pointers to other runtime objects.

When performing a function call, the calling function allocates the callee's call frame and populates it with everything except the return address. The caller then updates the call frame register to point to the callee's call frame and branches to the callee with a linking branch instruction (such as `bl` or `blx`). The callee immediately saves the return address (which was placed into the `LR` register by the linking branch instruction) to its call frame. After it performs its computation, the callee loads its 64-bit return value (as described in § 5.1.5) into registers `R0` and `R1`, restores the call frame register to point to the callee's call frame, and branches back to the caller using the value it saved to its call frame.

5.2 Proof of concept gadget chaining attack

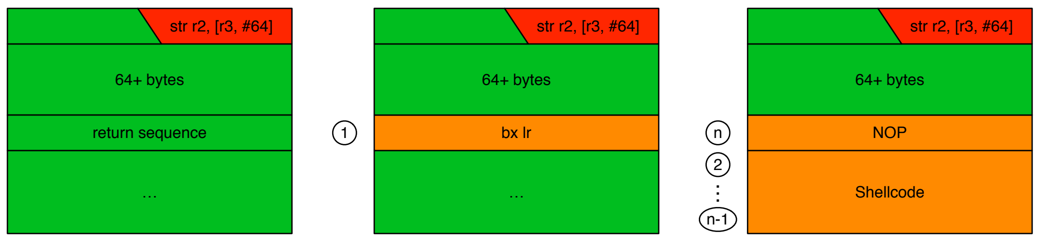
In this section we demonstrate the use of gadget chaining in the construction of an end-to-end proof of concept attack against JavaScriptCore on ARMv7-A. The attack uses a memory store gadget to corrupt JIT code memory and execute arbitrary code.

The high level overview of the attack is as follows: The attacker lures the victim's WebKit browser into loading and executing attacker-provided JavaScript. For example, the attacker could purchase web advertisements, enabling her to push arbitrary iframe contents to any client to whom the ad is shown. The attacker's JavaScript induces JSC on the client to repeatedly JIT-compile a JavaScript function containing a memory store gadget (a gadget beginning with an unintended `str Rt, [Rn, #imm]`), producing multiple copies of its JIT code in RWX memory; this is the spraying stage. The sprayed function is devised in such a way that the memory store gadget resides at a known offset in each 4 KB page that has been sprayed.

The attacker is *assumed* to have corrupted a function pointer using methods outside the scope of this dissertation (e.g., via a use-after-free bug).[2] The function whose pointer was corrupted should have the the following two properties:

1. Execution can be induced on-demand by the attacker

[2]We simulate a use-after-free vulnerability by adding a virtual function, `hijackVFT()`, that can be called from JavaScript to WebKit's HTMLInputElement object. The function overwrites the vtable pointer for the object against which it is called so that it uses a fake VFT with one of its values overwritten with an attacker-provided value.



(a) Initial memory layout.

(b) First gadget invocation replaces return sequence.

(c) Subsequent invocations write shellcode and finally replace return sequence with NOP, allowing execution to fall through into shellcode.

Figure 5.2. Progression of our proof of concept attack's self-modifying code. Encircled numbers to the left show the order in which the self-modifying writes occur.

2. Accepts two attacker-provided 32-bit arguments that will be passed in registers rather than on the stack. These two registers correspond to the `Rt` and `Rn` register fields in the sprayed memory store gadget

The attacker guesses the address of a page where a sprayed instance of the memory store gadget will reside and uses the memory store gadget's known page offset to form the address used to corrupt the function pointer.

The attacker uses the corrupted function pointer to invoke the memory store gadget, providing values for its `Rt` and `Rn` fields as arguments to the function. The memory store gadget enables the attacker to write arbitrary words to arbitrary memory locations, and the attacker leverages this capability to modify the gadget's own code. Initially the gadget's memory is laid out as shown in Figure 5.2a. The first invocation of the memory store gadget performs self modification of the gadget's original return sequence, overwriting it with an alternative return sequence (Figure 5.2b). This allows the gadget to return control flow back to JavaScript without crashing. Subsequent invocations of the gadget copy shellcode into the memory following the alternative return sequence. Finally, the gadget is invoked to overwrite the alternative return sequence with a NOP instruction, allowing execution to fall through into the shellcode (Figure 5.2c). In Figure 5.2, the correspondence between the 64-byte offset in the gadget's store instruction and the size of the 64+ byte gap between the store instruction and return sequence is merely coincidental. The store instruction's offset is a product of the encoding of the intended instruction from which it is derived, and the 64+ byte gap is dictated by the size of an instruction cache line on the 32-bit ARMv7-A machine on which we tested our attack.

In the remainder of this section, we explain the details of our proof of concept attack, most notably those that pertain to the following four major components of deploying an attack using gadget chaining:

1. generating the store gadget
2. pinpointing gadgets in memory
3. preparing registers and branching to gadgets from JavaScript
4. returning from gadgets without crashing

5.2.1 Gadget generation

We know from our discussion of controlling JIT output in Section 4.2 that we can setup registers in such a way that the statement

```
var R4 = R2 ^ 0x09a00000;
```

will be compiled by the DFG JIT to the 32-bit Thumb-2 instruction

```
eor.w r4, r2, #161480704
```

The second half of this instruction encodes the 16-bit unintended Thumb instruction

```
str r2, [r3, #64]
```

which we use in our memory store gadget. The use of registers `R2` and `R3` as operands in the store instruction is deliberate, and the reason will become clear in Section 5.2.3. The displacement value of $+64$ is due entirely to the choice of `Rd = R4` in the intended `eor.w` instruction, and the code that calls the gadget can adjust by simply subtracting 64 from the desired store target address. Although we used the XOR operation, the other bitwise operations (i.e., AND & OR) would work as well.

5.2.2 Pinpointing gadgets in memory

To set up the hijacked function pointer, the attacker must first guess an address where she hopes a memory store gadget has been sprayed. This guess must be correct down to the halfword, since being off by even one halfword will result in failure to execute the unintended instruction(s) at the beginning of the gadget. This is a direct result of the fact that it is difficult to chain long sequences of unintended instructions together in JIT-emitted code on ARM, making it impossible to create a NOP sled.

To increase the probability of correctly guessing the memory store gadget's address, we leverage the predictability of JSC's code generation and memory allocation to place instances of the gadget at the same known offset on each 4 KB page. It is advantageous to the attacker that the gadget be produced by the DFG JIT rather than the Baseline JIT. This is because JSC's Baseline JIT performs a weak form of random NOP insertion as a JIT spray mitigation. The idea behind random NOP insertion is that if a NOP instruction is inserted into the middle of a sequence of intended instructions, the behavior of the unintended instruction stream that it encodes would no longer be predictable and might even resynchronize with the intended instruction stream. The Baseline JIT implements random NOP insertion by emitting a 2-byte NOP instruction at the beginning of each compilation output with 50% probability. The DFG JIT, on the other hand, does not perform random NOP insertion, so the gadget is guaranteed to begin at a fixed offset from the beginning of the function's DFG JIT code.

What remains to be shown is how one can force each instance of the store gadget function's DFG JIT code to be placed at a fixed offset on each sprayed page. We leverage the following properties of WebKit to create memory holes just large enough for the DFG JIT code at a predictable offset on all sprayed pages:

# 1: Native code emitted by both the Baseline JIT and DFG JIT share the same pool of executable memory regions.

# 2: The Baseline JIT does not perform dead code elimination, but the DFG JIT does. The introduction of dead code can lead to DFG JIT code that is considerably more compact than its Baseline JIT counterpart.

# 3: JIT-emitted code "shrinks." That is, the code initially produced by a JIT compiler is larger than the code that is eventually executed, and the extra space is released back into the pool of free memory regions. Shrinkage occurs due to minor space-saving optimizations of certain instructions. For example the original form of a 4-byte PC-relative branch in the code's final form may have formerly been a register-based branch consisting of two 4-byte instructions to load the register with an absolute address followed by a 2-byte instruction to branch through the register.

# 4: The executable memory allocator used to allocate JIT code minimizes the number of committed pages when fulfilling an allocation request by selecting the smallest free memory region that is at least as large as the requested size and allocating the space from either the low-addressed end or the high-addressed end of the region in such a way that the newly-allocated space spans the fewest pages (with preference for the low-addressed end of the region if both ends would result in the same number of spanned pages). If no existing free region is large enough to fulfill the allocation

request, a new 16 KB region is mapped and added to the pool of free regions, but it is not merged with existing regions that are contiguous with it.

# 5: The executable memory allocator allocates chunks at 32-byte granularity, rounding up allocation requests if necessary. The Baseline JIT does not return unused bytes created by this rounding to the free pool.

Let $|Baseline_{\mathrm{ps}}|$, $|Baseline_{\mathrm{s}}|$, $|DFG_{\mathrm{ps}}|$, and $|DFG_{\mathrm{s}}|$ denote the size in bytes of the store gadget function's pre-shrinking Baseline JIT code, shrunken Baseline JIT code, pre-shrinking DFG JIT code, and shrunken DFG JIT code, respectively. The store gadget function is written in such a way that the following constraints are met:

- $|DFG_s| \leq |DFG_{\mathrm{ps}}| < |Baseline_{\mathrm{s}}| < |Baseline_{\mathrm{ps}}|$
- $|Baseline_{\mathrm{ps}}| \approx 4076$
- $|Baseline_{\mathrm{ps}}| + |Baseline_{\mathrm{s}}| > 4096$
- $|Baseline_{\mathrm{s}}| + |DFG_{\mathrm{ps}}| \leq 4096$

Suppose the attacker's JavaScript control program is the only thread initiating requests to the executable memory allocator and is repeatedly allocating space for Baseline JIT code. Once the memory pool no longer contains regions 4 KB or larger, a fresh 16 KB region will be mapped and added. Figure 5.3 depicts how the allocator will place consecutively-allocated Baseline JIT code instances in this region. The first allocation occurs at the beginning of the region, and subsequent allocations walk backwards 4 KB at a time from the end of the region, leaving between them holes just large enough for the DFG JIT code to be placed.

The reason this occurs is simple, yet subtle. The pre-shrinking Baseline JIT code will result in an allocation request that is rounded up to 4 KB. The pre-shrinking size of

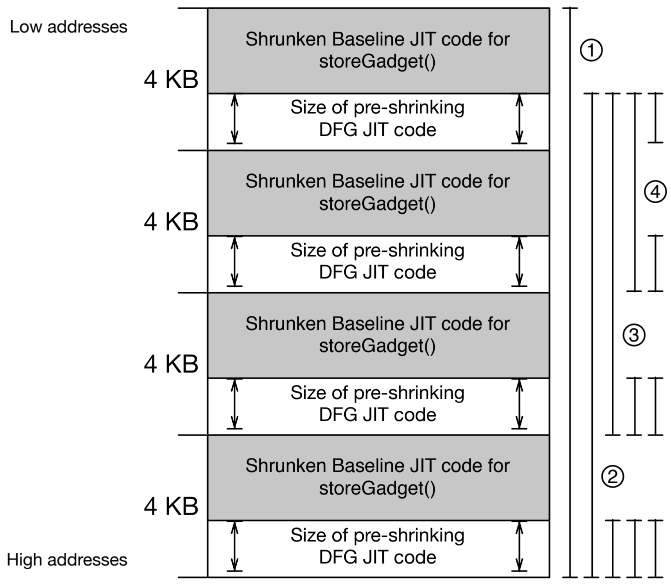

Low addresses
High addresses
4 KB
4 KB
4 KB
4 KB
Shrunken Baseline JIT code for storeGadget()
Size of pre-shrinking DFG JIT code
Shrunken Baseline JIT code for storeGadget()
Size of pre-shrinking DFG JIT code
Shrunken Baseline JIT code for storeGadget()
Size of pre-shrinking DFG JIT code
Shrunken Baseline JIT code for storeGadget()
Size of pre-shrinking DFG JIT code
①
②
③
④

Figure 5.3. Layout of executable memory showing holes created between instances of the store gadget function's Baseline JIT code. Bars on the right show the progression of the available space as instances of the Baseline JIT code are allocated. The encircled numbers indicate the order of Baseline JIT code allocation.

approximately 4076 bytes ensures that despite small size reductions due to uncontrollable optimization and small expansions due to constant blinding, the allocation request will be 4 KB. The first Baseline JIT allocation request will be placed at the beginning of the 16 KB region since it is guaranteed not to span more than one page. It will then be shrunk, and the returned free space will be merged with the rest of the free space in the region. The next 4 KB Baseline allocation will start 4 KB from the end of the region rather than immediately after the first allocation because that will prevent the second allocation from spanning multiple pages. The third allocation will be placed 4 KB before the second and so forth. The unused fragments of memory left behind after shrinking the Baseline JIT code are large enough to allocate instances of the DFG JIT code, but not large enough that they can be used by Baseline JIT code instances. All of these memory holes begin at the same offset on each page because the allocator's 32-byte allocation granularity absorbs small deviations in the Baseline JIT code's shrunken size. The holes are filled by spraying at least as many DFG JIT instances as we did Baseline JIT instances.

5.2.3 Preparing registers and branching to gadgets from JavaScript

The memory store gadget's unintended `str` instruction requires the following two inputs: a register containing the word to be written and a register containing a base address to which a known offset will be added to form the memory address. Since the ARM ABI places some function arguments in registers, a hijacked function pointer is a perfect avenue for both loading registers with attacker-provided values and branching to the gadget. The `HTMLInputElement` object in the WebKit DOM exposes the virtual method `void setRangeText(replacement, start, end, selectionMode);` to JavaScript. This function is perfect for our attack because the `start` and `end` parameters are 32-bit integer values that are passed in registers (they are among the first four parameters, counting the `this` pointer). Furthermore, since it is a virtual function, it is a candidate for a vtable hijacking attack.

Once the vtable of an instance of the `HTMLInputElement` class has been hijacked, the attacker can issue a call to the hijacked object's `setRangeText` method with specially-chosen values for `start` and `end`, whose values will be placed into `R2` and `R3`, respectively. Execution will then branch to the gadget. Supposing the unintended store instruction in the gadget is of the form `str r2, [r3, #imm]`, the value passed as the `start` parameter will be stored into memory at the address given as the `end` parameter (plus the offset value).

5.2.4 Returning from gadgets without crashing

The `setRangeText` method whose function pointer we hijack is a so-called "host function," meaning it is compiled ahead of time and is exposed to JavaScript for calling through Web IDL. Host functions operate on the native call stack rather than the special

JavaScript stack, and they do not observe the same register-preservation practices as JavaScript. In order for JavaScript code to call host functions, a “prototype function” serves as an intermediary. A prototype function is responsible for storing callee-saved registers onto the native stack, placing arguments passed from JavaScript into their proper locations for a native function call, and calling into the host function. Once the host function returns into the prototype function, it marshalls the return value into a JavaScript value, and returns to the JavaScript caller. The prototype function acts as the wrapper function shown in Figure 4.9.

However, the return sequence found in the gadget cannot be used to return to the prototype function’s JavaScript caller. This is because the return sequence at the end of the gadget performs a *JavaScript* return, which retrieves the return address from the JavaScript call frame. This requires that the register holding the JavaScript call frame pointer be preserved by the prototype function, a property which is not guaranteed and in fact does not hold for `setRangeText`’s prototype function. Moreover, even if the call frame pointer were preserved, returning back to the JavaScript controller from the gadget will leave certain registers unrestored and the native call stack in an inconsistent state due to the saved registers pushed onto it by the prototype function. Any subsequent computations that rely on the contents of the saved registers or the native call stack are likely to crash the process if the saved registers are not popped off the stack.

In order to decouple our attack from JSC’s choice of call frame register and ensure the integrity of registers and the native call stack, the first invocation of the store gadget must overwrite its own return sequence with a `bx lr` instruction (Figure 5.2b), which will cause execution to return to the prototype function, where the saved registers will be popped off of the stack before control is returned to the JavaScript controller. Since the prototype function calls the host function using a linking branch, we can expect `LR` to hold the correct return address so long as the new `bx lr` instruction precedes any instructions in the gadget which would overwrite it. Fortunately, the only such instruction in the gadget is the instruction in the gadget's original return sequence which loads the return address from the JavaScript call frame.

A final concern for returning from a gadget is ensuring that the newly-written `bx lr` instruction does not reside on the same i-cache line as the unintended store instruction. If they were to be on the same i-cache line, the overwritten instruction would exist only in the data cache and/or main memory. The intended instruction that we wanted to overwrite would remain intact in the i-cache and would be executed, leading to the crash we were trying to avoid. In order to prevent this scenario, we pad the store gadget function with code that will yield at least 64 bytes (the size of an i-cache line on our 32-bit ARMv7-A test machine) of instructions between the unintended store and the return sequence.[3] With the padding in place, the newly-written return sequence will only be loaded into the i-cache after the unintended store has executed. In order to ensure that the cache line containing the instructions to be overwritten is not in the cache prior to executing the gadget, the attacker should use JavaScript to induce JSC to execute many non-sprayed functions after spraying and before invoking the gadget. The same cache flush should be done after the shellcode has been injected but before the new return instruction is overwritten with a NOP instruction.

5.2.5 Analysis of the proof of concept attack

One of the most important metrics when evaluating a spraying attack is its success rate. The success rate of a JIT spraying attack (P_{success}) is expressed by the following

[3]This is another reason it is important to generate the gadget with the DFG JIT. Baseline JIT code loads and saves operands onto the call stack for each bytecode instruction, whereas the DFG JIT can allocate scratch registers to avoid memory accesses. We do not know of a method to generate 64 bytes of padding code with the Baseline JIT that does not invoke a memory access through the call frame pointer. Since we would like to avoid relying on the validity of the call frame pointer register, DFG JIT code is ideal.

equation:

$$P_{\text{success}} = P_{\text{vuln}} \times P_{\text{page}} \times P_{\text{offset}} \times P_{\text{bytes}}$$

where P_{vuln}, P_{page}, P_{offset}, and P_{bytes} are defined as follows:

- P_{vuln}: the probability that the attacker's control flow vulnerability results in populating the gadget's input registers with the appropriate values and branching execution to an attacker-chosen address. We *assume* a best-case value of 1.

- P_{page}: the probability of correctly guessing a page containing sprayed instructions. The attacker can maximize this probability by spraying more gadgets and using an address disclosure vulnerability. We estimate this quantity via an empirical measurement. In a 32-bit address space, there are $2^{20} \approx 1M$ pages. We were able to spray about 200,000 pages (19.1% of the virtual address space) of JIT code on a 32-bit machine with 4 GB of physical memory before the browser process exhausted its memory and crashed. The fraction of pages that can be JIT sprayed is limited by the presence of LLInt bytecode and other heap objects that are allocated for every instance of the sprayed function (63 pages for every 37 pages of JIT code). The blocks of pages containing this support data are interleaved with regions of JIT code pages. Therefore a perfect pair of address disclosures which tightly bounds the memory region containing all JIT code and support data cannot improve P_{page} beyond 37%, and without the address disclosure, $P_{\text{page}} = 19.1\%$.

- P_{offset}: the probability that the function containing the sprayed gadget on the guessed page begins at the expected page offset. A sprayed function can begin at an unexpected page offset if the memory hole into which it was sprayed was misaligned as a result of several low-probability events causing the size of Baseline JIT code to vary unpredictably. Fortunately for the attacker, the allocation offsets for Baseline JIT code have an opportunity to resynchronize for every new 16 KB region that is allocated, so misalignments do not cascade. We measured the alignment of 100,000 consecutively-allocated memory store gadgets and found that 97,826 of them were correctly aligned, giving us an empirical estimate for P_{offset} of 97.826%.

- P_{bytes}: the probability that the intended instruction that encodes the gadget is the instruction expected by the attacker. JSC randomly applies the constant blinding JIT spraying defense, which scrambles constants in the instruction stream, leading to unexpected instructions. However, only 1 out of 64 constants are scrambled at random, giving us $P_{\text{bytes}} = 63/64$.

The success rate of our proof of concept attack is 35.6% if it makes use of address disclosures that perfectly bound the sprayed pages or 18.4% without them (randomly guessing for 200,000 sprayed pages). Without our techniques for placing JIT code at known page offsets, Blazakis' x86 JIT spraying attack, which relies on an 80% probability NOP sled ($P_{\text{offset}} = 80\%$), would succeed under JSC's memory layout constraints with probability at most 29.1% and 15.0% with and without perfect bounding address disclosures, respectively.

Chapter 5, in part, is a reprint of the material as it appears in *Proceedings of the 2015 Network and Distributed System Security Symposium.* Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 6

ARM Gadget Chaining against V8

6.1 The V8 JavaScript Engine

V8 is the JavaScript engine used by the the popular Chrome and Chromium web browsers on all platforms except iOS, which only allows the JavaScriptCore JavaScript engine to dynamically-generate code. In this section, we describe V8's JITs as they exist at Git commit `1398078`; this is the version of V8 against which we targeted the proof of concept JIT spraying attack in Section 6.2.

Unlike JSC and SpiderMonkey, V8 does not include an interpreter—although one named Ignition is currently under development. Instead, all code is JIT compiled for its first execution. V8 is a two-tier JavaScript engine composed of the Full Codegen JIT and the Crankshaft JIT.[1] The first time any JavaScript function is executed, it is compiled by the Full Codegen JIT; therefore, the Full Codegen JIT must work fast. Each function is first parsed into an abstract syntax tree (AST), which the Full Codegen JIT uses to emit non-optimized code. The Full Codegen JIT does not employ a register allocator; local variables and intermediate values that persist across the boundaries of an operation are stored on the stack. The code emitted for each operation loads operands into statically-predetermined registers before the operation is actually carried out. Full

[1]A third tier optimizing JIT called Turbofan was added to V8 after the V8 version under study. We do not include it in our discussion in this chapter.

Codegen JIT code collects type information which is used by Crankshaft to produce optimized JIT code.

Crankshaft is V8's JIT compiler that kicks in to compile "hot" functions. Using the type information gathered during the execution of Full Codegen JIT code, Crankshaft produces code that is optimized for the value types that the function is expected to encounter (i.e., the types that that function has seen in the past). Since Crankshaft compilation is asynchronous, its compilation process need not be as fast as the Full Codegen JIT's; and it uses this relaxed schedule to perform register randomization and other compiler optimizations such as loop-invariant code motion and inlining. If a Crankshaft-compiled function encounters values with unexpected types, its Crankshaft code is discarded, and execution falls back onto Full Codegen code. The function is eligible to become hot again, at which point it will once more be compiled by Crankshaft.

Both the Full Codegen JIT and Crankshaft emit ARM instructions on 32-bit ARM machines, and like JSC, V8's the memory protection status of JIT code pages is permanently read-write-execute (RWX) to support runtime patching. V8 limits the amount of JIT code memory that can be allocated. On 32-bit sytems, the limit is 256MB, and on 64-bit systems, it is 512MB.

6.2 Proof of concept gadget chaining attack

In this section, 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 blind JIT spraying attack that uses JIT-emitted instructions 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 in Section 4.5; 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 HLL code that invoked it.

The high level structure of the proof of concept attack is the same as the gadget chaining attack against JSC (Section 5.2). After JIT spraying a particular intended 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.

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

6.2.1 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 state of registers is not guaranteed to result in valid memory addresses; moreover, store instructions could clobber critical machine state. They must also store results only to caller-saved registers because they will execute as intended and must preserve the values of callee-saved registers for when the gadget returns.

It is necessary to inject a return instruction rather than allowing control flow to 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 the sprayed store 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.

V8 performs code caching; therefore redefining and invoking the same function more than once will not result in multiple copies of that function being compiled. To circumvent code caching, we inject a constant counter value as a term in the computation of the function's return value and only use each counter value once.

6.2.2 Artificial control flow vulnerability

For the purposes of 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 which performs the desired corruption. Hijacked virtual functions are especially useful for a gadget chaining because they can serve two purposes, which are 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`.

6.2.3 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 is unable to fulfill an allocation request from the current pools of free JIT code memory. Allocation requests are satisfied starting at the low-addressed end of the new chunk. If we are able to 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 only need to guess 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 defines 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 6.1 illustrates how we accomplished this by placing a conditional return early in the sprayed function to take

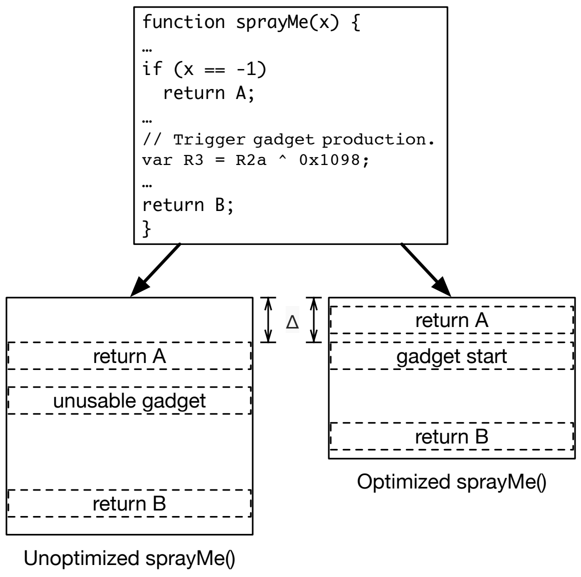


Figure 6.1. Illustration of how a return instruction in unoptimized JIT code is aligned to the same function offset Δ as a gadget in optimized JIT code.

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 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 6.2 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

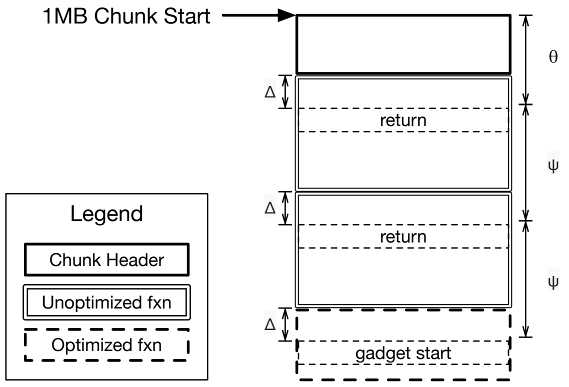


Figure 6.2. Illustration of the beginning of a 1MB chunk that can be probed for the location of a gadget in a failure-tolerant manner. An incorrect guess of θ or $\theta + \psi$ will only execute a harmless return instruction. Δ is the common offset of both the return instruction and the gadget in both the unoptimized and optimized code.

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 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.

6.2.4 Analysis

Recall that 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%.

Chapter 6, in part, has been submitted for publication of the material as it may appear in appear in *Proceedings of the 2017 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 7

ARM-to-Thumb Self-sustaining JIT Spraying against SpiderMonkey

7.1 SpiderMonkey JavaScript Engine

SpiderMonkey is a three-tier JavaScript engine developed by Mozilla and integrated into the Firefox web browser. Like V8, SpiderMonkey is not deployed on iOS due to its code signing policy; instead, Firefox for iOS uses WebKit and JavaScriptCore for rendering and JavaScript support. In this section, we describe SpiderMonkey at Git commit `ce31ad5`, the version against which we base our attack in Section 7.2 and on top of which we based the implementations of diversification defenses described in Section 8.4. SpiderMonkey features a bytecode interpreter, the non-optimizing *Baseline* JIT compiler, and the optimizing *IonMonkey* JIT compiler. Both of the JIT compilers emit code from the ARM instruction set on 32-bit ARM machines. We discuss each of these execution tiers in turn.

7.1.1 Bytecode Interpreter

After SpiderMonkey parses JavaScript to an abstract syntax tree, a code generator traverses the tree and emits the bytecode which will serve as the low-level representation of the code throughout its lifetime. SpiderMonkey bytecodes are instructions to a stack-based virtual machine; each bytecode retrieves its operands off the top of a stack and pushes 0 or more results onto it afterwards. All JavaScript functions are first executed in the interpreter, during which time information about the types of values encountered by the function is recorded. The interpeter maintains a JavaScript call stack that is separate from the native C stack; this is in contrast to the Baseline JIT and IonMonkey, which both use the native C stack.

7.1.2 Baseline JIT

SpiderMonkey's Baseline JIT was built as a replacement for the now-deprecated JaegerMonkey JIT compiler; its purpose is to serve as an intermediate JIT between the interpreter and IonMonkey. Once a function becomes hot in the interpreter, it is compiled by the Baseline JIT. Like JSC's Baseline JIT compiler, SpiderMonkey's Baseline JIT does not aim to produce the most optimized code possible. It quickly compiles bytecode to native code that executes orders of magnitude faster than its interpreted counterparts. Another similarity to JSC's Baseline JIT compiler is the lack of a register allocator; the native code emitted for each bytecode operates on a fixed, pre-determined set of registers and saves values that must persist across bytecodes onto the stack.

Baseline JIT code collects type information via inline caches, and the performance of a function can improve over time as new optimized inline cache stubs are added. The collected type information will be used if the function becomes hot enough to warrant compilation by IonMonkey. Since Baseline JIT code collects type information, there is no need for a Baseline-compiled function to ever bail out back to the interpreter if unexpected types are encountered (unlike its predecessor JaegerMonkey, which did not collect type information).

7.1.3 IonMonkey JIT

IonMonkey is SpiderMonkey's optimizing JIT compiler that is reserved for very hot functions. Like V8's Crankshaft, IonMonkey makes use of type information gathered during execution in lower tiers to produce code tailored to the types that a function is expected to encounter. IonMonkey's compilation process occurs in the following five phases:

1. The function's bytecode is lowered to an architecture-independent intermediate representation called MIR.
2. Numerous analyses and optimizations are performed over the MIR.
3. MIR is lowered to an architecture-dependent intermediate representation called LIR.
4. Physical registers are allocated to values in LIR.
5. LIR is provided to a code generator, which emits native code.

If JIT code produced by IonMonkey encounters a type that was not optimized to handle, execution bails out back to Baseline JIT code.

7.2 Proof of concept Turing-complete self-sustaining payload

7.2.1 Implementing an SBNZ One Instruction Computer

Using the method described in Section 4.4.2, we constructed an ARM-to-Thumb self-sustaining JIT spraying payload that implements the interpreter loop for a One Instruction Set Computer (OISC) [37], an abstract universal machine that has only one instruction. There are many options for the single instruction; we implemented *Subtract*

```
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 7.1. Pseudocode for the `sbnz` instruction.

and Branch if Non-Zero (SBNZ). Listing 7.1 shows the pseudocode for the `sbnz` instruction. Given enough memory, an OISC is capable of universal computation; therefore this payload demonstrates the feasibility of performing Turing-complete computation using an ARM-to-Thumb self-sustaining payload.

In addition to the general purpose register that must be repurposed as a virtual PC in order to optimize the memory usage of Thumb-to-ARM self-sustaining payloads, we designate another general purpose register as the OISC PC and use unintended instructions to implement the SBNZ instruction semantics and update the OISC PC. The OISC PC is initialized with the value of the stack pointer. In other words, the first instruction, composed of four consecutive 32-bit addresses corresponding to the four instruction operands, is expected to reside at the top of the stack when it begins executing.

The instructions used to build the interpreter are shown in Table 7.1. 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.

7.2.2 Encoding challenges

We overcame three major hurdles during the construction of the SBNZ OISC interpreter payload. They were IonMonkey's register allocator, dead store elimination op-

Table 7.1. 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 (× 11)	adds r6, r7	and r4, r1, #1040187392
13	non_zero_loopback	subs r6, #162	and r3, r1, #2592
14–15	vpc_advance (× 2)	adds r6, r7	and r4, r1, #1040187392
16	zero_loopback	subs r6, #217	and r3, r1, #3472
1–16	branch_vpc (× 16)	mov pc, r6	and r4, r1, #191889408

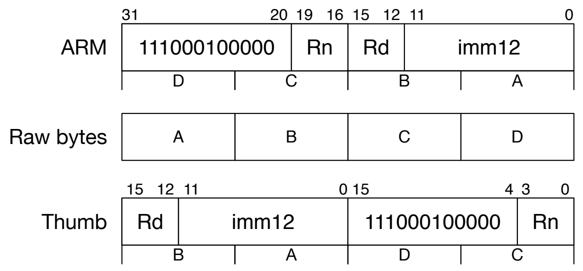


Figure 7.1. Illustration of how the immediate-operand bitwise AND instruction from the ARM instruction set (top row) can be decoded as two 16-bit Thumb-2 instructions (bottom row).

timizations, and the ARM architecture's restrictions on the encodings of 12-bit modified constants. We discuss each of these challenges in turn in this subsection.

Register allocation

To a first approximation, IonMonkey's register allocator iterates over values in order of decreasing lifespan, allocating them to registers (by probing for an available register between `R0` and `R11`, inclusive, in order of ascending register number). Once registers have filled up, a shorter-lived value can evict a longer-lived value from its registers if the shorter-lived value has a higher use count-to-lifespan ratio (a.k.a. spill weight). The register allocator allows a small number of evictions on behalf of a value before the value's lifespan is partitioned, and allocation is re-attempted on the resulting fragments.

The JavaScript function that generates the proof of concept payload creates variables named after ARM core registers and coerces IonMonkey into storing those variables' values into the registers corresponding to their names. This is advantageous because subsequent uses of and assignments to those variables in the JavaScript result in predictable instruction operands in the native code. For instance, the JavaScript statement

```
R11 = R1 & 130;
```

will be compiled to the ARM instruction

```
and r11, r1, #130
```

We influence the allocation of variable values to registers by fine tuning the lifespans of variables to alter the order in which they will be allocated to a register. By creating only the necessary number of variables, evictions do not occur, and the variable with the longest lifespan will be allocated to `R0`, the next longest to `R1`, etc. In particular, the strategy we employed was to work upwards from the lowest-numbered registers,

```
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 7.2. The structure of the JavaScript function sprayed to produce the self-sustaining SBNZ OISC payload. Mutually-exclusive conditional blocks prevent dead store elimination.

extending the corresponding variables' lifespans until they were correctly allocated. Although tedious, this process is relatively straightforward.

There were a number of registers that were not used in the payload and for which an exact variable-register mapping was unnecessary; in those cases, we simply ensured that a non-essential variable was allocated to the register.

Dead store elimination

Many consecutive intended AND instructions in the proof of concept payload store their result into the same register (e.g., all intended instructions that encode branch block 3 in Table 7.1 are identical). To prevent IonMonkey from emitting only the last instruction in such a sequence (dead store elimination), we place each bitwise AND in a separate mutually-exclusive `else-if` block, as shown in Listing 7.2. The consequence of this code structure is that intended AND instructions cannot be emitted back-to-back. Instead, IonMonkey only emits one AND instruction every 36 bytes. We use this technique consistently across all unintended instruction pairs, even between cases where dead store elimination would not occur, so that the virtual PC is always incremented by the same amount.

Modified immediate constants

As we mentioned in Chapter 4 when we introduced this payload type, some ARM instructions' immediate fields are composed of a 4-bit right-rotate portion followed by an 8-bit value portion that allow for the encoding of various 32-bit values. The 8-bit value is placed in the least significant byte of a 32-bit value and right rotated $2\times$ the value of the rotate field. Some 32-bit values have more than one possible 12-bit encoding; the only valid encoding (i.e., the only one that a JIT compiler will produce for that 32-bit value) is the one with the smallest rotation value. Consider the case of encoding the value 0x00000001. It could theoretically be encoded as 0x210, meaning 0x10 right-rotated $2 \times 2 = 4$ bits. However, the only valid encoding is 0x001 because it has the smallest rotation amount. This is important because if we require an intended immediate-operand AND instruction (which uses this type of immediate encoding field) with a 12-bit immediate field containing the value 0x210, it is impossible to coerce IonMonkey to produce it because 0x210 encodes the 32-bit value 0x00000001, which must be encoded as 0x001. This in turn limits the unintended Thumb instructions that can be encoded with those bits.

The consequences of this encoding constraint can be observed throughout the payload. For example, when the `zero` and `non_zero` branches loop back to the `interpreter_loop_top` label, they each perform two subtractions from the virtual PC, the composition of which move it back to the top of the loop. This is necessary because a single Thumb instruction performing the loopback all at once requires an invalid immediate encoding that has a non-zero rotation amount and an 8-bit value whose least significant two bits are clear (meaning that the 8-bit value, if non-zero, could be right-shifted to reduce the rotation amount).

Advancing the virtual PC requires even more sophisticated trickery to avoid invalid immediate encodings. Adding 36 to the virtual PC register in a single instruction requires an invalid immediate encoding, but rather than splitting the instruction into two smaller additions each time the virtual PC needs to be advanced, which would require the addition of another branch block (block 1 does useful work; block 2 partially advances the virtual PC; block 3 finishes advancing the virtual PC; block 4 diverts the real `PC` to the now-advanced virtual PC), we populated `R7` with the virtual PC advancement amount in two steps (instruction 1 in branch block 2 and instruction 2 in branch block 1 in Table 7.1) and advance the virtual PC by adding that register to the virtual PC. The first time around the branch blocks, `R7` has not been fully populated yet and cannot be used to advance the virtual PC. We work around this in instruction 1 of branch block 1 by preloading the virtual PC with the address of the next instruction in the branch block. Instruction 1 of branch block 2 is thereby free to initialize `R7` instead of manipulating the virtual PC (notice that the task of all other instructions in branch block 2 is to update the virtual PC, `R6`).

Recall that in Thumb mode, the value of the `PC` register is the address of the currently-executing instruction plus 4. Therefore, in actuality, instruction 1 of branch block 1 prepopulates the virtual PC with the address of the instruction 40 bytes later rather than the standard virtual PC advancement amount of 36 bytes. This was necessary because the minimum distance between unintended instruction pairs is 36 bytes due to the sizes of the mutually-exclusive `else-if` blocks, but to encode instruction 1 of branch block 1 as `add r6, pc, #32` would require an invalid modified immediate constant in the overlapping intended ARM instruction. Adding 4 bytes between instructions 1 and 2 in each branch block requires the instruction to instead be `add r6, pc, #36`, which can be encoded. Note that this is the only place where we are able to have an inconsistent distance between unintended instructions in the same branch block because in all other cases the virtual PC advancements are effected by adding `R7` to the virtual PC.

7.2.3 Encoding a NOP sled

It is possible to construct a NOP sled using the same payload encoding method by placing $n+k$ branch blocks prior to the branch blocks containing the shellcode/OISC interpreter. The unintended instruction pairs in the initial n branch blocks exist only to direct control flow forward to the final k branch blocks; in these n branch blocks, the first unintended instruction in each pair is irrelevant. 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 achieve back-to-back 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. Therefore the NOP sled must be encoded using AND instructions operating on other registers, and density must be sacrificed by using mutually-exclusive `else-if` blocks as described earlier. The spacing of AND instructions for the SBNZ OISC payload is 36 bytes (9 ARM instructions); a similar spacing should be achievable for a NOP sled. However, the NOP sled's success probability is greater than $\frac{1}{9}$ because the ARM instruction before each AND instruction is a conditional branch instruction used to skip the block if its condition was not met; by carefully selecting the conditions of the `else-if` blocks, the attacker can ensure that the conditional branch instruction's Thumb decoding does not have undesirable side effects. Namely, the Thumb decoding of the conditional branch must allow execution to fall through into the Thumb instructions encoded by the following ARM AND instruction. Branch instructions are one of the few instructions that IonMonkey emits that will have this property. The probability of choosing a correct offset within the NOP sled is therefore $\frac{2}{9}$.

Although the probability of successfully guessing a valid offset within an ARM-to-Thumb NOP sled is $2/9$, the probability of landing in the NOP sled at all must be considered as well. Using the size of the unoptimized (21440 bytes) and optimized (4140 bytes) code generated for the SBNZ OISC payload, we estimate that for every byte of optimized code generated, about 5.2 bytes of corresponding unoptimized code will be generated. Because SpiderMonkey must be able to fall back to unoptimized code if speculative optimization fails, the unoptimized code will not be garbage collected even after optimized code is generated. Therefore the probability of successfully landing an attack using this particular payload on IonMonkey is approximately $4140/21440 \times \frac{2}{9} = 4.3\%$.

7.2.4 System calls

Unintended Thumb system call instructions can be encoded in intended ARM instructions that store their result into the stack pointer and use certain immediate operands. In particular, the 4-bit rotation portion of the 12-bit immediate field must be 1111_2; consequently, the immediate must decode to a value between 516 and 1020, inclusive.

Unfortunately, instructions that modify the stack pointer in conjunction with immediate operands of that order of magnitude are rarely emitted in IonMonkey JIT code. Those that do so are found in function prologues and epilogues to allocate and free the call frame. On the ARM architecture, IonMonkey call frames smaller than 1024 bytes are rounded up in size to the nearest power of 2, with a minimum call frame size of 128 bytes. Since there are no powers of 2 between 516 and 1020, inclusive, prologues and epilogues in IonMonkey JIT code will not be able to generate an intended ARM instruction that hides an unintended Thumb system call instruction.

The SBNZ OISC interpreter cannot perform system calls, and in general, we are not aware of a method that would allow an attacker to encode an unintended Thumb system call instruction using the payload-encoding method used to implement it. However, the SBNZ OISC interpreter serves merely as a minimal proof of concept of Turing-complete computation. More powerful programs could be constructed that read from memory to launch a code reuse attack against static code containing system calls.

7.2.5 Design shortcomings

The 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.

Chapter 7, in part, has been submitted for publication of the material as it may

appear in appear in *Proceedings of the 2017 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 8

Defensive Just-In-Time Code Emission on ARM

8.1 Introduction

Researchers and practitioners have proposed numerous mitigations against the unique security risks introduced by JIT compilers. The mitigations vary widely in their peformance overhead, difficulty of integration into an existing JIT compiler, and defensive effectiveness. These efforts fall roughly into the following three categories: capability confinement, memory protection, and diversification.

Despite the vast abundance of JIT spraying mitigations proposed in the literature, hardly any of them are deployed in production-quality JavaScript engines—which are arguably the most ubiquitous language runtimes that employ JIT compilation. Although some mitigation proposals and methodologies are incomplete or carry shortcomings that are prohibitive to their widespread deployment, others that do not share these faults remain unimplemented. An obvious culprit is the war for performance, perpetuated by browser vendors grappling for market share. JIT compilation is, after all, meant to be performance-enhancing technique; slowing down JIT compilation and code execution runs in opposition to its very purpose. It is estimated that a defense technique must incur no more than 10% runtime overhead to have even a chance at being deployed in a real system [55].

To make matters worse, the various performance figures presented by those who have developed and evaluted their own JIT spraying mitigations are not based on a consistent set of benchmarks or testing hardware. Consequently, JIT developers seeking to understand the relative costs of JIT spraying mitigation are unable to make meaningful comparisons between the reported overheads.

In this chapter, we survey and qualitatively evaluate the effectiveness of the state of the art in JIT spraying mitigations and shed light onto the current state of mitigation deployment. Additionally, we present and quantitatively analyze the effectiveness of our own open source implementations of several diversification defenses on SpiderMonkey for ARM and x86-64 and empirically evaluate their performance overhead using a consistent set of industry-standard benchmarks on fixed hardware platforms.

From our study of JIT spraying mitigations in the wild, we conclude that most mainstream JavaScript JITs are woefully underprotected against JIT spraying attacks, even 5 years after its public debut in [13]. We find that a full complement of diversification defenses can mitigate the threat posed by blind JIT spraying by drastically reducing the probability of successful code reuse at the cost of less than 5% runtime overhead.

8.2 Survey of Proposed JIT Spraying Mitigations

We structure our survey of proposed JIT spraying mitigations by partitioning the approaches into the following three classes: capability confinement, memory protection, and diversification.

8.2.1 Capability confinement

Capability confinement defenses seek to make JIT code an unattractive reuse target by reducing the set of capabilities that JIT code can possess. Concrete approaches range broadly in sophistication and effectiveness. Some seek to guard against a small number of specific instruction sequences, whereas more sophisticated proposals go so far as to sandbox both the JIT compiler and the code it creates.

Heuristic JIT spray detection

Piotr Bania [11] proposes a JIT "intrusion detection system" that analyzes the code produced by the JIT in order to detect a JIT spraying payload as it is being compiled so that the JIT compiler can halt emission before the spray becomes exploitable. Bania's IDS relies on the observation that early JIT spray payloads included long sequences of instructions that operated on 32-bit immediate values. It raises an alarm if the number of such instructions following an instruction of the form `mov reg, imm32`—but preceding a control flow transfer—exceeds a certain threshold. In effect, this system confines JIT code to contain only short sequences of immediate-operand instructions (i.e., attacker-provided 32-bit values). An obvious evasion is to spray short sequences of immediate operand instructions interrupted by branch instructions, having the last unintended instruction in each sequence jump forward into the next unintended instruction sequence. Bania's IDS watches for this by disassembling constant values to detect chained jumps.

This defense raises the bar for Blazakis-style JIT spraying payloads (e.g., `var x = 0x3c909090 ^...`) on x86 by reducing the effectiveness of NOP sleds by forcing attackers to periodically encode instructions for detection evasion, but it does little to curb the encoding of ROP gadgets and gadget chaining gadgets in JIT code. Since a self-sustaining ARM JIT spraying payload constructed from chained immediate-operand instructions has yet to be discovered, there is no evidence that this mitigation is valuable for ARM.

Bania's disassembly strategy only begins counting for immediate-operand instructions after encountering a 32-bit value being moved into a register. Bania explains that this improves performance by avoiding exhaustive disassembly of emitted code; however it leaves the IDS vulnerable to a chain of instructions that accumulate operations against a program variable (e.g., `var x = y ^0x3c909090 ^...`). In this case, one would expect the instruction chain to begin with a memory value rather than an immediate being loaded into the accumulator register. Bania does not report the performance overhead of this proposed defense.

JITSec

JITSec [24] confines JIT code by adding runtime protections against system calls originating from the stack and heap regions in memory. JITSec is guided by the assumption that any system call invoked from dynamically-allocated code is evidence of an attack underway. JITSec is implemented as a kernel module that intercepts system calls and inspects their callsites before deciding whether to pass them on to the normal system call handler or to terminate the process.

As one might speculate, JITSec's overhead is more noticeable for simple system calls; for example `getpid` experiences a 11.71% reduction in throughput compared to 0.25% for `clone`. The SPEC CPU2000 Integer benchmark showed a 1.84% average overhead.

Unfortunately, while JITSec will prevent an attacker from executing syscalls from JIT code, it does not prevent JIT sprayed code from launching a code reuse attack by branching into statically-compiled code which then invokes the desired system call from a valid callsite.

NaCl-JIT

NaCl-JIT [6] is a system that extends the Native Client (NaCl) sandbox [62] 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.
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 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.

NaCl uses runtime checks to ensure that indirect memory accesses and control flow transfers do not break out of the sandbox. A check must be inserted before each instruction that could potentially escape the sandbox; and together, each runtime check and the instruction it guards are referred to as a *pseudo instruction*. Runtime checks and the code layout described below ensure that normal and pseudo instruction can only be executed from their first byte, eliminating the threat of unintended instructions.

Untrusted code must be laid out in a special way to support NaCl. The smallest unit of untrusted code allocation is called a *code region*, which is comprised of one or more 32-byte *instruction bundles*. Instruction bundles are aligned to 32-byte boundaries, and code is arranged in such a way that no instruction or pseudo instruction straddles an instruction bundle boundary. This provides the invariant that any branch targeting the beginning of an instruction bundle will execute an intended instruction along with all necessary runtime checks. The runtime checks for indirect branches take advantage of this by clearing the least significant 5 bits of the branch target address to force it into 32-byte alignment. To facilitate the proper alignment of subroutine returns, each call instruction is padded forward so that the instruction following it will be the first instruction or pseudo instruction in the next instruction bundle. Unused bytes preceding call instructions and at the ends of instruction bundles are filled with NOP instructions.

Untrusted code compiled to be run in a NaCl sandbox is already laid out in bundles with runtime checks in place. Before loading the code into the sandbox's executable memory region, NaCl validates it to check that code is properly laid out and that runtime checks are correct and present where needed. The validator also checks the instructions against a whitelist and ensures that direct branches stay within the sandbox and do not target the middle of an instruction or pseudo instruction.

For the sake of brevity, we have not exhaustively specified the behavior and constraints of a NaCl-validated binary. However, the overall effect is that sandboxed code that is executed from an intended instruction boundary then left to its own devices will never run unintended instructions or execute code outside the sandbox (except via an API call into the NaCl runtime); nor will it read or write outside of its own data region. Even if an attacker were able to commandeer the JIT compiler and dictate every byte of JIT code it emitted (which would need to pass NaCl validation in order to execute), she would not be able to escape the sandbox. This level of capability confinement comes with great performance cost, as shown in Table 8.1. The NaCl-JIT authors ported the V8 JavaScript Engine and the Mono Common Language Runtime to run in NaCl-JIT and evaluated their results on both x86-32 and x86-64 platforms running Ubuntu 10.04. The Mono JIT on x86-32 performed admirably under NaCl-JIT with just a 2% slowdown; but all other options—even the same JIT on x86-64—suffered at least 20% slowdowns.

Table 8.1. NaCl-JIT Slowdown Percentages

	x86-32	x86-64
V8 JIT (V8 benchmark v. 6)	28%	51%
V8 JIT (SunSpider100)	32%	60%
Mono JIT (SciMark C#)	2%	21%

RockJIT

RockJIT [42] offers a platform similar to NaCl-JIT with considerable performance improvements due to more efficient JIT code validation and its use of control flow integrity (CFI) [3] rather than code bundling to protect control flow. Like NaCl-JIT, RockJIT sandboxes the language runtime and its JIT code. Since the runtime is statically-compiled, and its control flow graph can be generated and analyzed offline, it is subjected to a fine-grained CFI policy (without a shadow stack). JIT code, on the other hand, is protected by coarse-grained CFI (meaning that any indirect branch site can target any indirect branch target). The decision to enforce coarse-grained rather than fine-grained CFI on JIT code improves the speed of dynamic code generation because computing and dynamically-updating a coarse-grained CFI policy is less computationally intensive than its fine-grained counterpart.

The sandboxed language runtime is expected to emit JIT code that contains runtime checks similar to those found in NaCl-verified code. Memory write targets are masked to confine them to the bottom 4GB of memory (assuming a 64-bit system), and indirect branches are instrumented to check their branch targets against the CFI policy and to mask their targets to 4-byte boundaries. Accordingly, instructions in JIT code that may be indirect branch targets are expected to be NOP-padded to 4-byte alignment. In the case of call-return pairs, the call instruction is padded forward until the subsequent instruction is properly aligned. Note that memory reads are not confined by runtime checks.

RockJIT provides sandboxed code a read-execute (RX) virtual memory mapping of the JIT code region while keeping a separate read-write (RW) mapping to the same physical pages for itself. The sandboxed RX mapping resides in the bottom 4GB of memory; whereas the RW mapping resides outside of the sandbox. This ensures that a guarded write in JIT code cannot corrupt the JIT code region. The language runtime installs code by calling into the RockJIT API and providing (1) a buffer of code to be installed; (2) the address in the RX mapping at which to install the code; and (3) a list of indirect branch targets in the new JIT code. RockJIT validates that system call instructions are not present, that all necessary runtime checks are installed, and that direct branches do not escape the sandbox or allow for the circumvention of runtime checks. If validation passes, RockJIT copies the code into the RW mapping at an address computed as a fixed offset from the requested installation address in the RX mapping; afterwards, it updates the CFI metadata so that indirect branches may target the newly-installed code.

In RockJIT's threat model, the language runtime is considered benign but potentially buggy. The language runtime is therefore allowed to make system calls and expose an interface through which JIT code may indirectly invoke them. The language runtime cannot issue system calls that manage memory mappings and memory protection bits, but it may indirectly request those operations via a RockJIT API call.

RockJIT's authors ported the V8 JavaScript engine to use RockJIT; and on the same set of benchmarks over which NaCl-JIT incurred 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, since RockJIT allows the (untrusted) language runtime to directly make system calls (excluding memory reprotection), and RockJIT's fine-grained CFI implementation does not use a shadow stack to ensure return address integrity, it is vulnerable to an attack called control flow bending [16]. Control flow bending attacks abuse the lack of a shadow stack to weave control flow through a fine-grained CFI control flow graph by taking legal, but un-paired, return edges until a system call is reached.

In general, sandboxing defenses such as NaCl-JIT and RockJIT do a good job of protecting the system outside the sandbox from bugs in sandboxed code. However, they fail to protect the language runtime against malicious JITed code. For example, if an attacker is able to compromise the sandbox, she could read from the language runtime's memory to steal cookies across web origins—assuming process-based origin isolation a la Chrome/Chromium is not in use. Furthermore, exploitation of bugs outside the sandbox is not mitigated at all. This means, for example, that a control flow vulnerability in NaCl-JIT or RockJIT is able to target any address in JIT code, even unintended instruction boundaries. On the other hand, it could be argued that most exploitable bugs occur in the language runtimes themselves (or the programs embedding them) and that a sandboxing environment represent a much smaller, easier-to-verify attack surface.

8.2.2 Memory protection

Perhaps one of the most glaring threats created by the introduction of JIT compilers is the omisson of W $\oplus$ X protection on JIT code regions. Unlike most statically-compiled code (e.g., C/C++ binaries and libraries), which is typically marked readable and executable (RX), JIT-emitted code, especially for dynamically-typed languages, is often permanently marked readable, writable, and executable (RWX). Permanent RWX permissions eliminate the memory reprotection system calls that would be necessary to support the performance-critical technique known as inline caching, which can require frequent modification of JIT code.

Inline caching is a technique initially developed for the Smalltalk programming language [26] that offers improved execution speeds for dynamic and dynamically-typed languages. In a dynamic language, object properties may be added on the fly; therefore the offset within an object of any particular fixed property name may differ across objects. Furthermore, a variable in a dynamically-typed language may be backed by any type, so the behavior of an operator taking a variable as an operand cannot be known until the runtime type of the variable is determined (e.g., in JavaScript, the expression '`x + 3`' evaluates to a number if `x` is a number and a string otherwise). Normally, accessing a property or using a variable as an operand requires a call into the runtime to inspect runtime types and dispatch the proper pre-compiled handler, a process which is typically expensive. Inline caching employs self-modifying code to save fast path code for recently-seen runtime types at each dynamic access site.

The intuition behind inline caching is that each variable access site in the source code tends to encounter values of the same primitive type or dynamic "class,"[1] and furthermore, if a property of a variable is being accessed at a particular location in code, it tends to refer to the same property of the same dynamic class. For example, consider the function shown in Listing 8.1, which access the same property of every element in an array and adds it to an accumulator variable. A reasonable argument to this function is an array of objects—all constructed in a common manner—whose `value` property holds an integer value. For such an input, accessing `e.value` could be optimized down to a short code stub containing a few machine instructions that read from a fixed memory offset in `e` into a register rather than an expensive call into the JavaScript runtime that accomplishes the same task less efficiently after searching through `e`'s type information. Similarly, the addition operation could be optimized down to an efficient code stub that performs integer addition over two registers rather than calling a JavaScript runtime routine that checks the runtime types of the operands and decides how to add them.

[1]For non-primitives, a common notion of a class in JavaScript implementations is any object with the same property names that were defined in the same order.

```
function (elements) {
  var s = 0;
  for (var i = 0; i < elements.length; i++) {
    e = elements[i];
    s = s + e.value;
  }
  return s;
}
```

Listing 8.1. Sample JavaScript function demonstrating access patterns that could benefit from inline caching.

Inline caches (ICs) are efficient runtime type checks that are placed at each potential optimization site. Whenever the code encounters an object whose runtime type matches the type checked for by the IC (IC hit), the optimized code stub is executed. If the type check fails (an IC miss), the slow path (through the language runtime) is executed; a fast path code stub optimized for the new type is generated; and the IC is updated to check for the new type and call the replacement stub on IC hit.

In 1991, Hölzle et al. [30] extended inline caching to support “polymorphic” sites that encounter a small number of types and/or properties and use the type information stored in the ICs to implement *adaptive optimization*, allowing JIT compilers to generate highly-specialized code based on information recorded in polymorphic ICs. Polymorphic inline caching and adaptive compilation have been widely deployed in all major browsers’ implementations of JavaScript [54, 38, 2], resulting in considerable performance gains. For example, V8 performance incurs a 12$\times$ slowdown when inline caching mechanisms are disabled [6].

RWX memory poses a significant threat to JIT security. Our gadget chaining proof of concept attacks against JavaScriptCore (Section 5.2) and V8 (Section 6.2) both leverage always-RWX JIT code pages to corrupt the JIT code buffer and install arbitrary shellcode. Memory protection defenses seek to rid JIT code of its always-RWX memory protection status by adapting W $\oplus$ X to dynamically-generated code. The two primary approaches to memory protection for JIT code are transient protection and dual mapping.

Transient protection

Transient protection defenses minimize the time during which memory has an undesirable protection status. Both JITDefender [18] and JITSafe [19] attempt to parry JIT code reuse by marking JIT code pages as non-executable upon completion of compilation and re-enabling the execution bit only before the language runtime calls into JIT code. Once the call into JIT code returns to the runtime, the execution bit is disabled. If the attacker were to trigger a control flow vulnerability to branch to the JIT code pages outside of this window, the segmentation fault would be raised, terminating the attack.

JITSafe and JITDefender never disable writeability of JIT code. Therefore, while JIT code is executing, it is marked RWX. This enables IC updates to proceed efficiently without the overhead of calling `mprotect` and clearing the TLB each time. As a result, JITDefender's transient protection overheads were 0.9%, 0.5%, and 0.1% for Tamarin, V8, and JavaScriptCore respectively (Windows 7, x86-32). JITSafe's transient protection component yields similarly minimal overheads of 0.5%, 0.5%, and 0.9% for Tamarin, V8, and JavaScriptCore, respectively (Windows 7, x86-32). In addition to transient execution protection, JITSafe also employs diversification techniques.

A shortcoming of JITSafe and JITDefender is the large window of opportunity for exploitation while JIT code is executing. 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.

JITScope [63] is a proposed defense that mitigates the threat of JIT code corrupting itself. JITScope wraps all control flow transfers from the language runtime into JIT code with a function which sets the JIT code region's protection bits to RX before branching to it; the wrapper does not remove the executable bit when control returns from JIT code. JITScope funnels all writes to the JIT code region (including IC updates) through a wrapper function which first sets the JIT code region to RW before invoking the desired write function. Once writing is complete, and control flow has returned to the wrapper, the JIT code region is reprotected to read-only (non-writable, non-executable). No benign path through the control flow graph leaves the JIT code in a state that is both writable and executable.

JITScope does not interfere with the predictability of emitted JIT code; therefore if there were a control flow vulnerability, it could divert control to and unintended entry point in the JIT code in the usual Blazakis-style JIT spraying fashion. To address this threat, JITScope protects all statically-compiled code with fine-grained control flow integrity (CFI) for forward edges and a shadow stack for backward edges, in addition to coarse-grained CFI (with a shadow stack) on JIT code.

Since JITScope must make a system call to modify memory protection bits each time the JIT code undergoes modification (e.g., to update an IC), its overhead is predictably much higher than JITSafe and JITDefender, which only modify protections when calling and returning from JIT code. JITScope benchmarks for the SpiderMonkey JavaScript engine on Ubuntu 12.04 for x86-64 indicate a 4.26% overhead for memory protection alone and 9.51% overhead for the entire system when CFI is enabled as well. However, we believe these performance numbers should be taken with a grain of salt, as a recent study has found that a traditional shadow stack implementation alone incurs a 9.69% CPU overhead [23].

In 2011, the SpiderMonkey team experimented with transient protections similar to those proposed by JITScope with overheads in the 2%–3.5% range on Windows 7 and Linux for (unspecified 32- or 64-bit) x86 [41]. For the SunSpider benchmark, against which both JITScope and the SpiderMonkey team's experimental work were

Table 8.2. Dual-mapping defenses and overheads

	Overhead %	Weakness
Single-process dual mapping	1%	Info leak
Lobotomy	387% (27% @ 95th %ile)	Code corruption
SDCG	5.6%-16.6%	Entire compiler trusted

evaluated, JITScope's W $\oplus$ X enforcement incurred $\sim$3.8% overhead (Ubuntu 12.04, x86-64), which is comparable to the 3.5% overhead for the SpiderMonkey team's work (unspecified Linux, unspecified 32- or 64-bit x86). After the initial push, transient write protection was not enabled in SpiderMonkey due to the performance cost and uncertainty concerning its security benefits.[2]

These concerns were justified when Song et al. [52] demonstrated the use of Web Workers [56] to launch a multi-threaded attack that circumvents transient memory protections by creating a race condition. The web workers specification enables the main JavaScript thread to spawn a separate thread in the same process that executes JavaScript and can communicate with the main thread via the `postMessage` interface. To launch the attack, the main thread instructs a worker thread to cause its JIT code region to become writable (e.g., by inducing JIT compilation of a new function) and simultaneously exploits a memory corruption bug to write to the writable region during the window of opportunity. Although this attack presents a code corruption threat, it could be adapted to pose a code reuse threat against temporarily-executable JIT sprayed code under the protection of JITDefender and JITSafe.

Dual mapping

As we saw in Section 8.2.2, varying JIT code's writability and executability over time is vulnerable to race condition attacks. Furthermore, toggling between RW and RX

[2]More recently, the SpiderMonkey team changed their mind and pushed transient protection (non-writable JIT code by default) into production [39].

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 [27] in order to thwart code injection attacks that attempt to grant executability to shellcode stored in a data buffer.

Prior to the publicization of JIT spraying [13], the SpiderMonkey team investigated an alternative to transient protection in which JIT code is allocated in physical memory pages that are mapped to two virtual address ranges within the same process [40]. One mapping would have RX permissions and would be used for executing JIT code, and the other would have RW permissions and would be used for code emission, IC patching, garbage collection, etc. The single-process dual mapping scheme, which incurred about a 1% slowdown on x86-32 and x86-64 on both Linux and OS X, was never adopted, in part due to doubts regarding its security.

SpiderMonkey never adopted the single-process dual mapping method because it can be circumvented with the aid of an information leak, which would allow an attacker to locate both the RW and RX mappings. RockJIT implements single-process dual-mapping but mitigates the abovementioned threat by confining memory writes in JIT code to the bottom 4GB of memory and placing the RW mapping at a higher address, safely out of reach of a guarded write.

Other research efforts address the shortcomings of single-process dual mapping by separating the RX and RW mappings of JIT code across process boundaries. An untrusted process will hold either the RX or RW mapping to JIT code residing in shared memory, and a trusted process will hold the other mapping. Whenever the untrusted process needs to perform an action requiring the trusted process's permissions to the JIT code, it will request it through some form of inter-process communication (IPC). If the untrusted process becomes compromised by an attacker, it can only directly have either write or execute access to the JIT code region.

Whether the untrusted process holds the RW or the RX mapping depends on the defense's motivating threat model. Lobotomy [33] addresses the code reuse threat and therefore allows only the trusted process to have the RX mapping of the JIT code. Consequently, an attacker who is exercising only one bug–a control flow vulnerability in the untrusted process–will not be able to branch directly into the JIT code region.

A prototype of Lobotomy was implemented for Firefox's now-deprecated tracing JIT, TraceMonkey, and evaluated by simulating user interaction with several popular websites via browser automation tools. Lobotomy incurred a 387% mean slowdown over the unmodified JIT, compared to a mean slowdown of 160% when the JIT was disabled. The authors cite corner cases as the source of Lobotomy's poor performance and also report the slowdown at the 95th percentile, which is only 27% for Lobotomy compared to 586% with the JIT disabled.

SDCG [52] responds to the threat of code corruption and takes the approach opposite to Lobotomy by implementing multi-process dual mapping with a trusted writing process. SDCG is designed around the assumption that the attacker can write arbitrarily to memory. Since an arbitrary memory write can be used to divert control flow to an attacker-controlled address, and the untrusted process holds an RX mapping to JIT code, the threat of JIT spraying must be addressed. SDCG assumes that JIT code diversification mechanisms (Section 8.2.3) are in place to reduce the likelihood that branching into a JIT sprayed payload will result in successful shellcode execution. To prevent the attacker from launching a code reuse attack that leverages non-JIT code in order to reprotect JIT memory as writable, SDCG interposes itself on virtual memory management system calls and blocks such requests.

The performance cost of this defense comes from RPC and cache coherency overheads. Table 8.3 shows the drop in V8 benchmark scores incurred by implementing SDCG on top of V8 revision 16619. There is a notable performance drop on both x86-32

Table 8.3. V8 benchmark score reductions for SDCG

	Threads pinned	Free scheduling
x86-32	6.9%	15.9%
x86-64	5.6%	16.6%

and x86-64 when trusted and untrusted threads are allowed to be scheduled on different physical cores.

8.2.3 Diversification mechanisms

The cornerstone of all attacks that reuse JIT-emitted code for malicious purposes [13, 48, 49, 44, 35, 9] 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.

Since many proposed defenses that include diversification share common mechanisms, we first describe high-level diversification mechanisms in order to establish a vocabulary for discussion. In the next subsection, we will exercise this vocabulary and delve into specific proposals in the literature that apply these techniques.

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.

Register randomization Register randomization permutes register assignment during compilation, resulting in instructions whose register operands are unpredictable [58, 59, 61] 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. For example, in the Thumb-2 instruction set, most 16-bit instruction encodings only support register operands `R0`-`R7`. Register randomization could cause higher-numbered registers to be used when they otherwise would not, resulting in the need for 32-bit Thumb instructions instead of 16-bit instructions.

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 [58, 44, 59, 61, 19, 31] 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. As long as the attacker does not know the secret value that is used, she cannot predict any of the immediate operands that will appear in the final instruction stream and therefore cannot rely on them to encode malicious instructions.

Listing 8.2 shows an example of blinding a bitwise XOR instruction on x86. By composing two instructions of the same operation, blinding can be carried out with only one additional instruction. However, consider the example in Listing 8.3, in which an

```
; Unblinded
xor   eax, 0xabababab

; Blinded
; Operand 1: 0x193da98c (secret)
; Operand 2: 0xabababab ^ 0x193da98c = 0xb2960227
xor   eax, 0x193da98c
xor   eax, 0xb2960227
```

Listing 8.2. x86 assembly demonstrating constant blinding of a bitwise XOR instruction.

```
; Unblinded
add   eax, 0xabababab

; Blinded
; Operand 1: 0x193da98c (secret random value)
; Operand 2: 0xabababab ^ 0x193da98c = 0xb2960227
mov   ebx, 0x193da98c
xor   ebx, 0xb2960227
add   eax, ebx
```

Listing 8.3. x86 assembly demonstrating the addition of the untrusted constant `0xabababab` to a register with and without constant blinding.

untrusted constant is placed into a register in two steps, and the original immediate-operand instruction is converted to a register-operand instruction. In addition to requiring one more instruction, this method requires a spare register to hold the untrusted constant. This method is preferred to composing two `add` instructions because the `add` instruction will update the processor flags. In optimized JIT code, it is common for the compiler to test for overflows and underflows after addition, subtraction, multiplication, and division. Those conditions can indicate that the operation's result cannot be represented as a 32-bit integer and that deoptimization to a floating point representation is needed; therefore flag-mutating arithmetic instructions should not be used when operating on constant blinding operands to prevent unwarranted deoptimization.

Even bitwise AND and bitwise OR instructions should be converted to register-operand instructions rather than composed because set and unset bits in the untrusted constants can result in predictable bits in the blinding operands. In particular, for

```
; Unblinded
add   R0, #2880154539  ; 0xabababab

; Blinded
; Operand 1: 0x193da98c (secret random value)
; Operand 2: 0xabababab ^ 0x193da98c = 0xb2960227
movw  R1, #43404  ; 0x193da98c & 0xffff
movt  R1, #6461   ; 0x193da98c >> 16
movw  R2, #551    ; 0xb2960227 & 0xffff
movt  R2, #45718  ; 0xb2960227 >> 16
eor   R1, R2
add   R0, R1
```

Listing 8.4. ARM assembly demonstrating the addition of the untrusted constant `0xabababab` to a register with and without constant blinding.

bitwise AND, each 1-bit in the untrusted constant forces corresponding bits in both blinding constants to be set; for bitwise OR, each 0-bit in the untrusted constant forces corresponding bits in both blinding constants to be unset. This can be most clearly observed by considering the corner cases where all bits in the untrusted constant are either set or unset. The instruction

```
and   eax, 0xffffffff
```

can only be blinded with composed `and` instructions as

```
and   eax, 0xffffffff
and   eax, 0xffffffff
```

Similarly, blinding the bitwise OR of 0x00000000 with composed `or` instructions requires blinding operands of 0x000000 and 0x00000000.

The limited number of immediate bits available in ARM and Thumb instructions forces most instructions to be converted to register-operand instructions in order to perform constant blinding. Moreover, two scratch registers rather than one are needed to hold the blinding operands, as shown in Listing 8.4. Note that in this case, the unblinded instruction could be encoded as an immediate-operand instruction thanks to Thumb immediate encoding rules that allow a 32-bit constant composed of the same 8-bit pattern replicated 4 times to be encoded in a 12-bit field.

As one might expect, the overhead of constant blinding comes in the form of both increased code footprint and increased execution time. Athanasakis et al. [9] estimate that constant blinding can result in up to 80% additional instructions.

Call frame randomization Call frame randomization [58, 59] 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 showed in our proof of concept gadget chaining attack against V8 (Section 6.2), can provide an attacker with a predictable memory access instruction that can be conveniently repurposed for memory corruption.

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.

Random NOP insertion Random NOP insertion [31, 32, 58, 59, 44, 61] 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.

Table 8.4. Table of the mechanisms and overheads of various concrete proposals containing diversification defenses

	Overhead %	Constant blinding	Register randomization	Call frame randomization	Random NOP insertion	Base offset randomization
INSeRT	2%	✓	✓	✓	✓	
RIM	< 1%	✓	✓		✓	
JITSafe	2.80%	✓	✓		✓	
librando	15%–250%	✓			✓	
Adaptive Diversification	5%-6%				✓	

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.

Base offset randomization Base offset randomization [44] 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 [53] used to pinpoint gadget addresses in our proof of concept gadget chaining attacks described in Chapters 5.2 and 6.2. Base offset randomization would have drastically reduced the reliability of those attacks with substantially less overhead than random NOP insertion.

8.2.4 Concrete diversification proposals

In this subsection, we will discuss five concrete defense system proposals from the literature. Table 8.4 provides an overview of these proposals with their overheads and the diversification mechanisms they include.

INSeRT

Wei et al.'s INstruction Space Randomization & Trapping (INSeRT) [58, 59] defense targets x86 and performs constant blinding of all immediates, register randomization, call frame randomization, and a variant of random NOP insertion. Their objective is to randomize all instruction operands as well as give a process under attack a chance to detect certain failed exploitation attempts if they do not crash the process.

Rather than inserting semantic NOPs, INSeRT's variant of random NOP insertion randomly injects what its authors call *trapping snippets*. A trapping snippet is a sequence of instructions that starts with a short, forward PC-relative branch, followed by a small number of breakpoint instructions (the x86 implementation uses `int 3`, which is typically used by debuggers and can be encoded in a single byte as `0xCC`). If execution begins at the first byte of the trapping snippet, as would be the case in benign execution, the relative branch will skip over the rest of the trapping snippet. However, if execution begins anywhere else in the snippet (for example, due to an attack underway), one of the breakpoint instructions will execute, and the interrupt can be handled by the process under protection, which can react accordingly.

The frequency of trapping snippet injection influences code size in the expected manner, where more frequent injection results in greater memory overhead, and can be fine tuned to balance security with memory usage. The authors of INSeRT implemented a prototype for V8 running on Windows 7 on x86; their average overhead across five runs of the SunSpider 0.9.1 benchmark showed only a 2% slowdown with trapping snippet injection tuned to 5.9% memory overhead.

The only design description Wei et al. provide for call frame randomization is to reorder "function arguments, local variables, and so on" in such a way that there are at least 256 potential offsets. The authors do not make any statements regarding whether or

```
; Original
xor  eax, 0xabababab

; Diversified with RIM
; 0xabababab ^ 0x22222222 = 0x89898989
push  ecx  ; ecx chosen at random
xor   ecx, ecx
xor   ecx, 0x89892222
nop   ; inserted with 50% probability
xor   ecx, 0x22228989
xor   eax, ecx
pop   ecx
```

Listing 8.5. x86 assembly demonstrating RIM performing constant blinding with random scratch register selection, immediate splitting, and random NOP insertion.

not values of different types should be intermixed or not.

RIM

Wu et al.'s Removing IMmediate (RIM) [61] defense applies constant blinding along with weakened forms of register randomization and random NOP insertion to immediate-operand bitwise XOR instructions. RIM selects a random register to use as a scratch register and converts the immediate-operand XOR instruction to a register-operand that uses the scratch register instead of the untrusted constant. The 32-bit untrusted constant and a randomly-chosen 32-bit secret value are bitwise XORed with each other to form a scrambled value, and both the secret and scrambled values are split into 16-bit halves. The upper half of the scrambled value and the lower half of the secret are concatenated and moved into the scratch register. A NOP instruction is emitted with 50% probability, then the remaining halves are concatenated and XORed against the scratch register. Listing 8.5 shows an example of RIM diversification in x86 assembly.

The prototype of RIM that its authors implemented and evaluated only applied diversification to the immediate operands of XOR instructions. Their assessment that RIM's diversification methods incur less than a 1% overhead for Tamarin on Windows XP

SP3 should be evalated with that in mind. If RIM were applied to all immediate-operand instructions, the overhead would be much higher, especially because of the pushes and pops needed to save and restore the scratch register. It should also be noted that RIM does not implement true register randomization or random NOP insertion. The only register that is randomized is the scratch register, and the only potential NOP insertion sites are between the pairs of immediate-operand XOR instructions.

JITSafe

JITSafe [19], which also deploys transient memory protection (Section 8.2.2), performs diversification of immediate-operand arithmetic and bitwise instructions using weakened variants of random NOP insertion, constant blinding, and register randomization. JITSafe stores untrusted constants in heap memory. It then replaces instructions that would have used them as immediate operands with a memory load into a random register followed by the register-operand version of the original instruction. A NOP instruction of random length is randomly placed between the load instruction and the new register-operand instruction to break apart any potential unintended instructions that might straddle the load and arithmetic/bitwise instruction.

This solution offers less entropy than true constant blinding since some of the bits in the memory addresses that are encoded into the instruction stream may be shared or predictable. The authors of JITSafe evaluated its performance overheads on Tamarin for Windows 7. The overhead of replacing the immediate-operand instruction with a memory load and register-operand instruction is 1.6%. Random register and random NOP insertion incurs an overhead of 0.7%, and the overall performance overhead of JITSafe, with both transient memory protection and diversification enabled, is 2.8%.

librando

Homescu et al. [31] set their sights on the challenge of deploying diversification defenses to JIT engines without having to replicate engineering effort across each engine. Librando is a runtime library that applies constant blinding and random NOP insertion to the JIT code emitted by any x86 JIT compiler without requiring modifications to the compiler.

Librando employs lazy diversification; it prevents undiversified code emitted by the JIT compiler from executing and only diversifies it once it is executed for the first time. To do this, librando intercepts calls to virtual memory allocation functions (e.g., `mmap`) and removes execute permissions from the allocated memory. The JIT engine using librando writes undiversified JIT code to the non-executable memory region as usual. Since the JIT code is non-executable, whenever execution branches to it—even under benign conditions—a segmentation fault is raised, triggering execution of a segmentation fault handler provided by librando.

If librando's signal handler encounters a piece of code it has not seen before, it begins the process of diversification. Librando disassembles the undiversified code, using the segfaulting address as the starting point and following direct branches to traverse the control flow graph. Each basic block is diversified and tracked by librando independently of other basic blocks in the control flow graph. A copy of the code with diversification defenses applied is written to a separate memory region and given RX memory permissions, and once diversification is complete, the segmentation fault handler branches to the diversified code.

To support and detect modification and garbage collection of JIT code, librando keeps the undiversified code in place and marks its memory pages as read-only. Any attempt to modify a read-only page will invoke a librando segmentation fault handler which will enable write permission on the page and tag all blocks in the page as dirty. The next time the segmentation fault handler catches an execution attempt for a dirty block, librando hashes all dirty blocks and checks a hash table to determine which have been modified and therefore require re-diversification. After re-diversification of dirty blocks, all blocks are returned to the read-only state.

Librando implements random NOP insertion in the following manner. For each instruction in a basic block, librando decides whether or not to insert a random NOP before it with 50% probability. If it does insert a NOP, it chooses randomly from a 1-byte, 2-byte, or 3-byte NOP.

Constant blinding is implemented for the following instructions with 32- and 64-bit immediate fields: `mov`, `push`, `imul`, `test`, and arithmetic instructions. Instead of using bitwise XOR to blind values, librando uses composed addition and subtraction via the `lea` instruction, which does not set the processor flags. The authors provide the example of loading an immediate into a register by loading the values (`constant` − `secret`) and `secret` into separate registers then using the `lea` instruction to add them. They do not show how they blind other types of instructions; but they do state that they "implemented different blinding code manually for each type of instruction." This may indicate that either 1) other instruction types were converted to their register-operand variants and prefixed with a `mov` that is blinded as described above; or 2) they use other methods for non-addition instructions and only used the `lea` method for the `mov` instruction. It would be unwise to blind addition and subtraction instructions by composition with `lea` because it could lead to masking real under/overflows or create unwarranted under/overflows. Consider the example shown in Listing 8.6; if the initial value of `eax` is `0x10` or greater, the addition of the constant value `0xfffffff0` should cause the overflow flag to be set, and under no circumstances should the underflow flag be set. However, in the diversified code, the underflow flag could be set if the initial value of `eax` is `0xe` or less, and the

```
; Undiversified
add eax, 0xfffffff0

; Diversified
push ebx
sub  eax, 0xf
mov  ebx, 0xffffffff
lea eax, [eax + ebx]
pop ebx
```

Listing 8.6. Demonstration of diversified code in which the `lea` instruction can mask overflow conditions and create an unwarranted underflow when used to blind addition by composition.

overflow flag cannot be set.

In addition to inserting random nops and applying constant blinding, librando modifies direct and indirect branches. Direct branches are rewritten to target diversified code so that unnecessary segmentation fault handling is avoided. In order to make the runtime stack appear as though execution were occuring in the undiversified code, librando converts `CALL` instructions into an instruction sequence that pushes the undiversified address of the instruction following the `CALL` instruction then jumps to the callee's diversified address. Consequently, subroutine returns will target undiversified addresses. To accelerate indirect branches, all of which target undiversified addresses, librando provides a lookup table from the addresses of undiversified block entry points and call return sites to their diversified counterparts and rewrites indirect branches to check the lookup table first.

The authors evaluated the performance impact of librando on the V8 JavaScript engine and the HotSpot Java JIT on an x86 Linux machine. They evaluated librando on V8 using the V8 benchmark suite and HotSpot using the Computer Language Shootout Game benchmarks. Table 8.5 shows the slowdown percentages, which indicate that the majority of the overhead incurred comes from juggling the separate copies of JIT code. However, diversification still contributed substantial overhead.

Table 8.5. Librando slowdown percentages

	Rewriting only	Full diversification
V8	170%	250%
HotSpot	8%	15%

When librando executes in "black box" mode, the JIT compiler whose code is being diversified never directly calls librando. We found that this operating mode has a severe flaw under certain circumstances. Librando learns basic block boundaries within JIT code by starting at an undiversified address that was the target of a branch into undiversified code and disassembling forward from that point. An illicit branch (caused by a control flow vulnerability) targeting what would be a ROP gadget or an unintended instruction stream in a JIT spray payload will cause librando to disassemble, diversify (an operation which preserves functional semantics), and execute the malicious instruction stream.

If librando's signal handler tracks and matches entire ranges of undiversified code that have already been diversified, it could detect that an illegal branch target has been used, as long as librando may assume that an indirect branch into the middle of a basic block is abnormal. Such a lookup would require either a slower or less space efficient lookup structure than the hash map described by librando's authors and would still not protect a JIT compiler that eagerly compiles functions before they are ever invoked legitimately.

Librando has a "white box" version in which diversification of code starting at a particular address is requested by the JIT engine via an API call into librando, obviating librando's handling of segmentation faults. The diversification API call returns the address of the diversified code, which the JIT engine uses to execute it. This closes the vulnerability we raised above but creates a attractive target for a return-to-libc-style attack.

Adaptive JIT Code Diversification

Jangda et al. [32] offer a diversification strategy which periodically rediversifies JIT code and adaptively reduces the diversification overhead in “hot” code. The objective of rediversification is to cause a time-of-check time-of-use condition that favors the defender. If the attacker is attempting to use an information leak vulnerability to defeat code diversification, by the time the attacker diverts control to her payload, the code may have already been replaced. If the attacker’s information leak vulnerability is slow or carries a high risk of crashing the victim process, she may miss her window of opportunity or crash the process trying to monitor the state of JIT code.

Adaptive overhead reduction capitalizes on rediversification to optimize the performance of diversified code where it stands to have the most impact. Adaptive JIT code diversification injects a randomly-chosen semantic NOP before each instruction with a probability inversely proportional to the relative hotness of the enclosing function. The objective is not to increase the number of random NOPs in cold functions, but rather to decrease the number of random NOPs in hot functions. The performance of very hot functions is further optimized by excluding execution counters in functions whose execution count has exceeded some threshold. Very hot functions are not exempt from rediversification and random NOP insertion; they share in having the lowest probability of random NOP insertion before any particular instruction.

Periodic rediversification occurs in a separate thread, so it does not block normal execution of JIT code. The length of the interval between rediversifications is drawn uniformly at random from 0 up to some maximum sleep time. During each round of rediversification, functions are recompiled in order of decreasing hotness so that optimizations can take effect sooner in hotter functions.

A prototype of adaptive JIT code diversification was implemented on the Jikes

Research Virtual Machine, an open source Java Virtual Machine written in Java. The prototype was evaluated on an x86-64 machine running Fedora 17 using the DaCapo 2009 Benchmarks. Through experimentation, Jangda et al. arrived at the conclusion that a max sleep time of 1000 seconds and a threshold of 1000 for considering a function "very hot" provide the best performance. The remainder of their performance evaluation uses those parameters. The performance results that they report speak well to the value of reducing the NOP insertion probability in hot code. For non-adaptive NOP insertion probabilities of 50% and 30%, overheads were 10% and 9%, respectively. With adaptive NOP insertion probability ranges of 10-50% (10% for the hottest functions and 50% for the coldest) and 0-30%, the overheads were 6% and 5%, respectively.

The maximum sleep time of 1000 seconds is problematic for security, as the average time between rediversifications is over 8 minutes. For comparison, Snow et al.'s Just-In-Time code reuse attack [51], which constructs a ROP payload by using a memory leak to read one byte of memory at a time, is able to exploit Internet Explorer 8 in under 25 seconds. With a max sleep time of 1000 seconds, fewer than 1 in 40 recompilations are expected to expire before Just-In-Time code reuse could construct an exploit for IE8. The adaptively inserted NOPs could make finding enough useful gadgets more difficult, but the attacker can improve her chances by executing her spray functions enough times for them to be classified as very hot in order to minimize the number of inserted NOPs. Ironically, increasing the frequency of rediversification can actually be a boon to a Just-In-TIme code reuse attacker, as it gives the attacker more opportunities for all the gadgets she needs to appear in JIT code simultaneously during a given period of time. In order to properly defend against Just-In-Time code reuse attacks, a max sleep time of less than a minute is probably necessary.

8.3 State of Mitigation Deployment

Although there is an enormous quantity of work concerned with defensive JIT code emission, performance concerns prevent all but a very limited set of ideas from the research community from making their way into the systems we use on a daily basis. Even then, many implementations make unacceptable security tradeoffs to reduce performance overhead. In this section, we provide a critical analysis of the defensive JIT code emission practices deployed by the following four open-source JavaScript engines: JavaScriptCore, V8, SpiderMonkey, and Chakra. The lack—and shortcomings—of JIT spraying mitigations in all of these JavaScript engines except Chakra sustain our argument that the current state of JIT spraying mitigation deployment needs to be revisited.

8.3.1 JavaScriptCore

As of the version embedded in WebkitGTK version 2.2.2-1 port for Debian, JavaScriptCore deploys random NOP insertion and constant blinding. However, their implementations fall short due to either a mis-estimation of how JIT spraying can manifest itself on the ARM architecture or a reluctance to sacrifice performance in exchange for robust coverage.

Random NOP Insertion

JavaScriptCore's non-optimizing Baseline JIT performs a rudimentary form of random NOP insertion. Instead of inserting NOP instructions randomly throughout the emitted code, it emits a single NOP instruction at the beginning of each compiled function with 50% probability on a per-function basis. The semantic NOP used is always the 16-bit Thumb NOP instruction. The optimizing DFG JIT does not insert random NOP instructions under any circumstances.

First and foremost, the lack of random NOP insertion in the DFG JIT is a major oversight. Optimized code looks and behaves more like ahead-of-time compiled code and affords the attacker far greater control over executable memory than unoptimized code, whose function is typically to shuffle values around between the stack and registers before calling into a stub which performs the actual computation. In fact, all three of our attacks presented in this dissertation relied on code generated by the JavaScript engines' optimizing compilers, and Blazakis' [13] original attack—although it targeted the ActionScript JIT—could only be adapted to JavaScriptCore if it were to target the DFG JIT.

Suppose the Baseline JIT's random NOP insertion were deployed on the DFG JIT. A Blazakis-style x86 self-sustaining payload would of course have no trouble in the face of a single NOP randomly inserted at the beginning of the function. We argue that it could be defeated on the ARM architecture with a gadget chaining attack that includes a read gadget encoded in the second half of an intended 32-bit Thumb instruction, so long as the intended return sequence in the sprayed function can be used to correctly return to the gadget's caller. For example, if the attacker is able to corrupt the code pointer for a JIT-compiled function's entry point, the gadget could be invoked via a JavaScript-to-JavaScript call rather than a JavaScript-to-C call; consequently, the intended JavaScript return in the gadget could be used instead of having to write a new return instruction as was necessary in the gadget chaining attacks described in Sections 5.2 and 6.2.

Suppose the attacker sprays her read gadget along with any others needed by her attack and assumes that a random NOP will not be inserted when guessing the address of the read gadget. She then invokes her read gadget and passes the address of the read gadget itself as the memory location to read. If a NOP was not inserted, the read gadget will return its own encoding. If a NOP was inserted, the chosen address will not point to the unintended read instruction, and the read gadget will fail to return its own encoding. In either case, the attacker learns whether or not a NOP was inserted and can adjust

Table 8.6. Blinded operations and blinding methods used in JavaScriptCore. Operations listed with more than one method may be blinded differently depending on the context.

Operation	Blinding methods
Addition	addition, XOR
Subtraction	addition, XOR
Multiplication	XOR
Register assignment	rotation, XOR
Memory store	rotation, XOR
`int` to `double` cast	XOR
branch	XOR
bitwise AND	AND, XOR
bitwise OR	OR, XOR
bitwise XOR	XOR

Table 8.7. Conditions under which JSC will never blind a constant value $\mathscr{V}$.

Condition
$\mathscr{V}$ == 0xffff
$\mathscr{V}$ == 0xffffff
$\mathscr{V} \leq$ 0xff
$\mathscr{V} \geq$ 0xffffff00

her address guess accordingly. Since the NOP is only inserted at the beginning of each function, other gadgets sprayed in the same function will reside at a predictable offset from the unintended read instruction.

The Baseline JIT's random NOP insertion even fails to impact the relative offset between the entry points of sequentially-compiled functions because JSC rounds up allocation sizes to the nearest 32 bytes. If the attacker crafts her payload in such away that its size does not fall near a multiple of 32, rounding will absorb any increase in the size due to the single random NOP.

Constant blinding

JavaScriptCore performs constant blinding for operations for all the operations shown in Table 8.6, but it does not blind all constants. Certain constants are considered “safe”; Table 8.7 lists the conditions under which a constant value $\mathscr{V}$ will never be blinded; if any one of the conditions is met, the constant is ineligible for blinding. Furthermore, each eligible constant constant is only blinded with $1/64$ probability.

Occasional constant blinding is perhaps suitable for preventing JIT spraying on x86, since the canonical attack was to chain together long sequences of immediate-operand instructions. Randomly disrupting approximately 1 in 64 of these operands might be enough to prevent the attack from succeeding at minimal expense to performance. Using gadget chaining, however, very few immediates are needed to form the desired gadgets. Therefore, a higher constant blinding rate is needed. Moreover, the “safe” values that JSC does not blind are not necessarily safe on ARM. The `R2`-disclosure gadget we presented in Section 4.5 uses 16 as a constant, which is considered “safe” by JSC. In order to defend against JIT spraying, *every* constant needs to be blinded.

In addition to the canonical XOR blinding, JavaScriptCore uses rotation blinding for some operations. Rotation blinding moves a version of the untrusted constant that has been bitwise rotated a random number of bits to the right into a register, then rotates that register to restore the untrusted constant’s value. Compared to the XOR blinding method, which provides 32 bits of entropy, the rotation blinding method only provides $log_2 32 = 5$ bits of entropy.

JavaScriptCore uses composed addition blinding under some conditions to blind addition and subtraction instructions and takes no measures to prevent unwarranted overflows. It also uses AND and OR blinding to blind bitwise AND and OR instructions, respectively. As we warned in Section 8.2.3, AND and OR blinding produce blinding operands with predictable set and unset bits, respectively.

8.3.2 V8

Memory protection

V8 does not implement any of the memory protection defenses discussed in Section 8.2.2, but recent work by the V8 team takes the first steps towards eliminating the need to make JIT code writable by introducing non-patching ICs. In V8's original implementation of polymorphic inline caching, each property access site calls a dynamically-generated stub which essentially implements a switch statement that tail calls to the appropriate fast path handler on IC hit or calls into the runtime on IC miss. The stub and fast path handlers are dynamically generated and allocated in RWX memory. Whenever there is an IC miss, a new fast path handler is compiled, as is a new version of the stub that includes a branch in the switch statement for the new handler. The call to the old stub at the property access site is patched to point to the new stub's address.

V8's non-patching ICs do not encode previously-observed types and properties in stub code. Instead, this information is stored in a data structure called a *type feedback vector*. Type feedback vectors are allocated in non-executable memory and are accessed by generic stubs which then tail call to dynamically-generated handlers. Generic stubs are dynamically-generated, but each instance of the runtime only needs one copy of the generic stub for each type of non-patching IC; they can be shared across different property access sites of the same type and even across functions. Since each property access site utilizing a non-patching IC will always call the same generic stub, it is not necessary for the stub-calling code at the access site to be writable for patching.

Currently, the only types of non-patching ICs that are implemented in V8 are for loads (i.e., `obj.x`), keyed loads (i.e., `obj[x]`), stores, keyed stores, and function calls. Through communication with members of the V8 team, we have learned that non-patching ICs are between 1.5× (monomorphic sites) and 3× (polymorphic sites) slower than their patching predecessors; however these performance regressions have been offset by other work which creates more opportunities to compile code with V8's optimizing JIT, Crankshaft.

Partial deployment of non-patching ICs, unfortunately, does not eliminate the need for JIT code memory to be writable after compilation. Other IC types such as those for binary operations and comparisons appear frequently and necessitate modifiable JIT code. However, due to performance concerns, JIT vendors may be unwilling to part with patching ICs. For example, the V8 team has explained to us that binary operation ICs are currently considered too performance-sensitive to be converted to non-patching ICs. Moreover, static pre-compilation of fast path handlers is challenging, because their contents sometimes rely on detailed knowledge of the runtime state of objects' prototype chains.

Diversification

V8 only implements constant blinding; and it is only enabled for x86-32 and x86-64. V8's constant blinding implementation does not blind untrusted immediate operands used in all instructions; it only blinds constants >0x1ffff that are being moved into a register or pushed onto the stack. Arithmetic and bitwise instructions—even bitwise XOR, the instruction used in Blazakis' original JIT spray attack—are not protected by constant blinding.

Because V8's non-optimizing Full Codegen JIT always implements binary operations on untrusted constants as a move followed by a call to an IC stub (e.g., `mov eax, 3c909090h, call BINOP_IC`), all untrusted constants in unoptimized code are blinded, and it is protected from Blazakis-style JIT spraying payloads on x86. Crankshaft, V8's optimizing JIT, on the other hand, uses adaptive optimization to emit binary operation instructions that operate directly on untrusted immediate operands (e.g., `xor ebx,` `3c909090h`). Since V8 never blinds untrusted constant operands to binary operation instructions, Crankshaft-emitted code remains completely vulnerable to JIT spraying as described by Blazakis' 2010 attack against the x86 [13].

V8 for the ARM architecture does not implement any diversification defenses. This is most likely because JIT spraying was only seen as a threat against the x86 architecture.

8.3.3 SpiderMonkey

For a long time, SpiderMonkey has not deployed JIT spraying mitigations. In the past, the SpiderMonkey team developed experimental mitigation patches [41] based on recommendations by Rohlf and Ivnitsky [44], but those patches were never deployed to production due to concerns that the mitigations lacked security value commensurate with their performance costs. Recently, the SpiderMonkey team changed their mind on one defense and launched transient memory protection. Each unit of JIT code compilation is normally RX, and whenever it needs to be modified, it is temporarily reprotected RW. The SpiderMonkey team reports runtime overhead $< 1\%$ 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 transient memory protection because Baseline code does not require code patching in the event of an IC miss. New handlers are added to IC call sites by appending a pointer to a list in non-executable heap memory rather than patching or recompiling an executable IC stub. The first handler in the list for each call site is invoked via an indirect branch; this indirect branch never needs patching. Each handler checks whether the runtime types of its inputs match, and if they do not, it finds the next handler in the list and branches to it. If the end of the list is reached, the handler calls the slow path in the runtime. Although

IonMonkey's ICs require patching, they are less prevalent than Baseline ICs.

The addition of transient memory protection is a step in the right direction, but it is still vulnerable to a race condition attack [52]. Moreover, the lack of diversification defenses and default RX protection status leaves SpiderMonkey's JIT code vulnerable to JIT spraying attacks that do not rely on code corruption.

8.3.4 Chakra

The Chakra JavaScript engine is used by the Microsoft Edge web browser (the replacement to Microsoft Internet Explorer). Chakra was initially closed-source, but Microsoft recently opened the source code of its core components under the name Chakra-Core. Chakra implements random NOP insertion and large integer constant blinding for x86-32, x86-64, and ARM. Chakra decides when to insert NOPs by generating a counter and decrementing it after every instruction. When the counter reaches zero, a NOP is inserted, and the counter is re-generated. Fresh counter values are drawn uniformly at random from 1 to n, where n defaults to 8.

Chakra's random NOP insertion implementation also randomizes the location of the first instruction in each function by randomly inserting up to 15 NOP instructions before it. Although this mechanism could be labeled as base offset randomization, it is unlikely to be a sufficient implementation thereof due to allocation size rounding. We do not have intimate knowledge of Chakra's executable memory allocator, but if it is like SpiderMonkey's or V8's, it rounds up the size of each unit of code compilation to some granularity (e.g., 32 bytes). Since a NOP instruction can be as short as a single byte, the increased size of functions due to this form of base offset randomization can be hidden by the allocator's size rounding.

Chakra blinds addresses, memory access displacements, and integer constants that it considers "large" using bitwise XOR. Chakra uses the term "large," but it is somewhat of a misnomer. What Chakra really blinds are information-rich values. Chakra does not blind 64-bit constant integers, which would be considered large by most definitions of *large*. A value is *not* considered "large" if and only if its upper or lower 16 bits (after being cast to a 32-bit integer) are all 0's or all 1's. As we mentioned in our examination of JSC's constant blinding implementation, which also ignores information-poor constants, even small constants can be useful to an attacker. However, in combination with code layout randomization (i.e., random NOP insertion), their utility to a blind attacker is considerably diminished.

Notable exceptions to Chakra's constant blinding policy are 64-bit integers, floating point nubers, call/branch offsets, and call frame metadata. The first two exceptions leave Chakra vulnerable to the obvious Blazakis-style JIT spraying payload, and as shown by Maisuradze, Backes and Rossow [36], unblinded PC-relative call and branch offsets can be used by a non-blind attacker to encode ROP gadgets. Call frame metadata presents a unique threat that, to our knowledge, has yet to be exploited in the literature. Function callers must convey to the callee the number of actual arguments provided in a call. Consequently, at each call site, the number of arguments (up to a Chakra-chosen maximum of 2^{24}) appears as part of a 32-bit immediate in the JIT code. An attacker could exploit her control of these immediate bits by compiling function calls with large argument counts. A shortcoming of this method is that since only 24 bits of the immediate are needed to encode the argument count, the upper 8 bits of the immediate are used for other metadata not easily manipulated by the attacker.

Of the JavaScript engines we have examined, Chakra's JIT hardening defenses are the most robust. Its random NOP insertion and constant blinding implementations cut far fewer corners than JSC's and V8's. Helpfully, the source code for Chakra's defenses is consolidated into a single `Security` class, and the design of the backend lends itself much better to easily integrating random NOP insertion than SpiderMonkey's.

We base this assessment on our experiences implementing random NOP insertion for SpiderMonkey on both x86-64 and 32-bit ARM, which we will describe in the next section.

8.4 Understanding the costs and benefits of diversification mitigations

As we saw in Section 8.3, 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 their effectiveness has been nullified. We argue that blind JIT code reuse can be effectively mitigated via a suite of fully-functional diversification defenses whose overhead, while modest, is less than that of capability confinement and effective memory protection defenses. However, the questions of how much performance overhead diversification defenses incur and to what extent they improve security have not been answered clearly in the literature. Various incarnations of the diversification mitigations described in Section 8.2.3 are mixed and matched to compose a multitude different defense systems mentioned in the literature. Many of these implementations are not fully specified, and what descriptions exist often vary considerably from author to author. 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 Section 8.2.3 on the SpiderMonkey JavaScript engine for both its non-optimizing (Baseline) and optimizing (IonMonkey) JIT compilers on the ARM32 and x86-64 architectures.[3] Our implementations are by no means highly optimized; instead, our priority is to avoid cutting corners and 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`.[4]

8.4.1 Implementations

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 operand or result registers which cannot be randomized at the level of the register allocator; how-

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

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

ever, 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. The instruction sequences implementing bytecodes are defined by C++ code that allocates abstract registers as operand, result, and temporary registers used by each bytecode’s implementation. To the C++ programmer, using an abstract register “feels” like using a physical register, but each one is simply a variable named after the corresponding physical register that holds an integer value identifying the physical register.

Randomizing registers for Baseline JIT code (and indirectly, some of IonMonkey’s code) involves permuting the underlying values that are assigned to the abstract register variables. 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 quite a few headaches. In these corner cases, violating these assumptions via randomization leads to incorrect behavior and data corruption that 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 well documented. 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 move them into floating point registers before performing 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 $(n-k)!/n!$.

Constant blinding

All of the constant blinding implementations we observed in real-world code only blind a 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 attacker-provided (meaning that it appears in the JavaScript code) integer and floating point constant. Attacker-provided 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[5] 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 called LIR, then finally compiled to native code. During lowering from bytecode to MIR, we tag numeric constants found in the bytecode as untrusted and only blind tagged constants. After the MIR has been optimized, we replace the untrusted constants that remain with a sequence of MIR instructions that implement

[5]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.

constant blinding by loading a variable in two steps via bitwise XOR (in the same manner as described above for Baseline JIT's constant blinding) then redirecting uses of the original untrusted constant to the constant blinding-loaded variable. 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 cope with the limited size of immediate operand fields in the ARM instruction set, IonMonkey places some constant values inline with JIT code and emits PC-relative load instructions that move them into registers when they are needed as operands. Multiple values are buffered and flushed into contiguous regions of memory, earning them the name *constant pools*. In 2011, Pete Beck [12] presented an attack against the Tamarin ActionScript engine ARM that exploited predictable constant pools populated with attacker-chosen floating point values. Without constant blinding, IonMonkey is also vulnerable to Beck's attack. Our constant blinding implementation blinds floating point numbers at the IR level with the bitwise XOR operation in the same way it blinds integers and has the side effect of blinding floating point values in constant pools. Because IonMonkey did not previously support computing the bitwise XOR of floating point values, we extended its implementations of MIR and LIR with instructions representing the bitwise XOR of double-precision floating point values and implemented the corresponding backend native code generation for both 32-bit ARM and x86-64.

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}.

Limitations We only blind constants that appeared in the JavaScript source code. Implicit values generated by the compiler are not blinded. As we showed in our discussion of an attacker's vectors for influencing JIT code in Section 4.2, PC-relative branch offsets, which never appear in high level language code, can be manipulated by changing the sizes of basic blocks. Random NOP insertion indirectly diversifies branch offsets by perturbing the sizes of blocks, but the range of potential block sizes is a function of the undiversified block size and is far smaller than the 32-bit space offered by constant blinding.

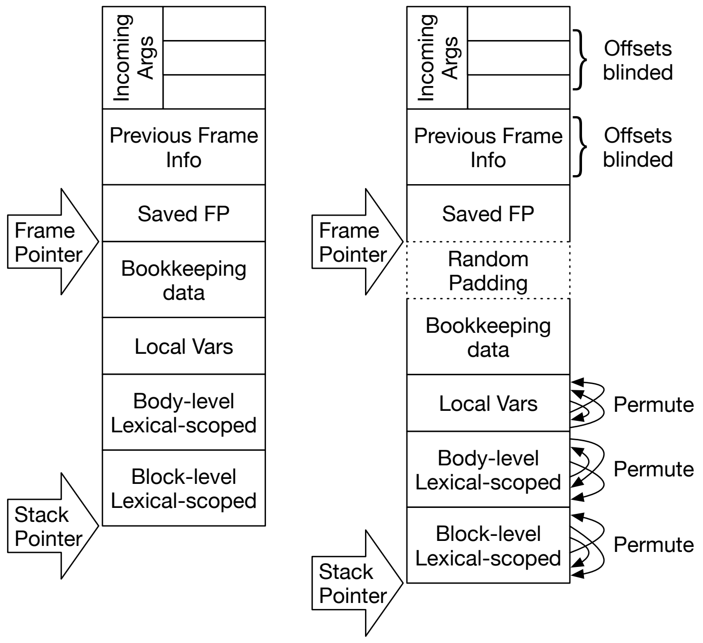
Maisuradze, Backes, and Rossow [36], who also demonstrated the construction of ROP gadgets with branch offsets on x86, proposed a branch offset randomization technique in which the branch target address is loaded into a register via two partial additions, and the PC-relative branch is replaced with a register-indirect branch. In particular, they add the values (offset & secret) and (offset & $\sim$ secret). The observant reader may recognize that this blinding method suffers from the same problem as composing bitwise ANDs to blind an AND instruction. Namely, bitwise AND only has the ability to randomly unset bits in the untrusted value that were already set to begin with; it will never randomly set a bit that was unset. Consequently, untrusted values with more unset bits are more likely to appear as desired by the attacker. A more secure version of branch offset blinding could be implemented using bitwise XOR to load the offset into a register in two steps before adding that register to the PC and performing a register indirect branch.

Another example of an implicit value under attacker control that may appear in JIT code is the argument count for a function call. SpiderMonkey pushes stack arguments, followed by the number of stack arguments onto the stack when making a function call. Since the number of stack arguments pushed at each function callsite is fixed at compile time, the count value is hardcoded into the callsite and is entirely under the control of the attacker. There is no IR-level object that represents the argument count; the argument count push is simply emitted as part of lowering the LIR call instruction to native code. Consequently, it is not possible to blind this value in an architecture-independent manner while traversing the MIR graph.

Under the threat model we defined in Section 2, the risk posed by undiversified implicit constants is significantly diminished when other diversification defenses are in place. This is largely because implicit constants are more difficult for an attacker to place close to one another than explicit constants because implicit constants make up only a small part of a high-level operation such as setting up a call or performing a conditional branch. Their low density makes them less useful for self-sustaining payloads in the presence of random NOP insertion, which will increase the difficulty of jumping from one instruction encoded by an implicit constant to the next; and code layout randomization diversification defenses make gadget chaining more difficult by disrupting the heap feng shui needed to execute gadgets blindly.

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



(a) Unrandomized **(b)** Randomized

Figure 8.1. Baseline JIT call stacks with and without call frame randomization. Stacks grow downward.

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.

Baseline JIT Figure 8.1a 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 8.1b 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 $[0, 64)$[6] and performs the access using the scratch register and a new offset that corrects for the shifted base register value. For example, the instruction

```
ldr r0, [fp, #12]   ; r0 = Mem[fp + 12]
```

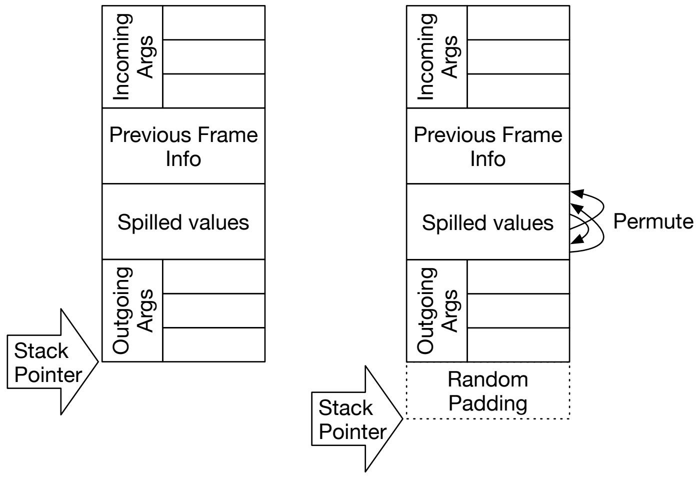
might be replaced with the following two instructions:

```
sub lr, fp, #48     ; lr = fp - 48
ldr r0, [lr, #60]   ; r0 = Mem[fp - 48 + 60]
```

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 difficult. Similarly, we do not permute incoming arguments on the stack because of the complex interactions between their stack positions and SpiderMonkey's implementation 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 in 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 or *Previous Frame Info* element 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

[6]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

Figure 8.2. IonMonkey call stacks with and without call frame randomization. Stacks grow downward.

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, and m is the number of possible padding amounts ($m = 16$ for our implementation).

IonMonkey JIT Figure 8.2a 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 8.2b 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.

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 (e.g., when emitting a branch table). To aid in identifying these situations, we took advantage of the fact that constant pools, which can normally be flushed into the instruction stream at any time, are not allowed to be flushed while the JIT code's layout is expected to be statically-predictable. We added a function which allows us to determine whether or not constant pools are allowed at any point during code emission and only emit random NOPs when they are not disabled.

The design of SpiderMonkey made implementing random NOP insertion non-trivial. Even at the level of LIR, the lowest-level, architecture-dependent IR, it is not possible to insert random NOPs between every machine instruction by injecting LIR NOPs. This is because even LIR instructions represent high-level actions that may be expanded into multiple machine instructions.

To achieve potential random NOP insertion between every machine instruction, we instrumented the low-level backend functions that write instruction encodings into the code buffer. For ARM, SpiderMonkey only has two such functions; adding fine-grained random NOP insertion was trivial once these functions were identified.

For x86-64, however, there are more than a handful of instruction emission functions. There are 26 opcode emission functions in the x86 backend; we instrumented all of them to randomly insert a NOP before emitting the opcode with $\frac{1}{8}$ probability. Although this potentially allows every instruction's opcode to be preceded by a NOP, this behavior is not completely correct because some x86 instructions have a prefix before their opcode. Inserting a NOP between the prefix and opcode interrupts the intended instruction. To remedy this, we temporarily disable random NOP insertion in the 79 instruction emission functions that emit an insruction prefix.

As we mentioned in Section 8.3, the design of Microsoft Chakra's code emission backend simplifies the task of implementing random NOP insertion. Instructions in Chakra's IR are stored in a doubly-linked list, and as they are lowered towards native code, new lower-level nodes are inserted into the list until each final machine instruction corresponds to at least one node in the linked list. Once that stage has been reached, but before the nodes are lowered to machine code, a random NOP insertion subroutine can traverse the linked list and insert NOPs. Since NOPs are inserted before final translation to machine code, Chakra's random NOP insertion implementation can be used by all architecures.

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 8.3.

The attacker might abandon guessing the precise offset of a useful instruction in

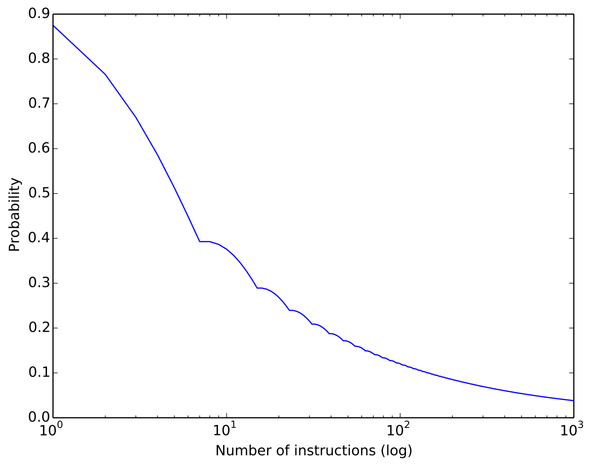


Figure 8.3. Probability that attacker's best guess correctly predicts the number of random NOPs inserted in n instructions, when the probability of random NOP insertion between each instruction is $\frac{1}{8}$.

JIT code in favor of spamming that instruction throughout its unit of code compilation and trying to jump randomly to it. Under these circumstances, the attacker's success probability against undiversified code is $\frac{1}{b}$, where b is the number of valid instruction boundaries between each instance of the useful instruction. For example on x86, b is the number of bytes between them; in Thumb mode on ARM, it is the number of bytes divided by 2, etc. Against random NOP insertion, the attacker's success probability is reduced to $\frac{1}{b+np}$ in expectation, where n is the number of intended instruction boundaries (i.e., potential NOP insertion sites) between each instance of the useful instruction. This probability assumes the inserted NOP instructions have the same length as the minimum instruction length; in other words, each inserted NOP is assumed to only add a single valid instruction boundary.

Lastly, 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 [13]), which occurs with probability $(1-p)^{n-1} = \left(\frac{7}{8}\right)^{n-1}$.

Because the only added benefit of Wei et al.'s INSeRT's trapping snippets [58, 59]

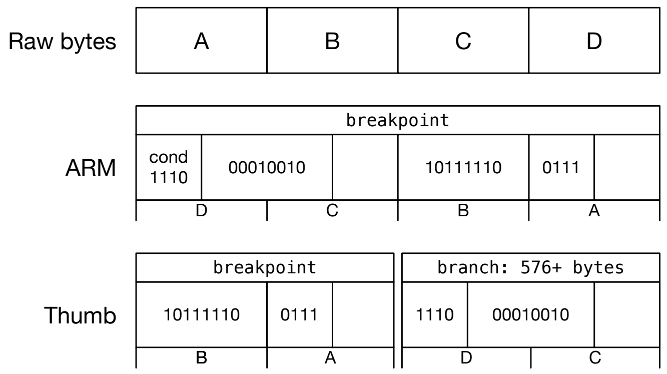


Figure 8.4. Diagram of breakpoint instruction encodings as they might appear in an INSeRT-style trapping snippet on ARM.

over normal random NOP insertion is enabling the protected process to catch exploitation attempts that branch into the middle of a trapping snippet (creating an intrusion detection system, we did not implement or measure them. Designing a trapping snippet for the ARM architecture requires careful attention to detail due to the potential for a control flow vulnerability that enables the attacker to execute the trapping snippet in an unintended instruction set mode. After the initial instruction that branches over the remainder of the trapping snippet, the x86 trapping snippet contains back-to-back 1-byte breakpoint instructions. On the ARM architecture, this is a poor design because it requires the snippet to encode many breakpoints when executed in both ARM and Thumb execution modes. Figure 8.4 shows that it is possible to encode a Thumb breakpoint instruction in the lower half of an ARM breakpoint instruction; but because ARM breakpoint instructions must have an Always condition code (1110_2), the upper half of the ARM breakpoint instruction must contain the prefix 111000010010_2, which encodes a Thumb PC-relative branch instruction of at least 576 bytes.

If an attacker's control flow vulnerability lands in the middle of a trapping snippet composed of instructions as shown in Figure 8.4 while in ARM mode, a breakpoint instruction will execute, and the IDS will be tripped. However, if the attacker lands at a random address in the trapping snippet in Thumb mode, she has only a 50% chance

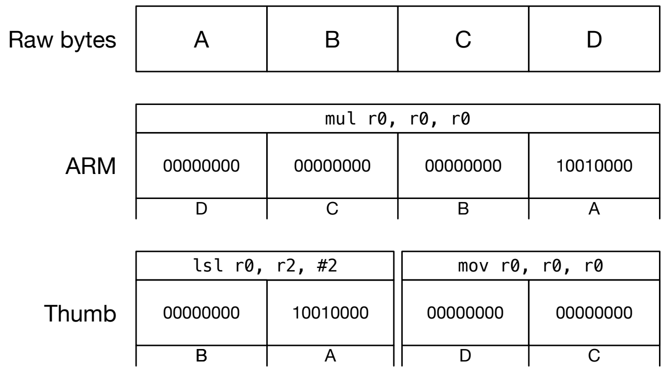


Figure 8.5. Diagram of pseudo NOP instructions that can be used to construct a pseudo NOP sled in an ARM/Thumb trapping snippet.

of landing on a breakpoint instruction. Since trapping snippets are usually shorter than 576 bytes, landing instead on the 576+ byte branch will skip the rest of the trapping snippet without tripping the IDS. If the attacker can control the least-significant two bits of the control flow vulnerability's target address, she can ensure that she never lands on a Thumb breakpoint instruction by taking advantage of the fact that the breakpoint instruction is formed by the lower 2 bytes of a 4-byte aligned ARM instruction. "Safe" addresses for the attacker have least significant two bits equal to 10_2, which guarantee landing on a branch in a trapping snippet.

Rather than attempting to fill the trapping snippet with breakpoint instructions, one could instead construct a NOP sled that leads into a breakpoint instruction. The encoding of the NOP sled need only avoid encoding instructions that cause control flow to skip the breakpoint instruction. Figure 8.5 illustrates a simple ARM instruction encoding and its corresponding Thumb decoding that can be used to construct a pseudo NOP sled which can be followed by the breakpoint encoding from Figure 8.4. These instructions do not form a true NOP sled, as they have side effects. This may be undesirable if the IDS intends to reconstruct the state of the machine. The `R0` register will not necessarily contain its true value at the time the control flow vulnerability was triggered. Those who wish to implement trapping snippets for ARM should consider finding a NOP sled encoding that does not clobber the machine state. The NOP sled approach has a second disadvantage that the IDS cannot identify the precise address in the snippet that the control flow vulnerability targeted since the breakpoint will always come from the end of the trapping snippet.

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 the allocation's header by rG where $r \in \{0,1,2,...,15\}$. This has the effect of increasing the size of the allocation request 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 $\left(\frac{1}{16}\right)^n$.

8.4.2 Evaluation

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 SpiderMon-

Table 8.8. Performance overheads for diversification defenses. 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.

(a) Performance overhead of diversification defenses on x86-64

	x86-64			
	SunSpider	Kraken	Octane	G. Mean
Register randomization*	-1.94%	**-0.829%**	-0.404%	-1.06%
Constant blinding	-1.36%	**2.65%**	**2.93%**	1.39%
Call frame frandomization	-1.68%	-0.199%	0.324%	-0.523%
Random NOP insertion	-0.762%	**2.12%**	**1.44%**	0.922%
Base offset randomization	-2.38%	-0.207%	**-0.846%**	-1.15%
All	1.71%	**5.70%**	**6.33%**	4.56%

(b) Performance overhead of diversification defenses on 32-bit ARM

	32-bit ARM			
	SunSpider	Kraken	Octane	G. Mean
Register randomization	**1.62%**	**0.456%**	0.265%	0.777%
Constant blinding	**1.62%**	**6.02%**	**4.39%**	3.99%
Call frame frandomization	0.138%	**-2.26%**	**-1.05%**	-1.06%
Random NOP insertion	**1.67%**	**1.76%**	**1.35%**	1.59%
Base offset randomization	0.498%	**0.302%**	0.223%	0.341%
All	**4.44%**	**5.48%**	**4.71%**	4.88%

key 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

Table 8.9. 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%

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 8.8. 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.

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 a time unit of code compilation is compiled. We executed the Sunspider, Kraken, and Octane 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 one or more diversification defenses; let $\bar{u}_i$ represent the average code size for the *i*th 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 8.9 shows the code size increases for each mitigation, broken down by architecture.

Our measurements indicate that most diversification defenses can be deployed at full strength with only modest 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 $\approx$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%.

A shortcoming of these measurements is that we did not have a reasonable method to measure the increased stack usage incurred by random padding introduced for call frame randomization.

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. In particular, we analyzed Blazakis' original JIT spraying and the three attacks presented in this dissertation. The results are shown in Table 8.10. 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 adapted each attack optimally. Lastly, the values shown in Table 8.10 only reflect the relative success rate of blindly executing 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 Blazakis attack and our JSC and SpiderMonkey attacks, 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 by the ARM V8 gadget chaining attack.

We were surprised to find that register randomization provided little defense against Blazakis' attack. The reason it is able to fare well against register randomization is because the attacker's payload is an unintended instruction stream 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

Table 8.10. Estimated relative success probabilities for concrete attacks against single diversification defenses. Lower values indicate better mitigation.

(a) Intra-instruction randomization

	Register randomization	Constant blinding	Call frame randomization
Blazakis 2010 [13] (x86-32)	92.9%	6.84e−47%	100%
JSC gadget chaining (ARM)	1.79%	1.53e−3%	100%
V8 gadget chaining (ARM)	$0.9\overline{09}\%$	100%	6.25%
SpiderMonkey self-sustaining payload (ARM)	2.51e−5%	4.15e−461%	100%

(b) Code layout randomization

	Random NOP insertion	Base offset randomization
Blazakis 2010 (x86-32) [13]	30.1%	100%
JSC gadget chaining (ARM)	3.74%	0.391%
V8 gadget chaining (ARM)	3.70%	0.229%
SpiderMonkey self-sustaining payload (ARM)	7.95e−21%	100%

opcode is either 1 byte long when XORing against `%eax` or 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 SpiderMonkey self-sustaining ARM payload as well as diversifying the locations of useful code gadgets used by the gadget chaining attacks. Base offset randomization, on the other hand, only defends against the latter pair of attacks by interfering with heap feng shui. 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 instructions[7] 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.

Parameter selection

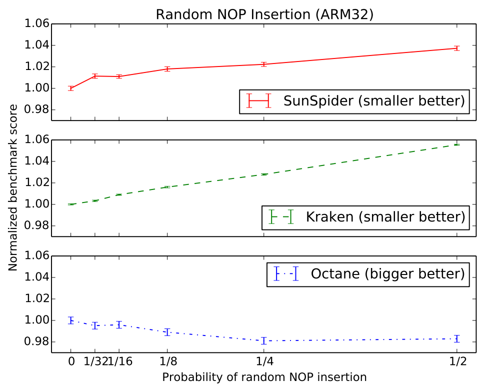
Here we consider the performance impact of varying randomization parameters for random NOP insertion, call frame randomization, and base offset randomization. Our findings may be beneficial to developers deploying and fine-tuning these defenses.

We evaluated each parameter value by executing the modified binary 100 times for each benchmark and taking the arithmetic mean of the 100 benchmark scores. Each arithmetic mean is normalized by dividing by the arithmetic mean of 100 executions of an unmodified binary on the same benchmark. In order to consistently normalize Octane, whose scores are interpreted as "bigger-is-better," we do not invert the ratio as we did previously to compute overheads. We plot normalized mean scores along with error bars representing the normalized standard error of the mean as a function of the randomization parameter. The value at $x = 0$ on each plot represents the unmodified binary; hence its normalized benchmark score of 1.

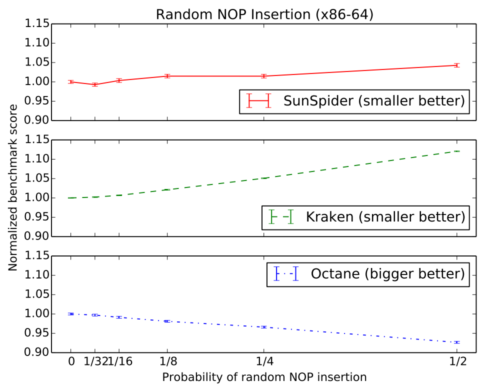
Random NOP insertion We varied the probability of inserting a NOP at each instruction boundary by selecting negative powers of 2 from $\frac{1}{32}$ through $\frac{1}{2}$. The results are shown in Figure 8.6.

As one might expect, performance suffers for all benchmarks as the probability of random NOP insertion at each instruction boundary increases. On ARM32, we see that performance drops off consistently for the Kraken benchmark suite; and the Octane

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



(a) ARM32

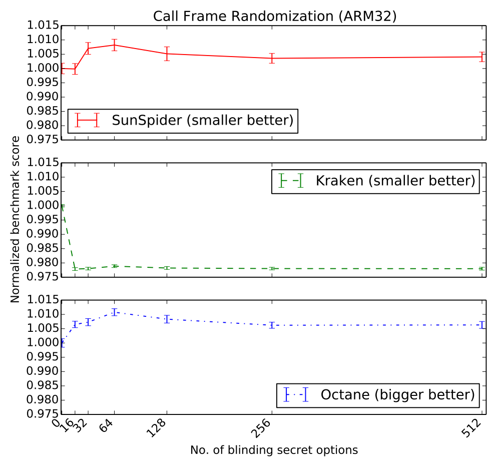


(b) x86-64

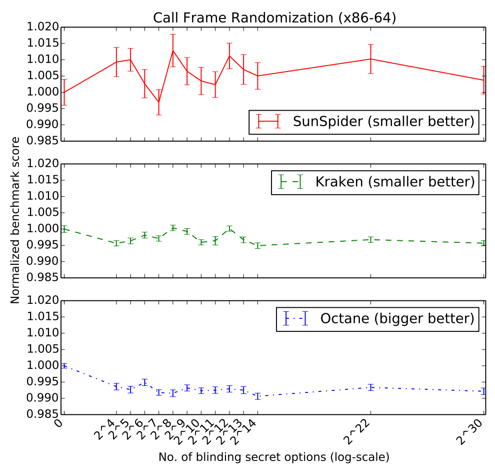
Figure 8.6. Normalized mean benchmark scores over 100 benchmark executions, across varying random NOP insertion frequencies. Error bars denote one standard error of the mean (normalized).

benchmark suite shows a large dip in performance from $\frac{1}{8}$ to $\frac{1}{4}$, making $\frac{1}{8}$ a good stopping point for ARM32. Likewise, on x86-64, Kraken and Octane overheads begin to approach 5% beyond $\frac{1}{8}$. In practice, the NOP insertion rate on ARM32 should be guided by the amount of memory overhead that the JIT developers are willing to tolerate, as we discussed previously.

Call frame randomization First, we varied the maximum magnitude of the blinding secret used to shift the call frame register when accessing incoming arguments and previous frame info in Baseline JIT call frames. One might expect increased overhead for larger blinding secrets due to varying encoding requirements for the blinding secret and blinded memory offset. The set of possible blinding secrets can be expressed as $\{0, 4\cdot 1, 4\cdot 2, 4\cdot (n-1)\}$. On ARM, we tested power-of-2 values of n from 16 up to 512. Using larger blinding secrets causes SpiderMonkey's code generation backend to generate a code sequence that loads the blinded memory offset into a scratch register, but our use of an additional scratch register to shift the call frame register exhausts the available



(a) ARM32

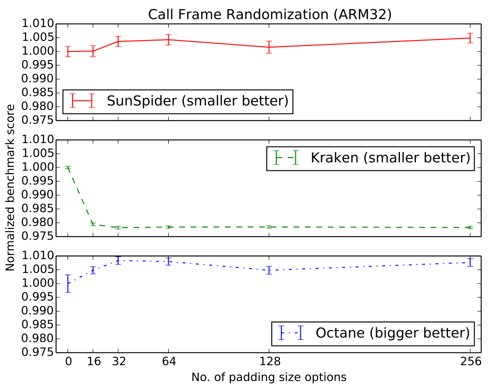


(b) x86-64

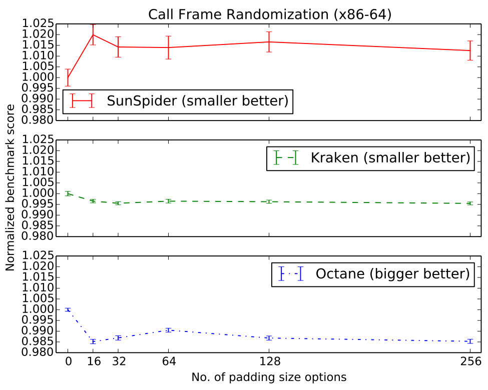
Figure 8.7. Normalized mean benchmark scores over 100 benchmark executions, across varying sizes of call frame blinding candidate pools. Error bars denote one standard error of the mean (normalized).

scratch registers. Additional modification of SpiderMonkey's ARM assembler is needed to permit larger blinding secrets. Since x86-64 supports much larger, immediates and memory displacements, we do not expect large blinding constants to affect performance. For x86-64, we varied n in such a way that $4 \cdot (n-1)$ is the largest multiple of 4 that can be encoded in 1, 2, 3, and 4 bytes ($\{2^6, 2^{14}, 2^{22}, 2^{30}\}$); we also evaluated power-of-2 values of n from 16 through 2^{13}. Figure 8.7 shows normalized ARM32 and x86-64 benchmark overheads as functions of n. Neither ARM32 nor x86-64 show a trend towards declining performance as the magnitude of blinding secrets increases.

We varied the set of possible call frame padding sizes, expressed as $\{0, k, 2k, ..., (n-1)k\}$ where k is the size of a unit of stack alignment, by varying n in power of 2 increments from 16 to 256. During these measurements, the blinding of incoming arguments and previous frame info offsets for Baseline JIT frames was enabled, but the blinding amount was fixed at 0. Figure 8.8 shows the mean benchmark scores for 32-bit ARM and x86-64, normalized.



(a) ARM32



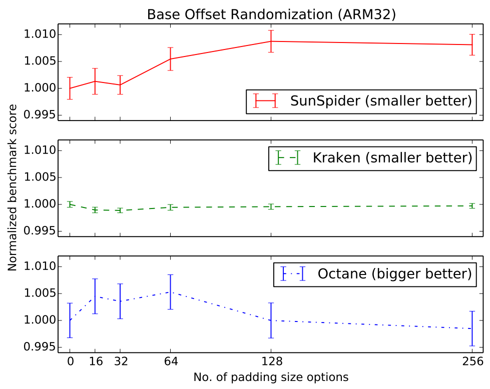
(b) x86-64

Figure 8.8. Normalized mean benchmark scores over 100 benchmark executions, across varying sizes of call frame padding candidate pools. Error bars denote one standard error of the mean (normalized).

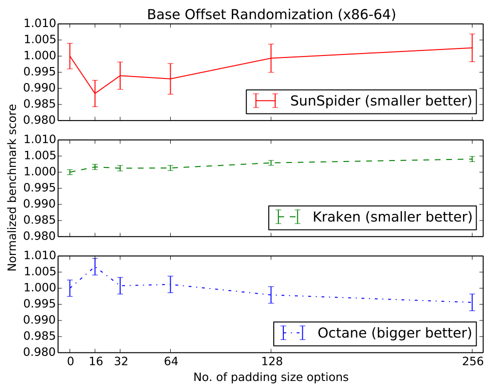
The results were surprising. On ARM32, more aggressive call frame randomization had approximately no effect on SunSpider and improved benchmark performance for Kraken and Octane. On x86-64, we see widely-varying results for SunSpider, improved performance for Kraken, and slight performance decreases for Octane as randomization aggressiveness increases. It is unclear what is causing the improved performance.

Base offset randomization We varied the set of possible base offset padding sizes, expressed as $\{0, G, 2G, ..., (n-1)G\}$ where G is the size of a unit of code alignment granularity, by varying n in power of 2 increments from 16 to 256. Figure 8.9 shows the mean benchmark scores for 32-bit ARM and x86-64, normalized.

Our measurements indicate that our implementation of base offset random could be made more aggressive by a factor of 16 without incurring more than 1% total performance overhead. More frequent exhaustion of free JIT code memory pools and the resulting memory allocation requests most likely account for the increase in memory overhead. However, the slight performance boost at 16 units on SunSpider and Octane



(a) ARM32



(b) x86-64

Figure 8.9. Normalized mean benchmark scores over 100 benchmark executions, across varying maximum code alignment units. Error bars denote one standard error of the mean (normalized).

on x86-64 is inconsistent with this hypothesis, and its cause remains unknown.

Chapter 8, in part, is a reprint of the material as it appears in *Proceedings of the 2015 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 8, in part, has been submitted for publication of the material as it may appear in appear in *Proceedings of the 2017 Network and Distributed System Security Symposium*. Wilson Lian, Hovav Shacham, and Stefan Savage, Internet Society, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 9

Conclusion

JIT compilers offer untrusted parties immense control over the contents of executable memory. The lax response to this threat has prompted a considerable amount of attention from attackers. In this dissertation, we demonstrated that JIT spraying is a general and pervasive threat by extending the attack to the ARM architectures. We analyzed vectors by which an attacker can control JIT code and encode payloads; and we used our findings to devise JIT spraying attacks against three different popular JavaScript engines.

We surveyed and analyzed the state of JIT spraying mitigations in the literature and appraised those deployed by four open source JavaScript engines. Our findings were less than encouraging. All four JavaScript engines cut corners in one way or another, and many defenses were rendered useless as a consequence.

We implemented our own versions of several diversification defenses on the SpiderMonkey JavaScript engine, with the explicit goal to avoid cutting corners. Our implementations have been made publicly available online. We conducted empirical evaluations of our defenses using industry-standard JavaScript benchmarks and found that our implementations can drastically reduce the success probabilities of known blind JIT spraying attacks with a performance penalty of $<5\%$. We urge JavaScript engine developers to consider integrating robust JIT spraying defenses to finally mitigate the threat of blind JIT spraying.

Future Directions and Final Thoughts

Once attacker-provided constants are blinded and registers are randomized, what remains to be controlled by an attacker are the choices of opcodes and implicit constants not provided directly in high level language code, but generated internally by the compiler. These parts of the instruction stream are of limited utility to a blind attacker facing base offset randomization and random NOP insertion, but they can be exploited by a more powerful attacker with a memory disclosure vulnerability.

Against an attacker with memory reading capabilities, JIT compilers can benefit from defenses built to counteract Snow et al.'s Just-In-Time code reuse attack (JIT-ROP) [51]—which, incidentally, need not involve code emitted by a Just-In-Time compiler. JIT-ROP allows an attacker to defeat fine-grained ASLR by abusing an arbitrary memory read capability to disassemble memory and discover the randomized addresses of ROP gadgets. In light of this threat, many new defenses have been proposed which prevent memory from being both disclosed and executed [60, 10, 21, 22, 29, 14].

However, a recent attack [36] has shown that reading a JIT-sprayed gadget is not necessary to confirm its address and, by proxy, its contents (when those contents are a PC-relative call to a fixed target). This attack highlights the importance of eliminating attacker influence on JIT code under stronger attack models. Beneficial future work may involve ferreting out attacker-influenced instructions and ensuring that they are properly diversified.

JIT compiler design could also be improved in a way that makes integrating security mechanisms less painful. We sang Chakra's praises during the discussion of our implementation of random NOP insertion. The design of its IR and lowering pipeline makes diversification considerably easier than what we encountered in SpiderMonkey.

New JIT compilers should take heed and improve upon, or at least follow Chakra's example.

Starting with the iPhone 5S and iOS 7, Apple began deployment 64-bit ARMv8-A processors in its smartphone line. Similarly, Android introduced 64-bit support with Android 5.0 Lollipop, and newer model Android devices are being manufactured with ARMv8-A chips. Future work should investigate the feasibility of JIT spraying on ARMv8-A. Instruction decoding ambiguity will be more challenging to exploit if the language runtime under attack is running in 64-bit mode. The 64-bit ARM instruction set (called A64) uses fixed-width, aligned 32-bit instructions; therefore same-mode instruction decoding ambiguity is not possible. Furthermore, ARMv8-A does not support interworking between 64- and 32-bit modes the same way it does ARM-Thumb interworking. Instead, it requires an change from one protection ring to another; and 32-bit mode can only be entered when changing to a less-privileged ring.

Bibliography

[1] /gs (buffer security check). http://msdn.microsoft.com/en-us/library/8dbf701c%28v=vs.80%29.aspx.

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

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

[4] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49):365, 1996.

[5] S Andersen and V Abella. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. *MSDN online library*, 2004.

[6] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 355–366, New York, NY, USA, 2011. ACM.

[7] ARM Holdings. Procedure Call Standard for the ARM Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf, November 2013.

[8] ARM Holdings. ARM Holdings plc Strategic Report 2014. http://phx.corporate-ir.net/External.File?t=1&item=UGFyZW50SUQ9NTY5OTEzfENoaWxkSUQ9MjczMTk2fFR5cGU9MQ==, 2014.

[9] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In Engin Kirda, editor, *Proceedings of NDSS 2015*. Internet Society, February 2015.

[10] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[11] Piotr Bania. JIT spraying and mitigations. *arXiv preprint arXiv:1009.1038*, 2010.

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

[13] Dion Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. Presented at BlackHat DC 2010, February 2010.

[14] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. Leakage-resilient layout randomization for mobile devices. 2016.

[15] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.

[16] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[17] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

[18] Ping Chen, Yi Fang, Bing Mao, and Li Xie. JITDefender: A defense against JIT spraying attacks. In *Future Challenges in Security and Privacy for Academia and Industry*, pages 142–153. Springer, 2011.

[19] Ping Chen, Rui Wu, and Bing Mao. Jitsafe: a framework against just-in-time spraying attacks. *IET Information Security*, 7(4):283–292, 2013.

[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pages 346–355, 1998.

[21] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical

code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.

[22] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 243–255, New York, NY, USA, 2015. ACM.

[23] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566. ACM, 2015.

[24] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. JITSec: Just-In-Time security for code injection attacks. In *Proceedings of WISSEC 2010*, pages 1–15, November 2010.

[25] Designer, Solar. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63, August 1997.

[26] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

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

[28] Hiroaki Etoh and Kunikazu Yoda. ProPolice: Improved stacksmashing attack detection. *IPSJ SIG Notes*, 75:181–188, 2001.

[29] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336. ACM, 2015.

[30] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[31] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 993–1004. ACM, 2013.

[32] Abhinav Jangda, Mohit Mishra, and Bjorn De Sutter. Adaptive just-in-time code diversification. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 49–53. ACM, 2015.

[33] Martin Jauernig, Matthias Neugschwandtner, Christian Platzer, and Paolo Milani Comparetti. Lobotomy: An architecture for jit spraying mitigation. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 50–58. IEEE, 2014.

[34] Tim Kornau. Return oriented programming for the ARM architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[35] Wilson Lian, Hovav Shacham, and Stefan Savage. Too LeJIT to Quit: Extending JIT Spraying to ARM. In Engin Kirda, editor, *Proceedings of NDSS 2015*. Internet Society, February 2015.

[36] Giorgi Maisuradze, Michael Backes, and Christian Rossow. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 139–156, Austin, TX, August 2016. USENIX Association.

[37] Farhad Mavaddat and Behrooz Parhami. Urisc: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education*, 1988.

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

[39] Jan de Mooij. W^X JIT-code enabled in Firefox. http://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox/, December 2015.

[40] 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].

[41] Mozilla. Bug 677272 - JIT hardening. https://bugzilla.mozilla.org/show_bug.cgi?id=677272, 2011. [Online; accessed 3 November 2015].

[42] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328. ACM, 2014.

[43] Filip Pizlo. Introducing the WebKit FTL JIT. https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/, May 2014.

[44] Chris Rohlf and Yan Ivnitskiy. Attacking Clientside JIT Compilers. http://www.matasano.com/research/Attacking_Clientside_JIT_Compilers_Paper.pdf, 2011.

[45] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 4, pages 159–169, 2004.

[46] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[47] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of CCS 2004*, pages 298–307. ACM, 2004.

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

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

[50] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php, 2004.

[51] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of IEEE Security and Privacy ("Oakland") 2013*, pages 574–88. IEEE Computer Society, 2013.

[52] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. In *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*, 2015.

[53] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

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

[55] László Szekeres, Mathias Payer, Tao Wei, and Dong Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

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

[57] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[58] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. Secure dynamic code generation against spraying. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 738–740. ACM, 2010.

[59] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. Insert: Protect dynamic code generation against spraying. In *Information Science and Technology (ICIST), 2011 International Conference on*, pages 323–328. IEEE, 2011.

[60] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 35–46. ACM, 2016.

[61] Rui Wu, Ping Chen, Bing Mao, and Li Xie. Rim: A method to defend from jit spraying attack. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 143–148. IEEE, 2012.

[62] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[63] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. JITScope: Protecting web users from control-flow hijacking attacks. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 567–575. IEEE, 2015.