Too LeJIT to Quit: Extending JIT Spraying to ARM

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

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

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

***Abstract*—In the face of widespread DEP and ASLR deployment, JIT spraying brings together the best of code injection and code reuse attacks to defeat both defenses. However, to date, JIT spraying has been an x86-only attack thanks to its reliance on variable-length, unaligned instructions. In this paper, we finally extend JIT spraying to a RISC architecture by introducing a novel technique called *gadget chaining*, whereby high level code invokes short sequences of unintended and intended instructions called gadgets just like a function call. We demonstrate gadget chaining in an end-to-end JIT spraying attack against WebKit's JavaScriptCore JS engine on ARM and found that existing JIT spray mitigations that were sufficient against the x86 version of the JIT spraying attack fall short in the face of gadget chaining.**

I. Introduction

It is no secret that programs are replete with bugs. Some of these bugs allow an attacker to subvert control of the program counter and divert execution away from its intended path; these are called control flow vulnerabilities. Unfortunately for a would-be attacker, a control flow vulnerability is not enough to execute arbitrary code on a remote machine. Defense mechanisms such as DEP and ASLR prevent attackers from writing code into a process's address space and decrease the likelihood that triggering a control flow vulnerability will cause an attacker's target code to execute.

JIT spraying is an attack which defeats both DEP and ASLR by enabling an attacker to predictably influence large swaths of the victim process's executable memory. The attack exploits Just-in-Time compilers built into many recent language runtimes for the purpose of speeding up the performance of frequently-executed code, but it has only been demonstrated for the x86 architecture. More and more handheld devices, which are predominantly powered by ARM processors, are connecting to the Internet and running web browsers, making themselves candidates for remote exploitation. Since most modern web browsers implement a JavaScript runtime environment with a fully-functioning JIT compiler, JIT spraying is a fantastic vector for attacking a browser. However, JIT spraying has historically been limited to the x86 architecture. In this paper, we challenge this trend and show that JIT spraying is indeed a viable attack against ARM.

NDSS '15, 8-11 February 2015, San Diego, CA, USA
 ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23288

A. Related Work

History holds no shortage of techniques for remotely executing arbitrary code. They can be broken down into roughly two categories, code injection attacks and code reuse attacks. Code injection attacks such as Aleph One's famous stack smashing attack [1] and SkyLined's heap spraying attack [19] introduce new executable code into a vulnerable process's address space and exploit a control flow vulnerability to divert execution to it. Data Execution Prevention (a.k.a. DEP, W $\oplus$ X, etc.) [2] has become the standard defense against code injection. DEP is a defense mechanism that allows processes to mark certain pages of memory as non-executable to prevent attackers from writing their own executable code. Usual candidates for DEP are the process's stack and heap, where an attacker can most easily inject bytes of her choosing.

Enter code reuse attacks, which circumvent DEP 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 [21] and return-oriented programming (ROP) [16]. The most widely deployed defense against code reuse attacks is known as Address Space Layout Randomization (ASLR). Code reuse attacks require the attacker to pinpoint the addresses of the instructions they intend to repurpose for their malicious computation. 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 memory image.

JIT spraying brings together code injection and code reuse in a hybrid that defeats both DEP and ASLR. In [7], Blazakis demonstrated JIT spraying by exploiting the insight that JIT compilers give attackers a tremendous amount of control over the contents of executable memory due to the predictable way immediate operands are handled. For example, consider the following ActionScript statement:

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

When it is compiled by the ActionScript JIT compiler, the following instructions are produced:

```
b89090903c    mov eax, 3c909090h
359090903c    xor eax, 3c909090h
359090903c    xor eax, 3c909090h
```

Observe that the immediate values in the ActionScript source code appear directly in the code in little endian byte order. Remember that since they were produced by a JIT compiler, they are meant to be executed and therefore reside in memory marked executable. When executed from the second byte, the instruction stream instead becomes the following:

```
90          nop
90          nop
90          nop
3c35        cmp al, 35h
90          nop
90          nop
90          nop
3c35        cmp al, 35h
...
```

This "hidden" unintended instruction stream is a NOP sled. If execution begins at an any of the unintended instruction boundaries (which occur at 4 out of 5 addresses), the NOP sled will execute. Notice that the 0x3c bytes, which were once interpreted as part of the immediate operands, are now acting as the opcodes for `cmp al` instructions, which are semantic NOPs that consume the original `xor eax` opcodes—the 0x35 bytes—as immediate operands. The chain of XORed immediates in the original source code can be extended, and as long as the most significant byte of each immediate is 0x3c, the `xor eax` opcode will continue to be masked, preventing resynchronization to the intended instruction stream. Later down the chain, the 0x90 bytes can be replaced with the encodings for shellcode instructions up to 3 bytes in length.

The fact that JIT spraying allows the attacker to control 4 out of every 5 bytes in executable memory makes it a code injection attack that defeats DEP. Since the attacker can spray numerous copies of each XOR chain, each with its own long NOP sled and shellcode payload, JIT spraying also gives an attacker a high probability of defeating ASLR through random address guessing. Once several hundreds of megabytes of memory are filled with JIT spray payloads, a random jump has a non-negligible chance of landing in a NOP sled at the correct offset to execute the unintended instruction stream.

Since [7], Sintsov explored the art of writing ActionScript JIT spray payloads for the x86 in greater depth, demonstrating the construction of a stage-0 JIT spray shellcode [18]. JIT spraying has been extended beyond the ActionScript JIT to other x86 JITs. In [17], Sintsov demonstrated the construction of an x86 JIT spraying payload with the JavaScriptCore Baseline JIT; and Rohlf and Ivnitskiy demonstrated JIT spraying on x86 against Mozilla's JaegerMonkey and TraceMonkey JavaScript engines as well as unlocking 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 [15].

In 2011, Pete Beck [6] proposed an idea for JIT spraying on the ARM platform using constant pools to encode malicious instructions. We do not employ Beck's idea, but in this paper, we exend JIT spraying to a JavaScript JIT running on the ARM architecture. In doing so, we make use of a concept similar to Rohlf and Ivnitskiy's ROP gaJITs, but our attack is ultimately something quite different from ROP.

B. Assumptions and Adversary Model

The goal of this paper is not to highlight a particular bug in a piece of software, but rather to demonstrate techniques that may be used to construct a working end-to-end attack once such a vulnerability is identified. Therefore, one of the fundamental assumptions in this paper is that an attacker is able to trigger a bug in a program that causes control flow to branch to an arbitrary address of her choosing.

This control flow vulnerability alone may not necessarily be sufficient for an attacker to induce the vulnerable process to execute arbitrary code. We assume that the vulnerable process may be protected by security mechanisms such as DEP and ASLR, which complicate the task of launching traditional code injection and code reuse attacks.

C. Our Contributions

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 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 [8], ARM [13], and the Zilog Z80 [9]. In this paper, we continue the tradition of extending attacks from the x86 to new architectures in the following ways:

1) We show for the first time that RISC architectures are not immune to JIT spraying attacks and highlight features in the ARM architecture that can enable an attacker to use a JIT compiler to inject unintended instructions into executable memory.
2) We present a new model for JIT spraying in which unintended instruction execution is interweaved with execution of a high level language and describe a proof of concept attack following this model.
3) We show that the state of JIT spraying defense deployment on ARM is insufficient and highlight its particular weaknesses and areas for improvement.

JIT spraying does not constitute a vulnerability in the absence of some way to redirect control flow. Nevertheless, we have shared our findings with WebKit's security team.

II. The 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 estimates that in 2010, ARM-based processors had seized 90% of the market share in both smart phones and feature phones as well as 70% of the portable media player market share [3].

Most modern ARM processors support a 32-bit address space with 32-bit arithmetic. Chips implementing the

newer 64-bit ARMv8-A architecture, introduced in 2011, are still rare, even in new devices; accordingly, we focus on ARMv8-A's predecessor, ARMv7-A, in this paper.

A. Instruction sets

Prior to ARMv4T, the ARM architecture supported a single instruction set known simply as "ARM." ARM instructions are stored as fixed-width 32-bit words aligned to 4-byte boundaries. Whereas the x86 instruction set supports conditional execution only of branch instructions, most ARM instructions can be predicated through a 4-bit condition code.

In 1994, ARM Holdings released the ARM7TDMI core implementing the ARMv4T architecture, which introduced the "Thumb" instruction set, composed of 16-bit 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. The Thumb instruction provides the advantage of improved code density over ARM, in part due to the removal of condition codes from nearly all instructions.

In 2003, the Thumb instruction set was enhanced with Thumb-2 technology, which added 32-bit instructions (separate from those found in the ARM instruction set) that can be intermixed with 16-bit Thumb instructions. Unlike ARM instructions, which are encoded as 32-bit words, 32-bit Thumb-2 instructions are encoded as two consecutive 16-bit halfwords. Thumb-2 support was introduced in the ARMv6T2 architecture and is mandatory in all cores implementing ARMv7 and above. For the remainder of this paper, the names *Thumb* and *Thumb-2* are used interchangeably to refer to the Thumb-2 instruction set containing mixed 16- and 32-bit instructions.

The ARM architecture includes support for two other instruction set modes, 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 paper.

Whether an instruction stream is interpreted as ARM, Thumb, ThumbEE, or JVM bytecode is determined by the instruction set state register (`ISETSTATE`), which 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. Interworking is implemented through a handful of instructions that always change the instruction set as well as the least significant bit of branch target addresses. When branching to an address using an instruction allowing for instruction set interchange (i.e., `bx` and `blx`), 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.

TABLE I. 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`			✓		

B. 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) [4] and is summarized in Table I.

The special-purpose registers have roles defined by the instruction set and implemented in hardware. The stack pointer register (`SP`/`R13`) is used to hold a pointer to the top of the current execution stack. Special variants of the `add` and `sub` instructions are hardwired to use `SP` as an operand in order to accelerate stack operations.

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 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 currently-executing instruction plus 8 while in ARM mode or the address of the currently-executing instruction plus 4 while in Thumb mode. Certain 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.

C. Endianness

ARM is a bi-endian architecture, meaning that it can interpret words and halfwords 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.

D. Security considerations

Several features of the ARM architecture could result in an ARM processor's interpreting instruction bytes differently than intended by the assembler that produced them, something we call "instruction confusion." First, the ARM-Thumb interworking feature can lead to instruction confusion because execution beginning at a particular address could result in multiple different instruction streams, depending on the contents of the ISETSTATE register. Second, the ability to change the endianness of instruction memory can affect the instruction decoding. Third, the addition of 32-bit Thumb-2 instructions to the Thumb instruction set effectively gives Thumb-2 variable-length instruction encodings since 16- and 32-bit Thumb instructions may be intermixed. Variable-length instructions were the cornerstone of Blazakis' original JIT spraying attack.

We analyze ARM-Thumb interworking and variable-length instructions as possible vectors for creating instruction confusion in § IV. We exclude instruction endianness switching from our analysis because it is not present in recent processors.

III. JavaScriptCore: WebKit's JavaScript Engine

We used a recent version of WebKit's JavaScript engine JavaScriptCore (JSC)[1] compiled for ARMv7-A as a test platform for generating JITed payloads. We chose WebKit for a number of reasons. First, JSC is open source, reducing the reverse engineering burden required to understand how its JIT compiler works. Second, WebKit and JSC are used by Apple for the Safari browser built-in to iOS. The iOS platform is popular (iOS devices made up 32.6% of the smartphone market in the US, according to Kantar Worldpanel[2]) and strictly locked down. All executable code pages on iOS are cryptographically signed, with one exception: code emitted by the Safari JIT compiler. This policy complicates traditional code injection attacks, making a JSC JIT exploit an attractive attack path.

JSC is a multi-tier JavaScript engine, meaning it applies increasing levels of optimization to code the more times it is executed. 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[3] by the Baseline JIT. Once the Baseline JIT's code has executed many times (60 times for functions or 1000 times for loops), the 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 simply varying the number of times the script is invoked.

Code memory pages emitted by both the Baseline JIT and the DFG JIT 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.

A. 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.

B. 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 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.

C. 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.

D. 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 [14]. As the FTL JIT has only recently been taken out of the experimental development phase and was not enabled in the version of WebKit that we studied, its functionality is out of the scope of this paper.

[1] We used WebKitGTK version 2.2.2-1 for Debian.

[2] Online: http://www.kantarworldpanel.com/smartphone-os-market-share/. Accessed 18 November 2014.

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

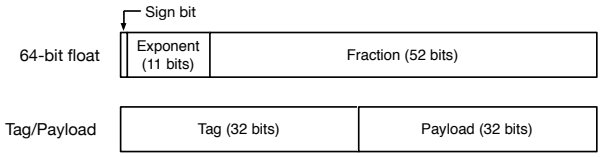


Fig. 1. Illustration mapping the bits of an IEEE 754 double precision floating-point number to the tag and payload portions of a 32-bit JSC JS value.

E. 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 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 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.

F. 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 § III-E) 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.

G. Controlling JSC's JIT output

One of the primary goals when devising a JIT spraying payload is maximizing the percentage of attacker-controlled bits. We define an attacker-controlled bit as a bit in an instruction's immediate or register field whose value an attacker can manipulate by varying her input to 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 instructions bytes that do not contain attacker-controlled bits. Blazakis' original JIT spray payload against x86 is a perfect example. It contained numerous back-to-back sequences of one non-attacker controlled byte (the instruction opcode) followed by four bytes of attacker controlled data (an immediate operand).

TABLE II. ARM AND THUMB IMMEDIATE ENCODING LIMITS

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

Unfortunately, ARM and Thumb instruction encodings cannot contain as many immediate bits as x86 instructions. On ARM, a 32-bit immediate constant must be split into halfwords and loaded into a register one halfword at a time. Contrast this with x86, which can load an entire 32-bit immediate in a single instruction. Table II lists the maximum number of encodable immediate bits in any ARM or Thumb instruction as well as in arithmetic instructions, whose immediate bits are more readily and precisely influenced by the JIT's input than other instructions such as load/store or branch instructions. Note how few consecutive immediate bits are available to an attacker when the JIT emits Thumb instructions.

Aside from the immediates in arithmetic instructions, it is plausible that an attacker could control the immediate bytes in a PC-relative branch by compiling an `if`-statement and bloating the size of the conditionally-executed block. Thumb's PC relative branches contain up to 24 bits of immediate offset interpreted as a signed integer and multiplied by 2. This translate to a branch distance of up to 16MB in order to control the most significant bits in the immediate. Chaining together multiple payloads of such a size in order to construct a full shellcode would consume exorbitant amounts of memory.

Aside from immediate bits, an attacker can control the choice of operand and destination registers used in instructions by carefully crafting her input to the JIT compiler. JSC's Baseline JIT uses "canned" JIT output whose instructions predictably operate on certain registers; it is therefore immune to this form of manipulation. The DFG JIT on the other hand does perform scratch register allocation, enabling an attacker to craft a JavaScript paylod that causes arbitrary scratch registers to be used together with one another. Furthermore, an attacker can actually control which values the JIT places in which scratch registers. Listing 1 shows a function in which an attacker is able to place certain variables in predetermined registers and afterwards force the JIT to use certain registers together as the operands to bitwise operations.

The function'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 another register, again in predictable order. Since the attacker can control both the values of the arguments passed into this function at runtime and the immediate constants XORed against the arguments, she can control the value of all 8 registers identified in the function at the point in execution just after the definition of the variable `R11`. 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 2 shows the raw bytes and disassembly of the return value computation. 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

```
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 1. 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 2. Raw bytes and disassembly of the computation of the return value in Listing 1.

on the order of the operands. The result of (`R4 ^ R8`) is stored into `R4` because `R4` was written to the left of the XOR operator. Likewise, the result of (`R10 ^ R9`) is saved into `R10`, which was written to the left. This is in contrast with a JIT that might choose the accumulator register based on some other criteria such as whichever has a lower number.

Influence over what registers are used as operands translates to influence over the length of instruction encodings emitted by the DFG JIT. Three-bit register fields limit many 16-bit Thumb instructions to operating on 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 2 which shows that the DFG JIT chooses to use the 16-bit XOR encoding to XOR `R0` with `R1`, 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`.

Instead of chaining bitwise operations to induce the JIT to emit densely-packed attacker-controlled instructions, one might consider other types of arithmetic operations such as addition, subtraction, multiplication, or division. However, JSC injects runtime checks with these operations to test for overflows and underflows. These checks introduce several consecutive instructions that do not contain attacker controlled bytes and should therefore be avoided. For this reason, we consider chained bitwise operations to be the best method for generating long runs of tightly-packed instructions with a relatively high concentration of attacker-controlled bits.

In the remainder of this paper, we evaluate various techniques for JIT spraying against ARM with respect to the behavior of a particular version of JSC. However, we believe that the techniques presented are applicable in general against ARM, and we describe both techniques whose constraints are met by JSC and those that do not work against our version of JSC but may work against other versions or even other JITs.

IV. ARM JIT Spray Payloads for JSC

In JIT spraying, as in return-to-libc or ROP, the attacker repurposes existing benign code for malicious purposes. What distinguishes JIT spraying from other code reuse attacks is that the reused instructions in a JIT spraying attack are the result of compiling attacker-provided code. The instructions executed during the attack may be either the instructions intended by the compiler that produced them or an entirely different sequence of unintended instructions "hidden" among the intended instructions' encodings.

In this section, we describe three types of JIT spraying payloads making use of unintended instructions that are made possible by features in the ARM architecture. First, we discuss the original, self-sustaining JIT spray payload which strings together long sequences of unintended instructions that do not resynchronize to the intended instruction stream due to architectural support for variable-length, partially-aligned instruction encodings (§ IV-A). We were unable to find a method that makes it feasible to generate useful payloads of this type with either JSC's Baseline or DFG JITs. The second type of payload takes advantage of recent ARM chips' ability to interpret a region of memory as either ARM or Thumb instructions, depending on the state of the `ISETSTATE` register (§ IV-B). The basic idea is to generate JIT code intended to be executed in Thumb mode that encodes a malicious unintended instruction stream when executed in ARM mode. We found that certain incompatibilities between the Thumb and ARM instruction set encodings make it infeasible to create a useful Thumb-ARM reinterpretation payload. Although we were unable to generate self-sustaining and Thumb-ARM reinterpretation JIT spray payloads, we include our findings on them to serve as a reference for future work, should certain conditions change.

Finally, we describe a third payload style which, unlike self-sustaining JIT spray and Thumb-ARM reinterpretation payloads, does not attempt to construct a Turing-complete shellcode from long sequences of unintended instructions. Instead, the payloads consist of unintended instructions that are allowed to resynchronize to the intended instructions that follow them. Rather than expressing Turing-complete computation in the JIT sprayed payloads (which we refer to as "gadgets"), the JavaScript that triggers the control flow vulnerability provides Turing-complete computation and chains together calls to the gadgets. We call this technique *gadget chaining* and describe it in greater depth in § IV-C.

A. Self-sustaining JIT spraying payloads

Blazakis' original JIT spraying attack on x86 abused the architecture's support for variable-width instruction encodings, which allows any byte to be decoded as the first byte of an instruction. Consequently, once control flow is diverted into the middle of an instruction in a cleverly-designed JIT spray payload, it is possible to prevent execution from ever resynchronizing to the intended instruction stream. The encoding of an unintended instruction stream in an intended instruction stream is also possible on ARM, with some caveats.

The execution of unintended instructions brought about by branching into the middle of an intended instruction is limited to Thumb mode execution. Specifically, the CPU under attack must support the extended 32-bit 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 instruction. Likewise, in 16-bit-only Thumb mode, all 2-byte aligned branch targets are intended instructions.

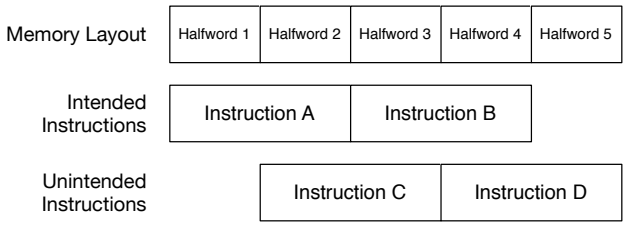


Fig. 2. Example of two possible Thumb-mode decodings of a sequence of halfwords.

However, with the introduction of Thumb-2, the ARM designers allowed 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 instructions, and the intended instructions encoding them must also be 32-bit Thumb-2 instructions. This is essentially a chain of intended 32-bit Thumb-2 instructions executed one halfword out of phase, as illustrated in Figure 2. 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 JSC's JITs to generate a chain of 32-bit Thumb instructions that encode an 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. This means it must begin with either the 1111_2 or 11101_2 bit pattern. The method of chaining bitwise operations does not satisfy this constraint because the second halfword of all 32-bit Thumb encodings of the AND, OR, and XOR operations begin with a 0-bit, which is a 16-bit Thumb instruction prefix. Thus, we can only execute a single unintended 16-bit Thumb instruction before execution resynchronizes. We do not believe it is possible to induce JSC's JITs to emit code that will encode a long, non-resynchronizing unintended Thumb instruction stream, because those 32-bit Thumb instructions whose second halfwords do satisfy the prefix constraint are not instructions that we can readily control and induce the JIT to produce back-to-back many times in a row.

In the next section, we describe a payload type that does not suffer from resynchronization to the intended instruction stream because it changes the instruction set execution mode of the processor.

B. Thumb-ARM reinterpretation

As we noted in § II, recent ARM processors support both the ARM and Thumb instruction sets and allow interworking between the two by setting or clearing the least significant bit of a branch target address. An obvious consequence of this 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. If this happens the processor is executing unintended instructions. An attacker could exploit such a vulnerability by crafting an input to a JIT compiler that leads to the emission of code that, when executed as an instruction set not intended by the JIT compiler, performs malicious computation.

On ARMv7-A, JSC's JIT compilers produce Thumb-2 code. To exploit ARM-Thumb interworking against JSC, an attacker must induce the JIT to produce Thumb-2 instructions whose bytes in instruction memory decode to a useful stream of ARM instructions and execute it in ARM mode via an interworking branch instruction. The attacker's ability to generate such a malicious JIT payload is impeded by the layout of certain instruction fields that overlap when Thumb instruction bytes are interpreted as ARM instructions.

The two ARM instruction fields that make it difficult to encode ARM instructions in a Thumb instruction stream are the condition code and ALU result destination register (denoted `Rd`). Recall from § II-A that most ARM instructions contain a condition code in their first four bits. The condition code predicates the runtime execution of the instruction on the status of condition flags (e.g., negative, zero, carry, overflow, etc.) set by previously-executed instructions. Although the 4-bit condition flag field can theoretically assume 16 possible values, only 15 of them are valid condition codes; 1111_2 is an illegal condition code that will cause a processor exception if encountered. Some ARM instructions do not treat the first four bits as a condition code. These instructions may only be executed unconditionally, and the first four bits 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 change to Thumb mode. These types of instructions are unlikely to be useful to an attacker's shellcode. Therefore inducing the JIT to emit an instruction stream with the 1111_2 bit pattern aligned to the beginning of ARM instruction boundaries should be avoided. Our go-to instructions for generating long sequences of back-to-back instructions with many attacker-controlled bits—bitwise operations with an immediate-operand—all begin with the 1111_2 bit pattern. Therefore, we must not allow the first halfword of these instructions to align with the high order two bytes of the unintended ARM instructions. This can be easily accomplished by padding the Thumb instruction stream in such a way that the first halfwords of the Thumb instructions are "out-of-phase" with the ARM condition codes.

However, even sidestepping the condition code is insufficient to avoid encoding undesirable ARM instructions in the Thumb instruction stream. Nearly all ARM instructions that produce an ALU result (e.g., `MOV`, `ADD`, `AND`, etc.) have their destination register (`Rd`) field located in the high-order nibble of the 3^{rd} most significant byte of the instruction, as depicted in Figure 3. Also shown are mappings of four consecutive instruction bytes into ARM and Thumb instructions. Note that the condition code and `Rd` field in unintended ARM instruction correspond to the most-significant nibbles of both halfwords in the Thumb-2 instruction stream. This means that even if

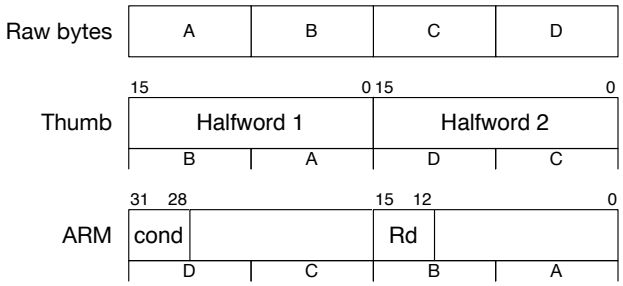


Fig. 3. Example decoding of four consecutive bytes of little endian instruction memory into two 16-bit Thumb halfwords and an ARM instruction with both a condition flag and an ALU destionation register. Bytes are denoted A-D. Note the correspondence between the first 4 bits of each Thumb halfword and the condition code and `Rd`.

the Thumb instruction chain is shifted out of phase by 2 bytes to avoid interpreting the Thumb instructions' 1111_2 bits as condition codes, they will still align with the `Rd` field. While this does give an attacker an easy way to encode branches (since writing to the `PC` results in a branch to the address written) it is difficult to perform useful computation without being able to write to other registers. 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`.

As we saw in Listing 2, it is possible to chain bitwise operations with register operands as well. Register-operand bitwise operations have both 16- and 32-bit encodings. The 32-bit encodings begin with 11101010_2, which means their first halfword can be aligned with the ARM instructions' condition codes. The 1110_2 condition code executes the instruction unconditionally, which saves the attacker having to predict the state of the condition flags at the time control flow is diverted to the shellcode. Unfortunately, the only ARM instruction that can begin with 11101010_2 is the `PC`-relative branch. If the Thumb instruction stream is shifted by 2 bytes so that the 11101010_2 prefix does not align with an ARM instruction's condition code, it restricts the ARM instruction's `Rd` field to `R14`/`LR`. If 16-bit encodings are used instead of their 32-bit counterparts, it is possible to unlock a broader array of unintended ARM instructions, but they are limited to subtraction, bitwise AND, bitwise XOR, and moving an immediate. In all cases, the unintended instruction's `Rd` must be `R4`. These restrictions make it extremely challenging to construct a Turing-complete ARM shellcode. Even interleaving 16-bit register-operand bitwise operations with 32-bit bitwise instructions with either register or immediate operands does not enable the attacker to construct a useful Turing-complete ARM shellcode or stack pivot.

The above observations suggest that chained bitwise operations are not a viable method for generating a JIT spray payload exploiting Thumb-ARM reinterpretation. Since we are unaware of a better method for inducing JSC's JITs to tightly pack instructions containing a high concentration of attacker-controlled bits, we conclude that with our current knowledge, Thumb-ARM reinterpretation is a nonviable JIT spraying payload technique for the version of WebKit under study.

C. Gadget chaining

In this subsection, we introduce *gadget chaining*, a novel technique for utilizing JIT sprayed payloads. Unlike self-sustaining and Thumb-ARM reinterpretation payloads, which operate by branching to a long sequence of unintended instructions comprising the entirety of the attacker's malicious computation, gadget chaining uses the high level language already available to the attacker to perform Turing-complete computation and augments it with short sequences of unintended computation generated by the JIT compiler.

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

Listing 3. 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. Selected instructions from the DFG JIT compilation of the `readGadget` function given in Listing 3.

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.

As we observed in § IV-A, it is challenging on ARM to encode long sequences of unintended instructions that can execute one after another. For this reason, most of the instructions executed in each gadget are actually intended instructions. The only unintended instructions in a gadget are at the beginning, after which execution resynchronizes to the intended instruction stream. Eventually execution reaches the end of the gadget, where a function return sequence is 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 the DFG JIT, the JavaScript function `readGadget` shown in Listing 3 will contain a gadget that will enable an attacker to read bytes from a specific address in memory into a JavaScript value. Listing 4 shows an excerpt from `readGadget`'s DFG JIT's compilation, 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 the tag field of the return value to indicate that the return value is a 32-bit integer.

Listing 5 shows the instruction stream if execution begins at the second halfword of the 32-bit instruction at offset 0x36. The halfword at offset 0x38 is a 16-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

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

resynchronizes to the intended instruction stream. The upper halfword of `R12` is loaded with the value `0x1111`,[4] and the value that was read from memory is XORed against the entire contents of `R12`. It is necessary to write `0x1111` into the upper halfword of `R12` because the attacker may not be able to predict `R12`'s value at the time the gadget is invoked. 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 depicts the overall process of calling and returning from the read gadget. A JavaScript control program calls a wrapper function, which places the 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. 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) in order for the gadget's return sequence to work properly and for subsequent execution to continue without 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. In § V-A, we provide a detailed explanation of methods we developed to successfully call into and return from gadgets in JSC running on ARMv7-A.

We have so far shown only one example gadget, the read gadget, but there are many other unintended instructions that can be encoded in gadgets. In the rest of the paper, we will touch on two other examples, the memory store gadget and the register-move (a.k.a. register-disclosure) gadget.

V. Applications of Gadget Chaining

In this section we demonstrate the use of gadget chaining in the construction of an end-to-end proof of concept attack against WebKit on ARMv7-A. We also describe other uses for gadget chaining that could be useful in the context of other control flow vulnerabilities or JavaScript engines.

[4]This pollution of `R12` is not harmful because JSC uses it as a short-lived temporary value.

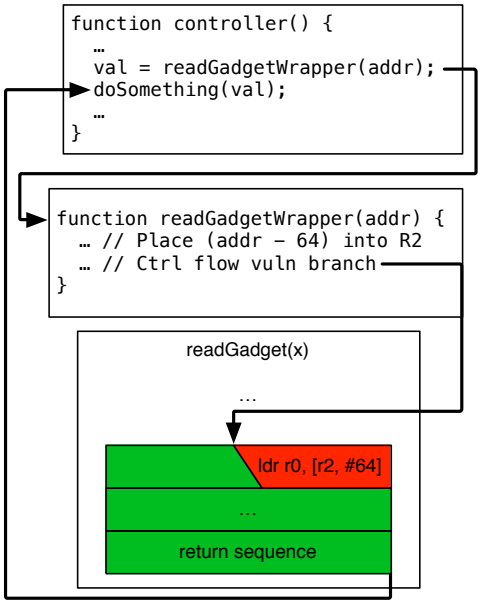


Fig. 4. Diagram of the invocation of the read gadget with arrows showing control flow.

A. Proof of concept attack

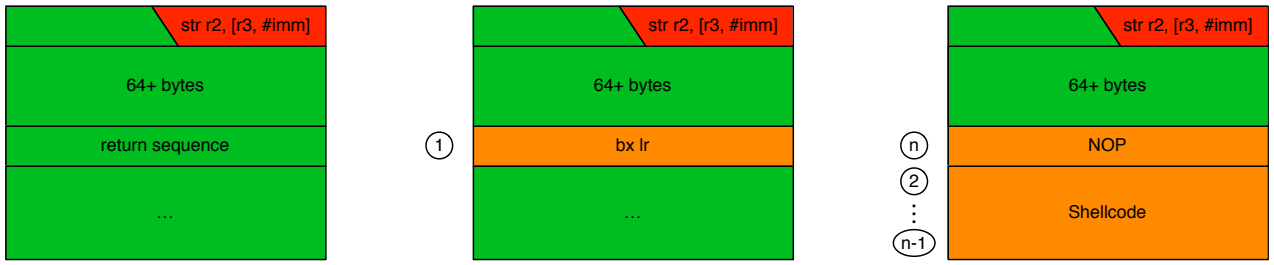
We devised a proof of concept attack which uses the gadget chaining technique to execute arbitrary code. The high level overview of the attack is as follows: The attacker lures the victim's WebKit browser into loading and executing attacker-provided JavaScript. For example, the attacker could purchase 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 paper (e.g., via a use-after-free bug).[5] The function whose pointer was corrupted should have the the following two properties:

1) Execution can be induced on-demand by the attacker
2) Accepts two attacker-controlled 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.

[5] We simulate a use-after-free vulnerability by adding a virtual function, `hijackVFT()`, 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.

Fig. 5. 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.

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 5a. 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 5b). 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 5c).

In the remainder of this subsection, we explain the details of our proof of concept attack, most notably those that pertain to the following three major components of deploying an attack using gadget chaining: (1) pinpointing gadgets in memory; (2) preparing registers and branching to gadgets from JavaScript; and (3) returning from gadgets without crashing.

1) 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. Contrast this with a spraying attack in which the attacker only needs to cause control flow to branch to somewhere in a region of memory where a NOP sled has been sprayed.

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 essential that the gadget be produced by the DFG JIT rather than the Baseline JIT. This is because JSC's Baseline JIT performs 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.
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.
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 $|Baseline_{ps}|$, $|Baseline_{s}|$, $|DFG_{ps}|$, and $|DFG_{s}|$ denote the size in bytes of the store gadget function's pre-shrinking

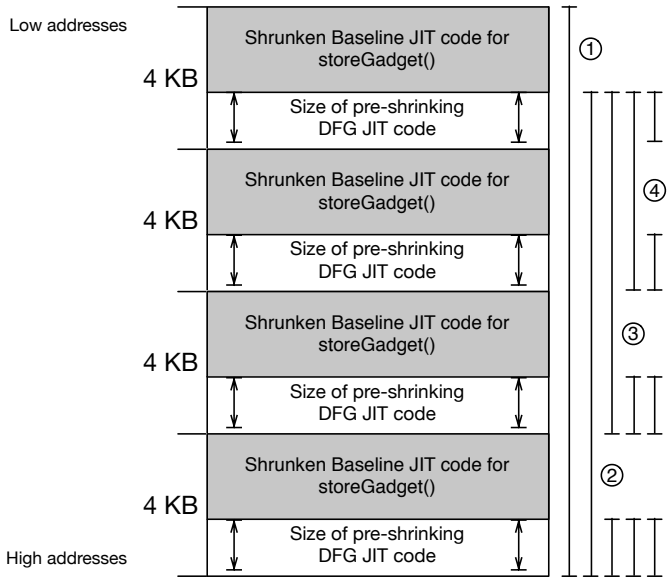


Fig. 6. Layout of executable memory showing holes created between instances of the store gadget function's Baseline JIT code. Bars on the right show the progression of the available space as instances of the Baseline JIT code are allocated. The encircled numbers indicate the order of Baseline JIT code allocation.

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:

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

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 6 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 approximately 4076 bytes ensures that despite small shrinkages due to uncontrollable optimization and small expansions due to constant blinding (§ VI-A2), 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.

2) 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 displacement will be added to form the store 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-controlled 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 displacement value).

3) 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 calling JavaScript code. The prototype function acts as the wrapper function shown in Figure 4.

However, we cannot return directly from the gadget to the JavaScript code that called the prototype function. 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 5b), 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 many recent ARMv7-A implementations) of instructions between the unintended store and the return sequence.[6] 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.

B. 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 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 machine with 4 GB of memory before the browser process 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.

C. Other gadget chaining applications

The details of our proof of concept attack are tailored to the version of WebKit under study. For example, our

[6] 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 integrity of the call frame pointer register, DFG JIT code is ideal.

decision to use a hijacked host function pointer influenced our attack, as we were forced to use self-modifying code to deal with constraints imposed by WebKit's host function calling convention. Nevertheless, our core new technique, gadget chaining, is robust against JIT compiler implementation choices and applicable in scenarios other than Turing-complete JIT spraying, as we explain below.

1) Just-In-Time code reuse: Snow et al. demonstrated an attack dubbed Just-In-Time code reuse [20], which 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 Snow et al.'s Just-In-Time code reuse attack is an existing memory disclosure vulnerability: a `ReadByte(address)` function. The read gadget example we gave in § IV-C provides exactly this functionality.

Let us assume that it is not possible to create a self-modifying gadget as we did in the proof of concept attack, motivating us to pursue the Just-In-Time code reuse attack as a means for arbitrary code execution. In order for the read gadget to return without crashing, it needs to be called from JavaScript code rather than a host function's prototype function. Fortunately, 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 JSC 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 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 held in an object which could potentially be corrupted by the attacker.

The task of loading a register with the memory address from which to load can be handled by making the wrapper function a DFG-compiled function, which we showed in § III-G enables an attacker to control the contents of several registers at certain points in the execution of the code. The version of WebKit we studied preserves register `R2` in the code used to prepare for a function call. Thus, the attacker has everything she needs to implement the `ReadByte(address)` needed by the Just-In-Time code reuse attack.

2) Address/Register disclosure: JavaScriptCore's tag-payload value structure makes it possible for a register-move gadget to disclose the value of any register (even the stack pointer or link register) into a JavaScript integer 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 32-bit integer.

The JavaScript statement `return arg0 ^ 0x10;` produces an `R2`-disclosure gadget that can be used to cast a JavaScript object to an integer. The gadget begins with an unintended `movs r0, r2` followed by `mov.w r1, #4294967295` (i.e., 0xffffffff, the 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.

Similar register-move gadgets can be devised to disclose the value of other registers in the manner described above.

VI. Mitigations

Many JIT spraying defenses were proposed [5], [10], [11], [22] in the aftermath of Blazakis' seminal work on JIT spraying. Two of them, random NOP insertion and constant blinding, are included in JSC, but their implementations fall short due to a mis-estimation of how JIT spraying can manifest itself on the ARM architecture. In this section, we review the shortcomings of JSC's JIT spraying defenses and identify additional mitigations that would greatly increase the complexity of launching a JIT spraying attack using gadget chaining.

A. Shortcomings of JSC's JIT spraying defenses

1) Random NOP insertion: Generally speaking, the idea behind random NOP insertion is to intersperse semantic NOP instructions at unpredictable locations among the intended instructions. Since Blazakis' original JIT spraying attack relies on creating long, predictable sequences of unintended instructions that are executed sequentially from start to finish, the inserted NOP instructions would derail the unintended instruction stream and cause the attack to fail.

JSC's Baseline JIT provides a rudimentary form of random NOP insertion. Instead of inserting various semantic NOP instructions randomly throughout the emitted code, it emits a single NOP instruction at the beginning of each compiled piece of code with 50% probability. The semantic NOP used is always the 16-bit NOP instruction. The DFG JIT does not perform random NOP insertion.

Since the NOP inserted is always only 2 bytes long, if the attacker guesses incorrectly regarding the presence of the randomly-inserted NOP, execution in the gadget will begin at an intended instruction boundary. Supposing the resulting intended instruction sequence does not crash the process, execution will fall through into the return sequence. If the attacker was exploiting a vulnerability that allows her to return with the intended return sequence (instead of needing self-modifying gadgets), the return sequence will correctly return control to the attacker's JavaScript controller. It is even possible for the attacker to inspect the gadget's return value to determine whether or not the unintended instruction in the gadget executed and to adjust her gadget address guess accordingly.

The read gadget is an example of a gadget that reveals whether or not the unintended instruction was successfully executed. To test whether or not her gadget address guess is correct, the attacker should try to read the word at the gadget's start address. If the returned value matches the expected first word of the gadget, then there is a high probability that the attacker's guess was correct. The attacker can add confidence to this finding by adding 1 to the memory address to be read and observing whether or not the returned word matches the

TABLE III. CONDITIONS UNDER WHICH JSC WILL NEVER BLIND A CONSTANT VALUE $\mathcal{V}$.

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

previously-returned word, except with one byte shifted out and a new byte shifted in on the other end. The read gadget can be used to detect random NOP insertion for the sake of another gadget by placing both the read gadget and the "real" gadget in the same function. Once the read gadget has revealed the presence or absence of a randomly-inserted NOP, the other gadget's address can be adjusted accordingly.

Random NOP insertion cannot disrupt gadgets which aim to execute only one unintended 16-bit instruction since it is impossible to insert an instruction into the middle of a halfword. Even if random NOPs were inserted by both the Baseline and DFG JITs at less predictable locations, we showed above that an attacker could simply probe for the gadget. One way to improve random NOP insertion is to use semantic NOPs that do not allow execution to fall through to the return sequence if execution branches into the middle of the NOP. Wei et al. [22] gave one example of such a semantic NOP in the form of a PC-relative branch followed by software interrupt instructions which trap into a security auditing routine. When executed from the branch instruction, execution will jump over the interrupt instructions, but a jump that is probing for a gadget and lands in the middle of the so-called "trapping snippet" will not fall through to the return sequence. These trapping snippets should be sprinkled throughout the JIT code instead of being placed only at the beginning of code blocks.

2) Constant blinding: Constant blinding seeks to eliminate an attacker's ability to predict the value of immediate fields in JIT-produced instructions. A canonical example is that of protecting the loading of a register with an untrusted immediate value. Rather than moving the immediate directly into the register, the XOR of the immediate and a random blinding value is moved into it. The register is then XORed against the blinding value, leaving the value of the immediate in the register since $(imm \oplus blind) \oplus blind = imm$. As a result, any immediate bits that the attacker had hoped to use as part of an unintended instruction sequence will be scrambled.

JSC performs constant blinding for many operations, but it does not blind all constants. Certain constants are considered "safe"; Table III lists the conditions under which a constant value $\mathcal{V}$ will never be blinded; if any one of the conditions is met, the constant will not be blinded. Furthermore, JSC randomly decides whether or not to blind a constant that does not meet a do-not-blind criterion, only blinding with a 1/64 probability. Avoiding increased i-cache pressure may be a motivating factor for not blinding every constant, as the blinding operation can quadruple the size of a 4-byte operation by adding up to two `mov` instructions as well as the XOR instruction.

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 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 as safe on ARM. The `R2`-disclosure gadget we presented in § V-C2 uses 16 as a constant, which is considered "safe" by JSC. In order to defend against gadget chaining, *every* constant needs to be blinded.

B. Suggested hardening

Below, we suggest defensive techniques that add to the complexity of successfully expoiting a program vulnerability with a gadget chaining attack. None of these mitigations can nullify the threat of JIT spraying on their own. Instead, they should be combined with each other and improved implementations of the defenses already provided by JSC.

1) Register randomization: Register randomization is the practice of allocating scratch registers in an unpredictable manner. The 32-bit immediate-operand instructions from which we are easily able to form 16-bit unintended instructions use a 4-bit `Rd` field, allowing the attacker to control 25% of the bits of the unintended instruction by manipulating the register into which results will be stored. If both register randomization and constant blinding are implemented for all instructions, the attacker will no longer be able to create 16-bit unintended instructions in the manner described in this paper.

Unlike constant blinding, register randomization does not increase the number of instructions. However, it may slightly increase i-cache pressure by allocating registers in such a way that a 32-bit instruction is required where previously a 16-bit instruction would suffice (due to the width of register fields in the instruction encodings). For example, the 16-bit XOR instruction with register operands may only operate on `R0-R7`; to use `R8-15`, the 32-bit encoding is required.

2) JIT allocation randomization: In § V-A1, we stressed the importance of identifying the addresses at which gadgets reside and described our technique for placing JITed functions at known offsets on every sprayed page. Eliminating the viability of this technique adds 12 bits of entropy to the location of each JITed function. At the cost of increased fragmentation and potentially more committed memory pages, an allocation policy that randomizes the base address of JITed code blocks would prevent the attacker from manipulating the allocator into placing her gadgets at known offsets.

Although it is indeed theoretically possible to probe the address space in search of a gadget using the method described in § VI-A1, the added 12 bits of entropy increase the search space by a factor of 2048 compared to the 2 possible locations offered by the Baseline JIT's random NOP insertion. Additionally, there are many places in the JIT code where a random branch will not immediately fall through into the intended return sequence. Slow path code which handles unusual type combinations is written at the end of each generated JIT code block. Branching randomly into slow path code may lead to unpredictable behavior or a crash. Finally, requiring the attacker to probe for gadgets constrains the class of exploitable control flow vulnerabilities to those that can take advantage of the intended function return sequence.

VII. Future Work and Conclusion

In this paper we demonstrate the viability of JIT spraying as a vector for exploiting an existing control flow vulnerability on an ARM system implementing both DEP and ASLR. To accomplish this, we leverage a novel JIT spray style called *gadget chaining*, which enables an attacker to augment the safe execution of code in a high level language with unsafe unintended insruction sequences that can be invoked just like subroutines. We presented an end-to-end proof of concept attack that uses gadget chaining and propose other potential uses for it. We believe our work demonstrates that RISC architectures are not de facto immune to JIT spraying as was previously thought.

Although we were unsuccessful in generating self-sustaining and Thumb-ARM reinterpretation-style JIT spray payloads on ARM with JSC's Baseline and DFG JITs, we cannot rule out the possibility that different JITs—future versions of these JITs, JSC's recently-released FTL JIT, or even JITs belonging to another language runtime entirely—will make these payload types a possibility on ARM. More importantly, however, more work is needed to determine if working analogs to the JIT spraying techniques described in this paper even exist and can be generated on other RISC architectures.

The discovery and exploration of the various JIT spraying payload types was accomplished through painstaking manual trial and error. Future work should prioritize developing the right tools for analyzing the output of of a JIT compiler and probing its capabilities. An abstraction such as REIL [12] could even be used to help automate the analysis of data flow through unintended instructions in order to generate a self-sustaining JIT spraying payload on ARM.

The extension of JIT spraying to the ARM architecture challenges the assumptions made by JSC's JIT implementation, and there are very likely other JITs that fail to protect themselves against gadget chaining. JIT spray mitigation implementations should be revisited and, should the security benefits be found to outweigh the performance costs, revamped to defend against this new threat.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1228967.

References

[1] Aleph One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, p. 365, 1996.

[2] 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.

[3] ARM Holdings, "Annual report and accounts 2010," http://media.corporate-ir.net/media_files/irol/19/197211/626-1_ARM_AR_040311.pdf, 2010.

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

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

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

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

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proceedings of CCS 2008*. ACM Press, Oct. 2008, pp. 27–38.

[9] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. 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, Aug. 2009.

[10] P. Chen, Y. Fang, B. Mao, and L. Xie, "JITDefender: A defense against JIT spraying attacks," in *Future Challenges in Security and Privacy for Academia and Industry*. Springer, 2011, pp. 142–153.

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

[12] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," Presented at CanSecWest 2009. Online: http://www.zynamics.com/downloads/csw09.pdf, Mar. 2009.

[13] T. Kornau, "Return oriented programming for the ARM architecture," *Master's thesis, Ruhr-Universitat Bochum*, 2010.

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

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

[16] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of CCS 2007*. ACM Press, Oct. 2007, pp. 552–61.

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

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

[19] 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.

[20] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of IEEE Security and Privacy ("Oakland") 2013*. IEEE Computer Society, 2013, pp. 574–88.

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

[22] T. Wei, T. Wang, L. Duan, and J. Luo, "Secure dynamic code generation against spraying," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 738–740.