NoJITsu: Locking Down JavaScript Engines

Taemin Park*, Karel Dhondt†, David Gens*, Yeoul Na*, Stijn Volckaert†, Michael Franz*

*Department of Computer Science, University of California, Irvine
†Department of Computer Science, imec-DistriNet, KU Leuven

***Abstract*—Data-only attacks against dynamic scripting environments have become common. Web browsers and other modern applications embed scripting engines to support interactive content. The scripting engines optimize performance via just-in-time compilation. Since applications are increasingly hardened against code-reuse attacks, adversaries are looking to achieve code execution or elevate privileges by corrupting sensitive data like the intermediate representation of optimizing JIT compilers. This has inspired numerous defenses for just-in-time compilers.**

Our paper demonstrates that securing JIT compilation is not sufficient. First, we present a proof-of-concept data-only attack against a recent version of Mozilla's SpiderMonkey JIT in which the attacker only corrupts heap objects to successfully issue a system call from within bytecode execution at run time. Previous work assumed that bytecode execution is safe by construction since interpreters only allow a narrow set of benign instructions and bytecode is always checked for validity before execution. We show that this does not prevent malicious code execution in practice. Second, we design a novel defense, dubbed NoJITsu to protect complex, real-world scripting engines from data-only attacks against interpreted code. The key idea behind our defense is to enable fine-grained memory access control for individual memory regions based on their roles throughout the JavaScript lifecycle. For this we combine automated analysis, instrumentation, compartmentalization, and Intel's Memory-Protection Keys to secure SpiderMonkey against existing and newly synthesized attacks. We implement and thoroughly test our implementation using a number of real-world scenarios as well as standard benchmarks. We show that NoJITsu successfully thwarts code-reuse as well as data-only attacks against any part of the scripting engine while offering a modest run-time overhead of only 5%.

I. Introduction

Browsers are among the most widely used programs and are continuously exposed to untrusted inputs provided by remote web servers. A substantial part of these untrusted inputs is JavaScript code. Browsers generally use a script engine with one or more Just-In-Time (JIT) compilers to execute scripts efficiently. Mainstream engines such as Mozilla's SpiderMonkey and Google's V8 evolve rapidly and grow continuously to keep up with the latest ECMAScript standard and with the users' demand for high performance. Consequently, they are prime targets for adversaries who exploit this increasing complexity and flexibility to gain remote code execution in the browser process [49], [66].

Initially, these exploits focused on the JIT compiler itself. This compiler transforms interpreted bytecode into natively executed machine code. When JavaScript JIT compilers first became popular, they wrote all run-time generated code onto memory pages that were simultaneously writable and executable throughout the execution of the script. This trivially enabled code-injection attacks [18], [55]. Later JIT engines added support for W⊕X policies by doubly-mapping JIT pages instead. This meant that JIT code could no longer be found on memory pages that were simultaneously writable and executable. While this undeniably improved security, attackers repeatedly demonstrated that JIT engines could still be attacked. *JIT spraying*, for example, lets an attacker inject small arbitrary instruction sequences into JIT pages without writing directly to the pages [7], [13], [37]. Defenders quickly thwarted these attacks through the use of constant blinding [13], constant elimination and code obfuscation [19], code randomization [32], or control-flow integrity [46].

Successfully defending JIT engines against code-reuse attacks proved more challenging, however, since an adversary can leverage memory disclosure vulnerabilities to iteratively traverse and disassemble code pages to dynamically generate a ROP chain at run time (an attack known as JIT-ROP [56]). A number of schemes protect against such attacks by leveraging randomization and execute-only memory [8], [9], [23], [29].

More recently, several efforts independently demonstrated that an adversary may still be able to inject code despite all of the above defenses being in place by resorting to data-only attacks. Both Theori et al. [62] and Frassetto et al. [27] showed that an attacker can force the JIT compiler to generate malicious code by corrupting the intermediate representation of the compiler without overwriting any code pointers. For this reason, recent defenses propose isolating the compilation from the execution of JIT code, through separate processes [42], [58] or hardware-based trusted execution environments [27].

In this paper, we show that isolating JIT code compilation from its execution does not suffice to prevent remote code execution. To this end, we first present a new attack that only leverages the bytecode interpreter component of the scripting engine. Previous work considered this component safe by design, since the bytecode is confined to a limited set of operations whose safety is validated by the interpreter before they are executed. We show that this assumption does not hold in practice, as we can corrupt the internal data representation of individual operations within the interpreter. This allows us to execute potentially malicious operations such as arbitrary system calls. Crucially, our new attack *does not* require JIT compilation of bytecode at any point in time. We implemented our proof-of-concept attack for a recent version of Mozilla's

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24262
www.ndss-symposium.org

popular and widely used JavaScript engine SpiderMonkey to verify its efficacy.

Unfortunately, previously proposed protections for JavaScript engines do not trivially extend to the bytecode interpreter, as their design is either tailored towards running in a trusted execution environment, or because they would incur substantial run-time overhead in the context of an interpreter. This is why we present a novel and general defense strategy, called NoJITsu, to defend the JIT engine against a wide variety of run-time attacks, including code injection, code reuse, and data-only attacks against any of its components. Our design leverages hardware-based memory protection features, such as Intel MPK, to isolate and protect each component of the scripting engine. In this way, we are able to effectively reduce the memory-access permissions of each component towards its minimally required working set. To demonstrate feasibility we then analyze, partition, and instrument SpiderMonkey, leveraging automated dynamic analysis techniques to scale our efforts to this complex real-world code base, while keeping our techniques implementation-agnostic. To the best of our knowledge, we are the first to present and fully implement hardware-backed, fine-grained access control for a JavaScript engine. We thoroughly tested and evaluated NoJITsu in a number of attack scenarios, which include code-injection, (dynamic) code-reuse, as well as data-only attacks, and analyzed its security in depth. Using standard benchmarks as well as real-world application scenarios we show that our prototype already offers practical performance, with a moderate run-time overhead of only 5% on average.

In summary, our contributions are as follows:

- **Bytecode Interpreter Attack.** We present a new attack against bytecode interpreters in modern JavaScript engines which have not been targeted in previous work. Our attack therefore works despite all existing defenses being enabled and enforced.
- **Fine-Grained Memory Access Control.** We propose NoJITsu, a novel approach that effectively secures the bytecode interpreter component of modern scripting engines. Our design leverages enhanced memory protection, such as Intel MPK, to completely lock down the entire scripting engine. To the best of our knowledge, we are the first to incorporate fine-grained memory access control into a large real-world JIT engine and also protect the JavaScript interpreter component. We implemented our prototype in a recent version of Mozilla's SpiderMonkey.
- **Thorough Evaluation.** We extensively evaluate our prototype of NoJITsu for its security using a number of real-world attack scenarios, for which we also re-implemented a fully working JIT-ROP attack against SpiderMonkey. As we are able to show NoJITsu withstands all previously presented attacks as well as our new data-only attack. We further evaluate performance using standard benchmarks and practical use cases, demonstrating that NoJITsu additionally offers low overhead with an average performance hit of only 5%.

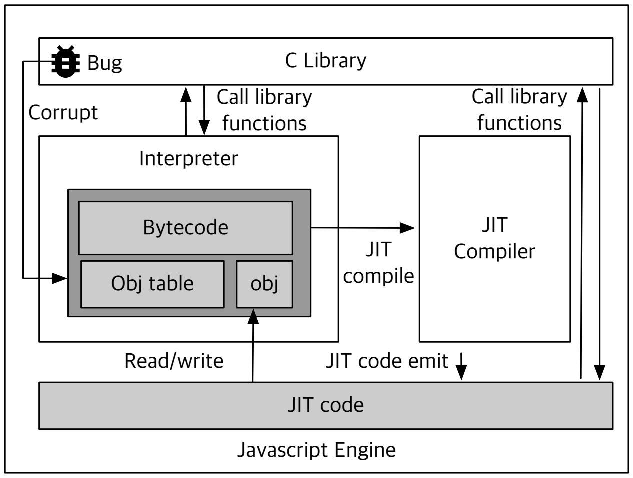


Fig. 1: High-level overview of our model. If an adversary is able to trigger a memory-corruption vulnerability in any part of the JIT engine, we show that the internal data used by individual bytecode operations can be exploited to make the interpreter call *external* system functions, which are always mapped as part of the application's address space. This strategy works despite state-of-the-art defenses for JIT engines being deployed.

II. Attacking the Interpreter

We constructed an attack on the interpreter component of Mozilla's SpiderMonkey, the JavaScript engine used in the Firefox web browser. This section provides the necessary background information and assumptions about SpiderMonkey's internals and then proceeds to describe our attack.

A. Threat Model

We assume a recent version of SpiderMonkey built with the standard Mozilla build environment and configuration. SpiderMonkey has many components that contain machine code compiled ahead of time (statically). We assume that at least one of these components contains an arbitrary memory read/write vulnerability.

We assume that the standard code-injection and code-reuse defenses are in place. Hardware vulnerabilities such as Rowhammer [31], [52], Meltdown [38], and Spectre [33] are orthogonal to software vulnerabilities and outside the scope of this paper. Our threat model is in line with those of related work in this area [7], [13], [19], [27], [32], [37], [46].

- **Memory-corruption vulnerability.** Some part of the scripting engine (or the surrounding application) contains a memory-corruption bug that enables an adversary to arbitrarily corrupt any part of the program's address space.
- **Code-injection defense.** We assume the scripting engine enforces a strong W⊕X policy [41], and, thus, that no memory pages are ever simultaneously writable and executable. Some engines enforce W⊕X by offloading the JIT compilation to an external process [42], [58] or trusted execution environment [27],

while others simply toggle the writable and executable permissions on JIT pages at run time [44].

- **Code-reuse defense.** We assume that the browser uses all code-reuse defenses that have seen widespread adoption. These defenses include, among others, ASLR [48] and coarse-grained CFI [3]. With these defenses in place, the base addresses of executable code sections are not known a priori, and control flow can only be diverted to legitimate function entry points. This set of defenses, however, doesn't prevent leaking function addresses by disassembling code pages on-the-fly to directly discover function locations encoded in the code pages or indirectly read a location of data structures such as PLT that contain legitimate function entry points.
- **Hardware-based Memory Protection Features.** We assume Intel Memory Protection Keys (MPK) [22] to be available on the target platform. We assume PKRU values, which control access privileges to memory domains, always stay in registers so adversaries with arbitrary memory read/write capability cannot directly manipulate these values. As coarse-grained CFI is in place, the attacker cannot leverage unintentional occurrences of instructions to modify in-register PKRU values.

B. SpiderMonkey Implementation

Modern scripting engines have at least two components that support the execution of JavaScript code: an interpreter and a JIT compiler (see Figure 1)[1] The interpreter takes a plain-text script as input, parses it, and generates bytecode instructions, object tables, and data objects. The data objects encapsulate all of the data used throughout the execution of the bytecode program. For example, this includes constant values, function arguments, local and global variables, properties, and function pointers. The object tables form an indirection layer between the bytecode and the data objects. Thanks to this indirection, the bytecode can refer to data objects using their index in an object table. This allows the JavaScript engine to generate highly compact bytecode. The engine then executes the script by interpreting the bytecode. When the interpreter executes a particular part of the bytecode often enough (i.e., the bytecode becomes "hot"), it invokes the JIT compiler, which compiles the bytecode into optimized native machine code. Among other things, this eliminates the overhead of interpreter distpatch.

1) Speculative Optimization: Some of the optimizations leveraged by the JIT compiler might be speculative in nature. The compiler might, for example, speculate that the types of certain program variables remain stable, when in principle types can change at any point. If one of the speculative assumptions does not hold during execution, the optimized code is de-optimized and execution falls back to the interpreter. To make the transition between interpreted execution and JIT code execution seamless, the interpreter and JIT compiler share many data structures and memory regions. For example, program variables, are stored in data objects, regardless of where the script is executing. Other data structures and memory regions might be used exclusively by one of the two components.

2) Native Functions: During its execution, a script may call C++ functions that are registered as so-called *JavaScript native* (JSNative for short) functions. SpiderMonkey has hundreds of JSNative functions. They provide the functionality of the built-in JavaScript types and operations. In many cases, calls to JSNative functions are not inlined, even when the caller is a JIT-compiled function. Instead, SpiderMonkey transfers control to the JSNative function using a regular function call. One important property of JSNative functions is that SpiderMonkey calls them using an internal calling convention. According to this calling convention, a JSNative function must receive a pointer to the global JavaScript context object as its first argument, an argument count object as its second argument, and a pointer to an argument array as its third argument. Within the argument array, there is one slot that is reserved to store the return value of the JSNative function. However, upon calling a JSNative function, SpiderMonkey stores a pointer to the callee's JavaScript function object in the return address slot.

3) Data Structures: Throughout our analysis of SpiderMonkey's implementation, we identified a number of key memory areas that play a crucial role in ensuring the correct and secure operation of the script engine: (1) the bytecode region, (2) the JIT code cache, (3) the JIT compiler data, (4) the JavaScript data objects, and (5) the object tables. The JIT code is mapped as an executable memory region whereas all the other areas are mapped as readable and writable regions. The bytecode region includes instruction opcodes, and operands, and is used by both the interpreter and the JIT compiler. The JIT code cache indicates the instructions generated by the JIT compiler, including normal JIT code and inline cache stubs. The JIT compiler data region includes JIT compiler-specific intermediate representations of the bytecode (i.e., the MIR and LIR code in SpiderMonkey's case) and other data structures that are used exclusively by the JIT compiler. The JavaScript objects are all kinds of objects that are backed by a garbage collector, and are used by both components. Several of these memory regions have been targeted by control-flow hijacking attacks in the past. Previously presented exploit mitigations typically protect either the JIT code cache or the JIT compiler data, but leave the other regions exposed. In practice, this was shown to be exploitable via crafted inputs that trigger type confusion in the engine's arena allocator [53]. Next, we will demonstrate that an adversary can construct successful exploits by attacking the interpreter, bypassing these proposed mitigations.

C. Our Interpreter Attack Against SpiderMonkey

We present a new attack that leverages the fact that most of the script engine's key memory regions remain writable throughout the execution of the script. A memory-corruption bug in any of the engine's components therefore allows us to manipulate any of the interpreter's data structures. We also exploit the extensive use of indirection in the bytecode. Aside from program variables, JavaScript objects also encapsulate function information. When one function calls another, the caller loads the callee's address from the callee's function

[1] Note that all three major JavaScript engines, SpiderMonkey, JavaScriptCore, and V8, have bytecode interpreters in their script execution pipeline. V8 has recently added the interpreter to reduce memory overhead in mobile environments.

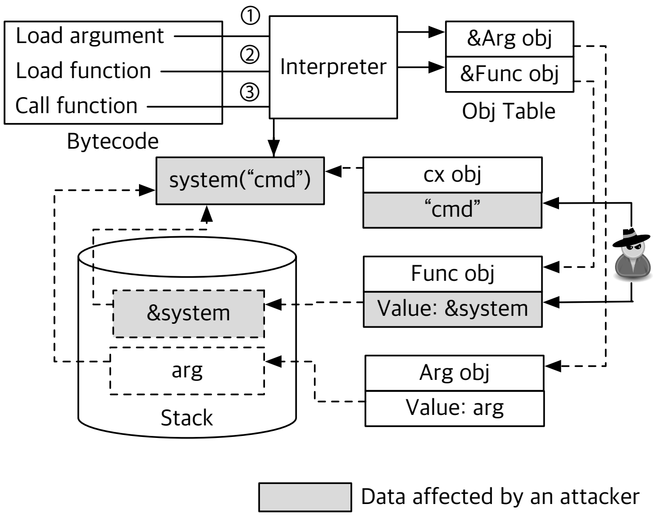


Fig. 2: Our attack proceeds in three steps: ① the attacker locates the JavaScript context object and the function object for a victim function. ② the attacker then corrupts the function object to point to an attacker-chosen target function and injects a command string in the context object. ③ finally, the attacker calls the victim function from JavaScript, causing the interpreter to invoke the target function with the injected argument string.

object. We show that we can execute arbitrary shell commands by locating and corrupting these function data objects at run time.

We successfully tested our exploit against a recent version of SpiderMonkey 60.0.0. Our attack proceeds in three key phases, which are illustrated in Figure 2. First, the attacker locates the JavaScript context object and the JavaScript function object of a victim function (i.e., any function we can call from JavaScript). After leaking the locations of these two objects, the attacker overwrites the function address contained in the function object with the address of a target function. We used the C library's `system` function as the target function for our attack. The attacker also overwrites the contents of the context object to hold a string representation of the path to the desired program to be executed (e.g., "/bin/sh"). Finally, the attacker invokes the victim function. The interpreter then loads the modified objects onto the stack and launches the program specified in the argument string encoded by the corrupted context object.

1) Implementation Details: We implemented the first step of the attack by exploiting a type confusion bug (CVE-2019-11707) present in SpiderMonkey versions 60.8.0 and below. This bug can be weaponized into a full-fledged arbitrary read-/write primitive, as was shown in related work [12]. Weaponizing the bug takes four steps. First, the JavaScript program allocates a number of small and consecutive `ArrayBuffers` on the heap. We gave all of the `ArrayBuffers` a size of 32 bytes in our exploit. Then, the program creates `Uint32Array` and `Uint8Array` view objects for one of the allocated `ArrayBuffers`. Next, the program triggers a type confusion between the two view objects. After triggering the type confusion, SpiderMonkey allows the program to read-/write 32 `Uint32` elements in the ArrayBuffer. Since the `ArrayBuffer`'s size is only 32 bytes, the program can now overwrite the metadata stored in one of the adjacent `ArrayBuffers`. Finally, the program overwrites the data pointer, which is part of the adjacent `ArrayBuffer`'s metadata, with a pointer to a memory location chosen by the attacker. All subsequent accesses to the adjacent `ArrayBuffer` now target this attacker-chosen location.

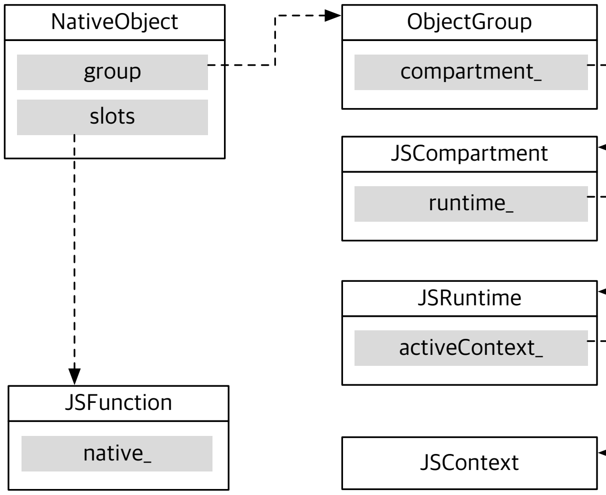


Fig. 3: Disclosing the locations of the victim JSFunction object and the JSContext object.

After weaponizing the bug, we leak the locations of the `JSContext` and victim `JSFunction` objects as illustrated in Figure 3. We start by reading the contents of the `NativeObject` struct which is embedded in the `ArrayBuffer` we just corrupted. From the `NativeObject` struct, we follow a pointer chain to the global `JSContext` object. Next, we write a reference to the victim JavaScript function into the `NativeObject` struct. We then use the memory vulnerability to read the raw value of this reference, thus revealing the location of the victim's `JSFunction` object.

We locate the target function itself by reading the current value of the native function pointer within the victim `JSFunction` object and by recursively disassembling the native function until we arrive at a call to a PLT entry, which we disassemble to find the start of the PLT. We then find the PLT entry of the target function using a priori knowledge of the layout of the PLT.

2) Discussion: As a remote attacker, launching an interactive shell session from within SpiderMonkey might not be advantageous. However, the attacker could also inject and pass a script to the terminal, e.g., by encoding it as a cookie file, which would then require passing the relative path to the cookie file on to the shell. In our tests, we were able to corrupt up to 32 bytes of the context object without causing the interpreter to crash, which leaves plenty of room for storing useful payloads in memory.

Crucially, our new attack cannot be prevented by previously proposed defenses tailored towards protecting the JIT compiler data [27], [42] since we attack the interpreter which always executes before any JIT compilation is invoked.

III. NoJITsu: Protecting JIT Engines

Motivated by the fact that state-of-the-art JIT defenses fail to stop attacks that target interpreted bytecode, we designed a novel defense that provides fine-grained memory protection to lock down real-world scripting engines. As switching between interpreted and JIT'ed code happens frequently (i.e., on a per-function basis) an efficient implementation of this mechanism is key to overall run-time performance. Hence, we cannot simply move the interpreter out-of-process as previously proposed for the JIT compiler [42]. Instead, our design leverages automated dynamic and static analysis to restrict memory-access permissions within the scripting engine to the bare minimum. This way, NoJITsu protects against a wide range of possible attacks, including code-injection, code-reuse, and data-only attacks. NoJITsu is designed to be compatible with and usable alongside existing defenses such as constant blinding [13], constant elimination and code obfuscation [19], code randomization [32], or control-flow integrity [46].

A. Overview

Our main goal is to enforce fine-grained security policies for different kinds of data structures in JavaScript. While limited policies may already be in place for code sections, current JIT engines do not distinguish between different kinds of data sections and have naive or no explicit security policies for them within the application's address space. In Figure 4, the JavaScript engine stores the bytecode, object tables, and JavaScript objects in writable memory regions for their entire lifetime, even though the engine rarely overwrites these data structures. This enables an adversary to manipulate the data structures and change the behavior of the JIT engine at run time, which can ultimately grant the adversary arbitrary code execution capabilities. Just-in-time code on the other hand is usually protected by mapping code regions as readable and executable, re-mapping it as writable temporarily when generating new JIT code. Unfortunately, this does not defend against more sophisticated attacks such as just-in-time code-reuse attacks that chain gadgets injected into the JIT code.

Our defense neutralizes these threats by deploying fine-grained security policies to lock down access permissions for each of the main data regions we identified based on their lifetime and usage within the JIT engine. Concretely, we force the JIT engine to store bytecode, object tables, and JavaScript objects in read-only memory, and to only grant write access when, where, and as long as it is needed. We do this by placing unrelated data structures into different memory domains, and by activating the write permission of a specific domain only when the subsequent code may write to the structures in that domain, revoking the permission shortly afterward. We additionally mark JIT code regions as execute-only, meaning that attacks that involve reading code (such as JIT-ROP) are no longer possible.

Figure 5 illustrates how our defense works conceptually. First, we ensure that every data structure is allocated with the correct memory permissions. We do this by assigning each type of data structure to a unique memory domain, and to associate every newly allocated data structure with the key corresponding to its data type. We also separate data structures upon allocation so that no memory page ever contains structures from multiple domains. Second, we infer the permissions each function in the engine needs based on the types of data it may access. For example, our design enforces read-only permission for all JavaScript objects to avoid adversarial data corruption, but there are times when the legal program flow requires writing to a data object. In such a case, we temporarily grant write permissions so intended program behavior remains intact. To identify the locations that require such a temporary permission relaxation, we dynamically analyze possible accesses to each object. Finally, we insert instrumentation code that sets the appropriate domain permissions at the locations identified by our dynamic analysis.

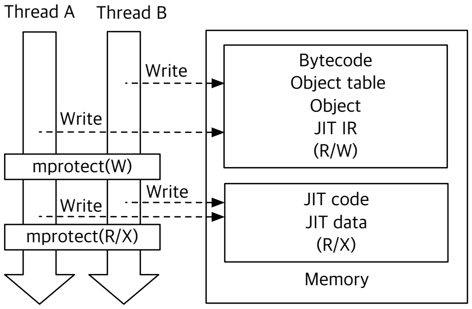


Fig. 4: Legacy design

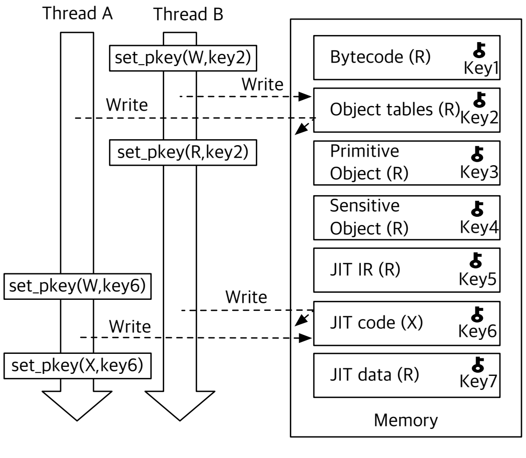


Fig. 5: NoJITsu

Figure 6 shows the overall design of our JavaScript engine protection, NoJITsu, which separates core data structures of the engine into different protection domains. There are several challenges we needed to overcome to implement our design. First, restricting memory access entails the recurring modification of access permissions on data structures, which can be costly. To solve this problem, we utilize a hardware-based mechanism that allows us to change access permissions

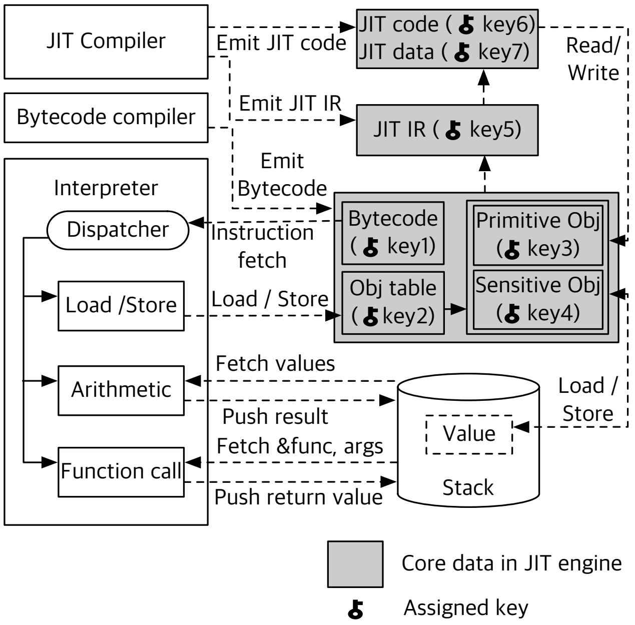


Fig. 6: Design of our script engine protection NoJITsu. Core data structures in the JIT engine —bytecode, object tables, objects, JIT IR, JIT code, and JIT data— are separated into different protection domains. We make the JIT code regions execute-only and the other data regions read-only. We grant the write access to the protected regions only when a legitimate program flow requires write permission. For instance, we temporarily grant write permission to the JIT code region when the compiler emits newly generated JIT code.

for individual memory domains without modifying page table entries or flushing the TLB. Hence, in contrast to traditional MMU-based protection mechanisms [8], [9], [23], [29], we can change access permissions without incurring substantial run-time overhead. Second, existing implementations of execute-only memory (XoM) do not apply to JIT code. Extending support to JIT code is not trivial, as the JIT engine might emit JIT code containing data (such as object tables) that must remain readable at all times. Our defense separates this data from the JIT code so that we can safely revoke read access to all JIT code regions. To the best of our knowledge, we are the first to implement execute-only support for JIT code.

1) JIT Code: The JIT code cache contains dynamically generated instructions that natively execute on the CPU. To defend against code injection attacks, it is important to keep this JIT code cache non-writable except when the instructions are generated. The JIT code cache also needs to be non-readable to avoid JIT-ROP attacks which require reading code regions to discover code-reuse gadgets at run time. In the original design of the JavaScript engine, however, the JIT code cache must be readable because it also contains readable data such as constant values, which are too wide to be embedded into instructions as immediate operands, and target addresses read by a jump table. To make the JIT code region execute-only, we first separate these readable data from the JIT code region. We move all readable data including constant values and the jump table targets into a dedicated, read-only region such that the JIT code cache is only composed of executable machine instructions. We carefully design this data separation to minimize the potential performance impact (see Section IV-B1). After separating JIT data from JIT code, we make the code execute-only and the data read-only. This protects the engine against JIT spraying attacks (which rely on injecting constants into the JIT code) [13] and JIT-ROP attacks (which rely on reading code) [56].

Our defense provides clear added value to other countermeasures against these attacks. Constant blinding, for example, also defends against JIT spraying attacks, but existing implementations do not blind small constant values (of less than three bytes) for performance reasons [40]. JIT spraying attacks using one- and two-byte constants are, however, feasible in practice [7]. Similarly, there are several existing implementations of execute-only memory, but they only apply to statically generated code [8], [9], [15], [21], [23], [29], [43]. This leaves these implementations vulnerable to JIT-ROP attacks that only use gadgets found in the JIT code cache.

2) Static Code: The static code regions include the code sections of the JavaScript engine itself and the dynamic libraries that the engine loads into the memory. Unlike the JIT code region, the attacker cannot inject malicious code into this static code region by running a malicious script. However, the static code region consists of a large code base which nearly always contains an abundance of code-reuse gadgets. Similar to the JIT code region, we make static code regions execute-only so the attacker cannot disclose executable memory regions to chain code-reuse gadgets. We leverage eXecutable-Only Memory Switch (XOM-Switch) [43] to enforce execute-only permissions for the static code regions.

3) JIT IR: JIT IR is an intermediate representation used during compilation of bytecode into JIT code. While this IR code has a short lifetime, researchers have demonstrated attacks that exploit race conditions to corrupt the IR code from another thread [27], [58]. Our defense protects the JIT IR code by granting write permission only to the thread that compiles the IR code into machine code. The attacker, therefore, cannot manipulate JIT IR using another thread unless that thread is also compiling IR code when the exploit takes place.

4) Bytecode and Object Tables: Similar to the JIT code cache, the bytecode and object tables should be writable only when they are generated during compilation. After their generation, the bytecode and object tables are only read throughout the remainder of the execution. Thus, we allow write access to these data structures only when the script parser generates them and immediately make them non-writable afterwards.

5) JavaScript Objects: Unlike bytecode and object tables, which must be written only once, data objects can be written to frequently at any point of the program execution. For example, a data object that contains a program variable may be overwritten at any time during the program execution. Moreover, every JavaScript object contains several kinds of flags that must be frequently updated, e.g., a reference counter for the garbage collector. Consequently, identifying all such locations that require permission changes for data objects is challenging. We therefore propose a dynamic analysis technique that automatically identifies permitted write operations for each data object (see Section IV-C2).

We separate JavaScript objects into two protection domains depending on the types they encapsulate: one for sensitive data objects and the other for primitive data objects. We consider an object sensitive if it contains sensitive information such as function pointers, object shape metadata, scope metadata, or JIT code. Corrupting a sensitive object allows the attacker to seize control over the JIT engine immediately. For example, primitive data objects may contain integers, characters, or arrays. Corrupting a primitive data object typically does not suffice to seize control over the engine, but it may be useful to subsequently corrupt a sensitive object. By separating sensitive objects from primitive objects, we can ensure those object classes are not writable at the same time. Thus, the attacker cannot leverage an object type confusion vulnerability to corrupt sensitive objects using a write operation for primitive data types. Moreover, we can further narrow down the writable time windows for each object type.

Note that the objects can also be written during the JIT code execution. Changing object permissions during the JIT code execution, however, may introduce substantial run-time overheads as the JIT code is generated for frequently executed code, and hence, highly optimized. Therefore, we lift all access restrictions to *primitive* data objects while JIT code executes, and enable the protection again when the JIT code transfers control back to the JavaScript engine itself. We do, however, enforce protection for sensitive data objects even during JIT execution. This way, the attacker can no longer manipulate sensitive objects (e.g., Shapes, Cells, Functions) which are frequently exploited in real-world attacks. Interestingly, write accesses to sensitive objects are rare during the JIT execution. One exception is the lambda function object whose properties can change dynamically. For this case, we instrument the JIT code region to grant valid write accesses to sensitive objects and then enable the protection again, as we do for the static code region.

IV. Implementation

We applied our defense to SpiderMonkey 60.0, which was released in late 2018 [45]. We modified the source code for SpiderMonkey's memory allocation routines to associate the correct domain keys with each structure and to ensure that different types are allocated on separate memory pages. We also instrumented all code locations that require write access to the bytecode, object tables, JIT IR, JIT code, and JIT data to enable and disable write access to the appropriate domains. To separate the JIT code from JIT data, we modified the JIT linker and assembler code. Lastly, we modified SpiderMonkey's signal handlers to support our automated instrumentation of data object accesses and to support our dynamic object-flow analysis (see Section IV-C2), which we conducted offline. For this last step, we used LLVM 8.0.0 to modify and transform SpiderMonkey's code automatically [35]. In total, our prototype consists of 9000+ added lines of code. We also wrote 1000+ lines of LLVM code and 200+ lines of Python scripts, which are used for processing results from our dynamic object-flow analysis.

A. Memory Protection Mechanism

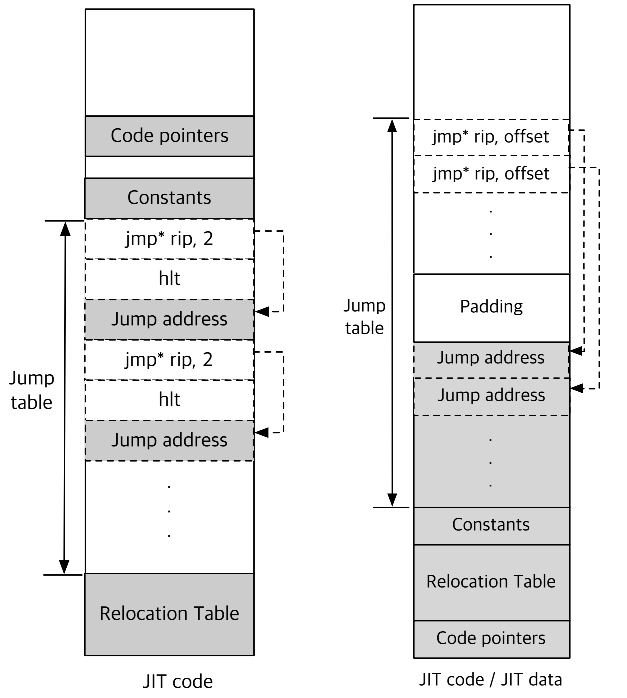
We implemented our domain-based access control on top of Intel Memory Protection Keys (MPK). MPK is a recently introduced hardware feature that allows user-space programs to manage access permissions for up to 16 memory domains. To change the access permissions for a domain, the program uses an unprivileged instruction to write to the thread-local PKRU register. Note that, while the PKRU write instruction is unprivileged, an adversary has to acquire arbitrary code execution to set its value. However, as we demonstrate in Section V, NoJITsu provides protection against a wide variety of attacks, including arbitrary code execution attacks based on code injection and code-reuse.

B. JS Engine Compartmentalization

NoJITsu enforces an execute-only memory policy for JIT code regions and statically generated code. This policy thwarts JIT-ROP attacks which rely on reading code to discover code-reuse gadgets. We use MPK to implement this policy. However, MPK, by itself, does not suffice to implement the policy because it can only toggle the write and read permissions through the PKRU register. To make JIT code execute-only, we therefore allow SpiderMonkey to allocate the JIT code cache onto memory pages that are marked as executable in the page table. We use MPK to make these pages readable and writable during JIT compilation, and to revoke read and write permissions when the compilation completes.

1) Jump Table Separation: During JIT compilation, SpiderMonkey emits jump tables such as those shown in Figure 7a. The `jmp* rip, 2` instruction loads a jump target address located at offset 2 from the jump instruction, and jumps to the loaded address. We modified SpiderMonkey to separate jump target addresses from the rest of the jump table. This allows us to make jump addresses read-only and not executable. Blindly moving the addresses into read-only memory would require us to reserve an additional register to store the base address of the jump target region. This could lead to additional register spilling, which would negatively impact the run-time performance of the JIT engine. We avoided this performance hit by designing the JIT code layout in such a way that the data region directly follows the JIT code region. This way, the jump target can still be loaded via relative addressing, without allocating an additional register. Figure 7b shows the layout of the JIT code after jump table separation. The jump addresses, constants, and any other data are separated from the JIT code and moved into a new JIT data section that immediately follows the original JIT code section. The jump instructions are patched accordingly.

2) Permission Change Routine: Listing 1 illustrates how we temporarily change the permission for a permitted write access. Before writing to a protected region, we insert a call to `set_pkru` to change the value of the `PKRU` register to enable write access. Although `write_pkru` is a simple register write operation and much more efficient than calling `mprotect` to change the page access permission, this instruction still takes longer to execute than a normal arithmetic instruction (i.e., a `WRPKRU` instruction takes around 20 cycles because it flushes the CPU pipeline to prevent a potential memory access violation caused by out-of-order execution [47]). Thus, instead of immediately writing to the `PKRU` register, the `set_pkru` function first checks if the current `PKRU` value already has the write permission. If so, the function returns without overwriting the register. If the page does not have the write



(a) Legacy Memory Layout (b) NoJITsu Memory Layout

Fig. 7: Memory layouts before and after JIT code (▭) and JIT data (▨) separation. We move all readable data —code pointers, constants, jump target addresses, and the relocation table— into a separate memory region which immediately follows the original JIT code region. This data separation allows the JIT code region to be execute-only and the JIT data region read-only.

permission, the function overwrites the PKRU register to allow the subsequent write access and returns the previous PKRU value which will later be used for PKRU value recovery. After the write instruction, we call recover_pkru to recover the previous PKRU value.

Note that values we load into the PKRU register are encoded directly into the machine code as immediate values, and thus these values are, in principle, never loaded from memory. However, the compiler could still spill PKRU values to the stack. Addressing this corner case is out of scope of this paper, however, potential mitigations orthogonal to our work are: i) to avoid spilling of registers containing PKRU values by assigning it a dedicated register [25], or ii) to randomize the PKRU values before being spilled to stack and keep the randomization secret in a dedicated register [11], [16].

3) JavaScript Object Protection: In the JavaScript engine, the garbage collector (GC) is responsible for allocating and reclaiming JavaScript objects on the heap. The GC mechanism in SpiderMonkey already provides a certain level of data isolation through *compartments*. JavaScript objects from the same origin (i.e., objects created from the same website) are within the same compartment, and JavaScript objects (including JavaScript function objects) are not allowed to access other objects from a different compartment. However, SpiderMonkey only enforces this isolation at the language level. An adversary that finds a memory vulnerability can still access (and potentially overwrite) JavaScript objects in other compartments. We propose a low-level and precise access control mechanism for JavaScript objects that works even in the presence of memory vulnerabilities. The JavaScript objects that we protect also include *shapes* which contain the object's layout information, *script objects* that point to bytecode, and any other objects that are allocated and collected by the garbage collector (GC).

In SpiderMonkey, the unit of memory managed by the GC is called a *cell*. Cells are classified based on their *allocation kind*, which determines the attributes of the object such as the size and the finalization behavior. *Arenas* are the memory allocation unit (i.e., 4 KB) that accommodates objects of the same allocation kind. In our design, we assign the key allotted for JavaScript objects based on its allocation kind (i.e., the originating arena). For instance, in our prototype we currently support different MPK domains for sensitive types (script, shape, function, etc.) and primitive types (scalar and array data types). Separating these sensitive and non-sensitive objects into different domains, the attacker cannot exploit vulnerability in array or any other non-sensitive data objects to corrupt sensitive ones such as a function object. In Figure 6, for example, we assign key3 and key4 for different types of JavaScript objects.

```
set_pkey(protection, key){
  current_pkru = read_pkru()
  if(need_to_change_protection(current_pkru,
                               protection, key)) {
      write_pkru(current_pkru, protection, key)
      return current_pkru
  }
  return 0
}
recover_pkru(pkru) {
  if(pkru) {
      write_pkru(pkru)
  }
}
function_A(...) {
  ...
  ...
  saved_pkru = set_pkru(W,key)
  instruction to write on MPK protected region
  recover_pkru(saved_pkru)
  ...
  ...
}
```

Listing 1: Permission change routine

C. Instrumenting Memory Accesses

After modifying SpiderMonkey to separate JIT code from JIT data, and modifying the memory allocation routines to allocate each data structure into the proper memory domain, we had to instrument all memory write instructions in the JIT engine to set the appropriate run-time memory permissions based on which data types the instruction may access.

Manually instrumenting all instructions that require write permission is infeasible given the complexity of JavaScript engines. We therefore implemented a mechanism to automatically identify and record the write accesses introduced by legal program flow, and we used the recorded information to refine our instrumentation code in a subsequent step.

Alternatively, we could have used a points-to analysis technique to identify write operations that may write to a JavaScript object. However, such analysis techniques are known to overapproximate the set of operations that may write to a specific memory location because they lack run-time information and/or trade precision off for scalability. Using overly conservative analysis outputs to support our code instrumentation would result in larger attack windows in which illegitimate write accesses to JavaScript objects are possible.

During the course of this research, we studied several LLVM-based pointer analyses and found these tools have known implementation bugs that cause false negatives (i.e., missing alias relationships) in the analysis output [36], [50], [60]. For instance, these tools miss tracking pointers passed as an element of structure type (aggregate) registers. Instrumenting SpiderMonkey based on analysis outputs with false negatives would lead to missing run-time permission changes and would cause legitimate object write accesses to fail and crash the program.

1) Code Transformation and Signal Handler: Our dynamic analysis intentionally traps write accesses to the protected region and catches the resulting segmentation faults in a custom signal handler. This signal handler records the trap location, temporarily enables write access, and restarts the faulting instruction.

```
write instruction
current_pkey = read_pkey()
if(isChanged(current_pkru))
{
  print function_name
  set_pkey(R, OBJ_key)
}
```

Listing 2: Code transformation for dynamic analysis

```
sig_handler()
{
  if(CausedByMPKViolation())
  {
      set_pkey(W, OBJ_key)
      return
  }
  else
      goto:legacy sig handler
}
```

Listing 3: Signal handler for dynamic analysis

2) Dynamic Object-Flow Analysis: Listing 2 and Listing 3 shows the code transformation and signal handler we use in our analysis. At the start of the analysis, we only grant read permission to JavaScript objects. When the JavaScript engine encounters a write access to the objects, a segmentation fault will trigger our signal handler. If our signal handler identifies that the fault is caused by a MPK violation it logs the faulting code to be processed later by our LLVM passes.

Within the signal handler, we modify the PKRU value so that we can re-execute this particular write access without causing another segmentation fault. The challenge here is that the interrupted process does not share its register state with the signal handler. Therefore, the signal handling routine cannot directly read or modify the PKRU register of the interrupted process. We address this issue by locating the PKRU register saved in memory before the context switch. Before entering the signal handler, the OS saves the register state of the interrupted process in memory and recovers the registers after the signal handler returns. We therefore directly modify the PKRU value located in the saved register state so that the PKRU value modification within the signal handler is properly updated when the register state is recovered. With the updated PKRU register, program execution then continues with the write instruction that now successfully writes to the protected region. After execution finishes we check if the PKRU register was modified from the initially loaded value. If so, we know that this write access touches the protected region and that this access should be permitted. We then record this code location. Lastly, we set the PKRU register back to read-only, such that future write access to the protected region will trigger our signal handler again. This way, we can precisely locate and record functions that require legitimate access to the protected region without altering the semantics of the scripting engine.

3) Accessor Functions: In the JavaScript engine, only a limited number of functions can directly write to an object; we call these functions *Accessor Functions*. Because of the way SpiderMonkey's code base is structured, any other functions should invoke one of these accessor functions to modify a JS object—the same is true for any other code bases that respect the abstraction principle of object-oriented programming and non-OOP code bases that use abstraction layers to access specific types of data. Our dynamic analysis therefore only needs to find these accessor functions and does not require entire code coverage of the engine. As long as we ensure that each of the object types (of which there are 29 in SpiderMonkey) is covered by one of the test cases, accessor functions will be fully exercised (see Section V-B). Consequently, any other characteristics of workloads will not affect the coverage of accessor functions.

Our dynamic analysis naturally captures 300 accessor functions out of around $100,000$ functions in SpiderMonkey. We categorize these accessor functions into four groups based on their behaviors: *Member Accessors*, *Payload Accessors*, *Initialization Accessors*, and *GC Accessors*.

Member Accessors are member functions of a JavaScript object class which write to private variables. *Payload Accessors* are special member functions to update the actual payload of a JavaScript object. Every JavaScript object class implements its payload accessor which either directly stores the payload or its reference. *Initialization Accessors* are functions that initialize JavaScript objects. Most initialization functions are member functions or constructors of a JavaScript object class, but there are few cases where an independent function initializes JavaScript objects, directly writing to public variables. Most of them are for efficiently initializing string objects. Lastly, *GC Accessors* update various allocation information for garbage collection. Apart from the JavaScript objects themselves, garbage collection also makes heavy use of *object metadata*, and hence, requires memory protection as well. We therefore automatically instrument the JavaScript engine to lock down metadata access by default, and only grant legitimate write accesses to such object metadata where appropriate. Since the behavior of garbage collection could be different in our profiling environment, we conservatively find and instrument all functions in the garbage collection scope that have at least one memory write.

D. Feedback-Driven Object Protection

Our fault-based dynamic analysis framework can be used to incorporate a feedback loop from alpha testers. This feedback loop can supplement the coverage of our dynamic analysis based on a predefined set of test cases. To this end, we enforce non-writable permission for JavaScript objects with appropriate permission changes for known, legitimate write accesses such that a new write access to an object will trigger a segmentation fault. Our signal handler will catch the fault and record the function information in the same way described in Section IV-C2. This recorded information will be fed into the continuous integration system such that the next alpha release will grant the object write accesses discovered in the previous cycle. In beta and stable releases (after finishing alpha testing), the fault handler will be disabled and any unknown object write access will be considered as a potential vulnerability or malicious behavior; our protected JavaScript engine will, therefore, immediately terminate the program execution for such an unknown access.

E. Optimization

With the MPK support, one can change access permission for a protection domain by simply updating the PKRU register. Writing to a PKRU register can take around 20 cycles or more [47], [63]. While this is much more efficient than calling the `mprotect` system call, updating a PKRU register can still incur high performance overhead if the permission needs to change frequently. In our design, we grant write permission for a protection domain only within accessor functions which have legitimate write accesses to JavaScript objects. These accessor functions are frequently invoked to update an object value and to maintain inlined object metadata required for garbage collection and optimizations. The performance impact of the PKRU register update can be amplified especially when the accessor functions are called within a small yet frequently executed code region, such as a small loop or a constructor/destructor function. We therefore optimize the number of run-time permission changes by hoisting the PKRU update instructions out of such a small code region. To do so, we first find the functions that are potentially involved in frequent permission changes and hoist the permission changes to parent functions in the call graph. Note that we implement our hoisting optimization only for primitive data objects so that the security guarantee for sensitive data objects is not diminished. Our proposed optimizations significantly reduce the number of redundant protection changes, and thereby minimize the performance impact of our protected JavaScript engine (see Section V-C).

1) Code Example: We show a code example that can highly benefit from our optimization. In Listing 4, a function `init` initializes static strings in SpiderMonkey. This function consists of three loops and each of them has two function calls. The called functions are used to create and initialize string objects, which means they have to call some of the accessor functions to update an object. Since these functions are executed within the loops, there will be many permission changes, leading to high performance overhead. We can reduce this overhead by hoisting the write permission changes out of the loops.

```
bool StaticStrings::init(JSContext* cx) {
  ...
  AutoAtomsCompartment ac(cx, lock);
  ...
  saved_pkru = set_pkru(W,key)
  for(uint32_t I= 0 ; I < UNIT_STATIC_LIMIT; I++){
    JSFlatString *s = NewInlineString(...);
    ...
    unitStaticTabls[I] = s->
         morphAtomizedStringIntoPermanentAtom(hash);
  }
  for(uint32_t I= 0 ; I < pow(NUM_SMALL_CHARS,2); I++){
    JSFlatString *s = NewInlineString(...);
    ...
    length2StaticTable[I] = s->
         morphAtomizedStringIntoPermanentAtom(hash);
  }
  for(uint32_t I= 0 ; I < UNIT_STATIC_LIMIT; I++){
    JSFlatString *s = NewInlineString(...);
    ...
    initStaticTable[I] = s->
         morphAtomizedStringIntoPermanentAtom(hash);
  }
  recover_pkru(saved_pkru)
  ...
}
```

Listing 4: Example: Redundant calls

$$Score = \begin{cases} 1 & \text{if accessor function,} \\ \frac{\sum \text{score of called function}}{\text{number of function calls}} & \text{otherwise.} \end{cases} \quad (1)$$

2) Selecting Hosting Targets: We introduce a heuristic that determines where to hoist PKRU register updates. Consider a call graph where the root represents the main function and at the ends of the graph are accessor functions. Intuitively, if we insert the PKRU update instructions at the accessor functions, the attack window will open only for this small code region, but executing these extra instructions at this small code region is relatively costly. If we, on the other hand, put the PKRU updates at the root of the graph, the performance impact will be almost diminished, but it will turn most of the code into the attackable window.

Our heuristic therefore aims to find functions that, when we put a PKRU update instructions, have less performance impact while opening only a limited amount of attack window. To implement the heuristic, we first extract the global call graph of SpiderMonkey by means of LLVM's call graph analysis. Then, we score each function based on the probability that the function can eventually reach any of the accessor functions. (see Equation 1).

We demonstrate how we score each function in Figure 8, where each node represents a function. We assign every *accessor* function with score 1, the highest score in our metric. Functions without a direct write access to the protected region are assigned the average score of their child nodes, i.e., the callee functions. We use the Bellman-Ford algorithm to traverse the call graph and calculate the scores of each function based on our metric. In the example shown in Figure 8, functions D, F, and G are *accessor* functions and thus their scores are set to 1. The scores of functions E, and H become 0, on the other hand,

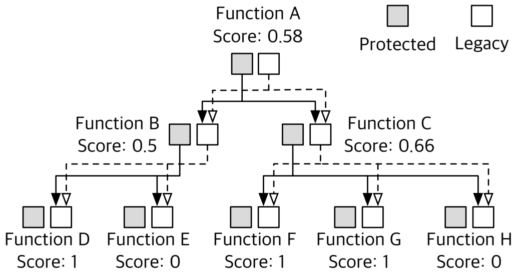


Fig. 8: Example call graph and scores for each node. The example is based on our heuristic to determine nodes to insert permission changes.

because they are neither *accessor* functions nor do they have a child node. The scores for the other functions such as A, B, and C are calculated by our metric. Based on the calculated scores, we select functions over a certain threshold to insert protection changes. The threshold is a tunable parameter that adjusts the trade-off between security and performance. In our experiment, we determine the threshold as 0.15 which incurs low performance overhead while less than 1% of the functions are additionally open for write accesses.

3) Permission Change Insertion: After selecting the functions to insert protection changes, we identify locations to which we will insert write permission changes. We could simply insert permission changes around all basic blocks that require the write permission. However, doing so may lead to frequent permission changes if multiple basic blocks require the write permission. Instead, we insert a permission change at the basic block that dominates all the legitimate write accesses. To this end, we perform the dominator analysis inside the target function. First, we find all basic blocks that can possibly visit accessor functions. We then find the nearest common dominator (NCD) of the basic blocks and insert `set_pkey` at the NCD. This will grant the write permission (to primitive objects) for all the basic blocks reachable from the dominator until the control flow reaches `recover_pkru` to strip the write permission. We insert `recover_pkru` into dominance frontiers of the NCD to prevent any of the basic blocks that are not dominated by the NCD from acquiring the write permission. In this way, we allow the write permission only for the limited number of basic blocks, without introducing excessive permission changes within a function.

4) Removing Redundant Calls: Accessor functions are by default instrumented with the write permission changes. After hoisting these permission changes to different functions, we need to remove redundant permission changes in the accessor functions. Removing such redundant permission changes from *accessor* functions is challenging: a particular *accessor* functions can also be invoked by any other functions to which, based on the scores, no protection changes have been added. We address this by maintaining two versions of a function: one with the protection changes (*protected*) and one without the instrumentation (*legacy*). We instrumented *protected* functions so that they always call the *legacy* versions of their callees to avoid redundant permission changes. Callsites in *legacy* functions are also instrumented so they call *legacy* versions of their callees by default, while the protected functions are called only at the selected call sites. In Figure 8, for example, the protected version of function A calls the legacy versions of functions B and C which also call the legacy versions of their callees.

V. Evaluation

In this section we evaluate the security and performance of NoJITsu in detail.

A. Security

The main goal of NoJITsu is to allow fine-grained memory permission management throughout the JavaScript engine at run time to protect against a wide range of memory-corruption-based exploits, such as code-injection, code-reuse, and data-only attacks. One of our key techniques to achieve this goal is to rigorously reduce the memory-access permissions of the engine's components to the bare minimum. As illustrated in Table III, the default access permissions are locked down significantly within NoJITsu for each of the components we identified in Section II-B. However, to retain compatibility and interoperability of these components within SpiderMonkey's legacy code base, we automatically instrument the respective code locations to allow non-default access permissions in a fine-grained manner temporarily. In the following, we evaluate the temporal granularity of our instrumentation. Furthermore, we verify the quality and coverage of our dynamic analysis that drives our instrumentation. We then subject our NoJITsu prototype to a number of real-world exploits, analyzing the effectiveness of our achieved protection in detail.

1) Approaching Minimal Access Requirements: We ran SpiderMonkey's built-in test suite containing more than 6,000 test scripts to drive our dynamic analysis. After we identified all code locations requiring access to sensitive JavaScript objects, we added the instrumentation code to enable access permissions where necessary. Our code transformations are similar to the ones described in Listing 2. We insert `set_pkey` and `recover_pkey` calls on a per-function basis. Thus, once the instrumentation code grants a function write access to a particular type of object, the function retains this access until it returns.

We made this design choice for two reasons. First, many of those functions issue multiple write operations to the respective objects. Therefore, changing protection in between those operations would often result in redundant permission changes. Second, the size of native functions operating on data objects is comparatively small, and hence, the instruction window within which access is enabled unnecessarily is also small.

To gauge the extent to which our defense limits the attacker's capability to corrupt JavaScript objects, we analyzed all functions that require write access to primitive and/or sensitive JavaScript objects. We also considered the types of the write accesses. Table I shows the results of our analysis. The single write row refers to functions in which the sensitive accesses are limited to regular `MOV` instructions that access a single memory location. The block write row refers to functions that can overwrite multiple memory locations using

TABLE I: Percentage of the functions that need write permissions

	Primitive obj	Sensitive obj	Both obj	Total
Single write	0.09%	0.16%	0.05%	0.29%
Block write	0.04%	0.02%	0.01%	0.07%
Total	0.13%	0.18%	0.06%	0.36%

TABLE II: Percentage of the write instructions executed in the write window of primitive objects, sensitive objects, or both

	Primitive obj	Sensitive obj	Both obj	Total
Single write	11.11%	3.29%	1.26%	15.66%
Block write	0.68%	0.19%	0.13%	1.00%
Total	11.79%	3.48%	1.39%	16.66%
Single write (Opt.)	13.08%	2.86%	2.45%	18.39%
Block write (Opt.)	0.86%	1.72%	0.43%	3.01%
Total (Opt.)	13.94%	4.58%	2.88%	21.40%

TABLE III: Default memory access permission at run time

	Permissions	
Data	SpiderMonkey 60.0.0	with NoJITsu
Bytecode	RW	R
Object tables	RW	R
JS Objects	RW	R
JIT IR	RW	R
JIT code	RX	X
JIT data	RX	R

`memcpy`-like instructions such as `REP MOVSB`. We deem these block writing functions more dangerous than single writing instructions as they are more susceptible to overflow attacks. Overall, the results are encouraging. Only a small fraction of all functions (0.36%) contain write operations targeting primitive and/or sensitive JavaScript objects.

Since our instrumentation operates at function granularity, it can sometimes leave sensitive JavaScript objects exposed to instructions that would not access these objects in memory-safe executions of the JavaScript engine. We extended our dynamic analysis and set up an experiment to measure how many extra instructions unnecessarily obtain write access to JavaScript objects. Concretely, we measured the total dynamic write instruction count while running the test suite and looked at the fraction of the write instructions that were unnecessarily in the write window. The upper half of Table II shows the results of the analysis. Here, we can see that 11.79% of the write instructions were executed while write access to the primitive JavaScript object domain was enabled, whereas 3.48% executed while access to the sensitive object domain was enabled. An additional 1.39% executed while both domains were accessible. In total, 16.66% of the executed write instructions had access to one or both domains. While this represents a large fraction of the execution, only 1% of the instructions in the write window were block writing instructions. We can, therefore, conclude that our defense substantially reduces the number of instructions that can feasibly corrupt sensitive JavaScript objects.

We hoisted some of the permission changes as part of our optimization (see Section IV-E). This optimization reduced the average performance overhead from 5% to 2%, as discussed in Section V-C. However, this performance gain may come at the cost of reduced security, allowing additional write instructions executed in the write window of sensitive JavaScript objects. To analyze the security trade-off of our hosting optimization, we measured the fraction of the write instructions that were unnecessarily executed in the write window when the optimization is enabled. The results are shown in the bottom half of Table II. Note that a small behavioral change like our hosting optimization can affect the timing when the next tier of execution (i.e., JIT code execution) is triggered in the JavaScript engine. This may introduce noise when we directly compare the dynamic instruction counts between optimized and non-optimized versions of the executions. For example, the percentage of single write instructions in the sensitive object write window slightly decreased after the optimization as a result of the noise. However, overall, the optimization led to a mild increase in the write instructions executed in the write window. After the optimization, the number increased from 0.19% to 1.72% while *only* sensitive objects were accessible and increased from 0.13% to 0.43% while both objects were accessible. This result suggests that the hosting optimization provides a reasonable trade-off between security and performance. The developer can decide the degree of hosting optimization according to the performance and security requirements of the system.

2) Code-Injection Attacks: While code-injection attacks are already mitigated to some extent by the existing deployed defense mechanisms [18], several advanced attacks aim at bypassing them, e.g., by injecting constants and exploiting unaligned instruction fetches [7], [13], [40]. These JIT spraying attacks proved challenging to mitigate in practice, since the performance overhead of constant blinding grows as the protected constants get smaller in size [19]. As a result, the current version of SpiderMonkey does not deploy constant blinding in the interpreter or the Baseline JIT compiler [59]. In NoJITsu we tackle this problem as part of our design policy to enable execute-only memory for JIT code. Since JIT code will be mapped non-readable in our prototype, we clearly separate readable data such as constants from code (see Section III-A1). This means that injected constants will no longer be mapped as executable at run time within NoJITsu.

3) Code-Reuse Attacks: To verify NoJITsu's ability to stop code-reuse attacks, we re-implemented a fully working JIT-ROP exploit based on CVE-2019-11707 which is already present in the SpiderMonkey 60.0. We achieved arbitrary read-write capability based on the CVE and launched our JIT-ROP attack. Our JIT-ROP attack exploits gadgets which the attacker dynamically inserts into the JIT code region, e.g., by forcing JIT compilation of maliciously inserted ad scripts. We verified that our JIT-ROP attack works reliably against the uninstrumented version of SpiderMonkey. We then ran the JIT-ROP exploit against NoJITsu and found that it was successfully stopped. The reason is that the generated code pages are no longer mapped as readable (eXecute-only), and hence, the attacker is not able to locate and disassemble potential gadgets at run time.

4) Bytecode Interpreter Attacks: As described in detail in Section II we developed and successfully tested a new attack against SpiderMonkey as part of our work. Since our attack corrupts data objects that are handled by the interpreter component, none of the previously proposed defenses were able to stop our attack. One of the main motivating goals behind NoJITsu is to resolve this situation. In our design, we carefully analyzed each major component within the JIT engine to identify and enforce the minimally required set of access permissions. In our attack setting this means that the attacker will no longer be able to write to the function object using the type confusion vulnerability (CVE-2019-11707), since NoJITsu separates sensitive objects and primitive objects into different protection domains. We verified and tested that NoJITsu indeed successfully prevents our new attack on SpiderMonkey. It is noteworthy that we not only protect the interpreter component. Indeed, each of the major relevant data sections such as the memory areas for Bytecode, Data Objects, Data Tables, and JIT Compiler Data are also protected using separate MPK keys in our scheme.

B. Coverage of Dynamic Object-Flow Analysis

As discussed in Section IV-C2, direct writes to JS objects are handled by a limited set of functions, which we call *accessor functions*. Our dynamic analysis, therefore, only needs to find these accessor functions and does not require entire code coverage of the engine. The accessor functions will be fully exercised as long as each of the object types (of which there are 29 in SpiderMonkey) is covered by one of our test cases.

To evaluate the soundness of our approach, we first ran our dynamic analysis with a subset of the test suite and checked whether the protected JavaScript engine based on the dynamic analysis tolerates the bigger test cases. To this end, we ran our dynamic analysis with the JIT test suite that contains 6,246 tests which is a subset of the full JavaScript test suite. Based on this analysis result we applied our object protection to the JavaScript engine. We then tested the protected version of JavaScript engine against the entire JavaScript test suite — which consists of 30,605 tests independent from the test suite that we used for the dynamic analysis. Then we checked if the new tests triggered any memory protection faults. A fault would indicate that an instruction that was not covered by our analysis wrote to a JavaScript object. We verified that our JavaScript engine instrumented with the subset of the test suite successfully passed the rest of the entire test suite without triggering any faults. This confirms that our dynamic analysis is able to cover all possible accessor functions with only the subset of test cases, and the resulting protection is robust enough to tolerate much bigger test cases.

C. Performance

We evaluated the performance of our defense on a Intel Xeon silver 4112 machine equipped with 2.60GHz CPU and 32GB memory. We ran benchmarks under Ubuntu 18.04.1 LTS whose kernel version is 4.15.0-47-generic. We used LongSpider [5] for our evaluation. LongSpider is a longer version of sunspider benchmarks. The reason for using LongSpider is that sunspider benchmarks are too microscopic. Most of the sunspider benchmarks are less than 10ms, which doesn't catch the performance overhead of our recurring changes of the protection. However, most of the LongSpider benchmarks are longer than 100ms so they are more suitable for our performance evaluation. Figure 9 shows the evaluation result. X axis is benchmarks and y axis is the performance overhead compared to the baseline. There are five different bars. The bar named `JIT_PROT` is for the overhead of JIT protection. `INTER_PROT` is for the interpreter protection. `ALL_PROT` is the combined performance overhead for both JIT and interpreter protections and `OPT` stands for the optimization. On average, our NoJITsu has less than 5% overhead and with optimization it becomes less than 2%. The overhead for `JIT` is marginal, which is 0.6% on average. Some benchmarks have better performance than the baseline because all expensive `mprotect` operations are replaced by cheap MPK register writes. However, benchmarks such as `bitops-bits-in-byte`, `date-format-tofte`, and `string-tagcloud` have higher overhead compared to the others. We found that the root cause of the overhead are cache misses. We need to position JIT code and data in different pages for code and data separation, which loses cache locality. For instance, if the size of code is too small, both the data and the code using that data can fit into the same cache line. Code and data separation introduces a large(r) offset between code and data regions, not allowing for both to fit into the same cache line. For the interpreter protection, there is almost no overhead from bytecode and table protection because the overhead comes from the generation of those data, which is marginal compared to whole execution. Most of the observed overhead is a result of the object protection, which keep changing the protection during the execution. In Figure 9, `date-format-xparb` and `string-base64` cause a significant overhead for the object protection because they involve frequent write operations to string objects. However, our optimization drastically reduces this overhead. As discussed earlier, we achieve this by hoisting certain instructions within our instrumentation.

VI. Discussion

A. Applicability to Other Systems

While we instantiated our attacks and defenses in SpiderMonkey, the underlying approaches are generally applicable to other script engines that employ bytecode interpreters. We analyzed two mainstream JavaScript engines, V8 [30] and JavaScriptCore [4], to clarify how our approaches could be applied to these JavaScript engines. The engines have a number of reported memory corruption vulnerabilities which may allow attackers to read and write arbitrary memory locations [1], [2].

a) Attack: Our interpreter attack leverages the facts that most of the key data structures of the interpreter remain writable throughout the execution and that the interpreter has a special way of calling native functions – in which contents of certain JS objects determine the target address and arguments of a function call. Specifically, our attack overwrites the two data structures in SpiderMonkey: (i) a function object which contains the address of the function to invoke, and (ii) the context object which is always passed as the first argument for native function calls (see Section II-C). We found that in V8 and JavaScriptCore any types of JS objects remain writable,

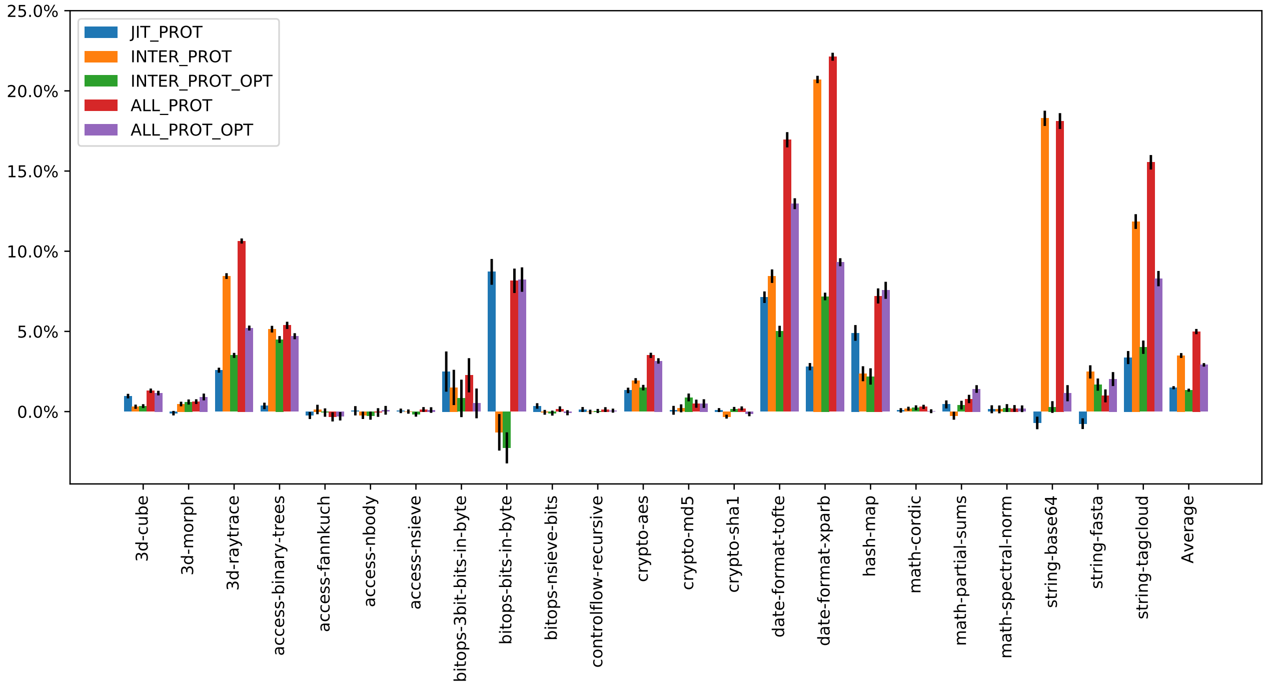


Fig. 9: NoJITSU performance

and both of these engines have internal calling conventions for native JS functions similar to SpiderMonkey. Therefore, our presented interpreter attack would be possible for these engines. For example, JavaScriptCore makes a native function call by reading a target address from a function object and passing a global object pointer as the first argument. Therefore, deliberately overwriting the function object and the global object in the call frame would allow the attacker to invoke his desired function.

b) Defense: The bytecode interpreters in V8 and JavaScriptCore have different implementations than SpiderMonkey – SpiderMonkey has a switch-based interpreter while V8 and JavaScriptCore implement threaded interpreters; SpiderMonkey's interpreter is a stack-based machine, whereas the other two are register-based machines. Despite such differences, the core mechanism of bytecode interpretation remains the same, that is, each bytecode instruction has a sequence of code that handles its desirable operation and, when necessary, the instruction can access JavaScript objects via object tables. Consequently, V8 and JavaScriptCore have the same components that NoJITSU protects in our SpiderMonkey prototype, i.e., bytecode, JavaScript objects, object tables, JIT IR cache, and JIT code cache. Like SpiderMonkey, V8 and JavaScriptCore have different types of JavaScript objects. We identified several primitive objects as well as crucial objects such as the function object which stores the address of the corresponding function. Therefore, NoJITSU's protection mechanism could be directly applied to these engines that use the similar data structures, by assigning minimum access permissions for individual data structures and temporarily grant extra permissions only when that is necessary.

Bytecode itself normally does not change after the code is generated and thus in NoJITSU the memory region containing bytecode remains read-only after initialization. Previous versions of JavaScriptCore had an optimization called bytecode inline caching which directly modifies a bytecode stream. Such an optimization could have induced more performance overhead to our defense since modifying the bytecode would require additional permission changes. However, this optimization is not used anymore for memory reason and thus we do not expect extra overhead applying NoJITSU to this engine [65].

B. Alternatives to Intel MPK

While our prototype uses Intel MPK, the design of NoJITSU is not heavily tied to its specific hardware implementation and using other hardware-based memory protection schemes that allow restriction of memory access permissions beyond traditional virtual memory protection, such as ARM Memory Domains [6], should be feasible in principle. This relation between Intel MPK and ARM Memory Domains was also noted by prior work on Software-Fault Isolation and Compartmentalization [20], [47], [63]. Similar to Intel MPK, ARM Memory Domains support 16 different protection domains. However, while Intel MPK allows domain switches in user space, ARM Memory Domains require a system call roundtrip. Although NoJITSU uses Intel MPK's ability to efficiently implement execute-only permissions, there are no conceptual limitations that would prevent leveraging non-MPK implementations [8], [15], [23] in support of that feature.

VII. Related Work

JIT compilers have been under constant siege by adversaries ever since they were introduced in mainstream web browsers. The earliest JIT compilers left the code cache writable and executable at all times. This trivially enabled code-injection attacks [18], [55]. Early attempts to address this issue included monitors that detected system calls originating from writable code regions [26]. However, as JIT compilers began to enforce strict W⊕X policies [18], either by double mapping the JIT code cache or by toggling the writable and executable permissions before and after code emission, JIT code injection became a less interesting attack vector.

As an alternative, Blazakis proposed JIT spraying, an attack technique that injects code indirectly by running a script that contains user-specified constants (e.g., as part of a long XOR computation) [7], [13], [40]. Since these constants appear as instruction operands in the JIT code cache, they can be executed as if they were valid instructions. Several countermeasures thwart JIT-spraying attacks by either eliminating user-specified constants through obfuscation or constant blinding [19], randomizing the JIT code [32], or by extending control-flow integrity to JIT code [46]. NoJITsu strengthens these existing defenses by additionally separating data constants from JIT code (see Figure 7). This enables us to enforce non-executable permissions for constants.

Snow et al. proposed to attack JIT engines through code-reuse attacks [56]. Their JIT-ROP attack leveraged a memory disclosure vulnerability to recursively disassemble the code region, thereby discovering useful code gadgets on-the-fly. These gadgets can then be chained together to launch a return-oriented programming attack [54]. Defenses against JIT-ROP included execute-only memory combined with randomization [8], [9], [23], [29], destructive code reads [61], [64], and cross-checking reads performed by JIT code [28]. However, some of these defenses were quickly bypassed [39], [57], while others were not deployed due to impractical design or resource requirements. As we demonstrate in our evaluation, NoJITsu thwarts even dynamic code-reuse attacks such as JIT-ROP with a low overhead. Our design is generic and leverages automated dynamic analysis and instrumentation to scale to complex real-world code bases such as SpiderMonkey.

Song et al. showed that direct code-injection attacks on the JIT cache were still possible by leveraging JavaScript worker threads [58]. Their proposed defense moved the JIT compilation thread to a separate process, thereby preventing the code cache from ever being writable in the JIT execution process. This approach was later adopted in Microsoft's Chakra engine [42]. However, Microsoft recently announced shifting their focus and replacing Chakra with V8 as part of their Edge browser [10]. NoJITsu does not require a re-design of the JavaScript engine but separates different components inside the same process to enforce fine-grained page-based permissions.

With several code-injection and code-reuse mitigations being adopted for JIT compilers, attackers turned their eyes to data-only attacks. Theori et al. presented a data-only attack that overwrites intermediate code structures in Chakra [62], whereas Frassetto et al. presented a similar attack on SpiderMonkey [27]. The proposed defense moved the JIT compiler to an Intel SGX enclave, thus protecting its data structures from corruption attacks. All of the above defenses focus on mitigating attacks against the JIT compiler. Crucially, this means they cannot prevent our attack which corrupts the bytecode interpreter (see Section II). For certain architectures, interpreter-based attacks were known to represent a fruitful target of attacks for some time: Cama et al. [17] presented an attack on the PS Vita that corrupted the virtual call table of a JavaScript object in Webkit's interpreter environment. While their attack targets similar components to ours, these two attacks are conceptually different. The PS Vita attack is based on a well-known COOP-style attack which overwrites object's virtual function table (vtable) pointer with a pointer to a fake vtable [51], and thus this attack would be prevented by existing defenses against vtable corruption or vtable reuse attacks [14], [24], [67]. In contrast, our attack deliberately overwrites the internal data of a JavaScript object (not the vtable pointers of any objects with virtual methods) to invoke a chosen function; in fact, this manipulates how the bytecode interpreter interprets the corrupted JavaScript object. In the technical aspects, their attack targets Webkit/JavaScriptCore for the PS Vita (ARMv7) and the context of the object being modified (via `this`) is saved and restored using `setjmp`/`longjmp` to be able to safely return to the JS environment. Our exploit is for SpiderMonkey (x86) and leverages the fact that the first 32-byte can be overwritten safely without restoring them. However, there are many more components inside a modern JavaScript engine that an attacker could exploit. As our evaluation shows, NoJITsu is able to mitigate a vast number of different memory-corruption attacks against each of the JIT engine's major components offering fine-grained memory protection.

There are previous works to provide secure isolation interfaces using MPK. Libmpk [47] provides a secure software abstraction to improve security and resolve technical challenges in using MPK. ERIM [63] utilizes MPK to isolate trusted and untrusted memory regions so it can be used to implement memory isolation mechanisms, such as the safe store in Code-Pointer Integrity (CPI) [34]. These approaches are orthogonal to our approach and some of their techniques could be combined with NoJITsu to further enhance performance and security. Instead of using glibc's MPK APIs to implement our defense, using libmpk could further improve security and performance of MPK operations. Also, ERIM's technique to detect PKRU-modification patterns and to remove them via binary rewriting could be integrated into our work to further improve security.

VIII. Conclusion

JavaScript engines are essential for performance and security of modern systems software, such as web browsers. Many existing works demonstrate attacks against JavaScript engines and also propose defenses to mitigate some of these attacks. In this paper, we show that previously proposed mitigations are unfortunately not sufficient to protect JavaScript interpreters against sophisticated adversaries. First, we demonstrate a new attack that leverages the interpreter, which was previously assumed secure by design, to execute arbitrary shell commands. Our attack works in the presence of all existing defenses that we're aware of. Second, we propose a novel defense design, dubbed NoJITsu, to bring hardware-backed, fine-grained memory access protection to complex, real-world JavaScript engines. As part of our security analysis we show that this allows us to provide protection against a wide range of possible attacks, including code-injection, code-reuse, and data-only attacks. As we are able to demonstrate NoJITsu successfully thwarts real-world attacks by minimizing memory access permissions between different components towards the strictly required minimum. Our prototype leverages automated dynamic analysis to instrument and scale to complex code bases such as SpiderMonkey, offering a moderate overhead of only 5%.

Acknowledgment

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, and by the National Science Foundation under awards CNS-1619211 and CNS-1513837. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government. The authors also gratefully acknowledge a gift from Oracle Corporation.

References

[1] "CVE-2016-4622." Jul.21 2016. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-4622

[2] "CVE-2019-5755." Feb.19 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-5755

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[4] Apple, "Webkit," https://webkit.org.

[5] ——, "Longspider," https://github.com/WebKit/webkit/tree/master/PerformanceTests/LongSpider, 2015.

[6] ARM, "Arm memory domains," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html, 2015.

[7] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The devil is in the constants: Bypassing defenses in browser JIT engines," in *NDSS*, 2015.

[8] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

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

[10] J. Belfiore, "Microsoft edge: Making the web better through more open source collaboration," https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/, 2018.

[11] S. Bhatkar and R. Sekar, "Data space randomization," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[12] bi0s, "Writeup for CVE-2019-11707," https://blog.bi0s.in/2019/08/18/Pwn/Browser-Exploitation/cve-2019-11707-writeup/, 2019.

[13] D. Blazakis, "Interpreter exploitation: Pointer inference and JIT spraying," *BlackHat DC*, 2010.

[14] D. Bounov, R. G. Kici, and S. Lerner, "Protecting c++ dynamic dispatch through vtable interleaving." in *NDSS*, 2016.

[15] K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen, "Leakage-resilient layout randomization for mobile devices." in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[16] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Technical Report MSR-TR-2008-120, Microsoft Research, Tech. Rep., 2008.

[17] A. Cama, "Ps vita level 1: Webkitties," http://acez.re/ps-vita-level-1-webkitties-3/, 2014.

[18] P. Chen, Y. Fang, B. Mao, and L. Xie, "JITDefender: A defense against JIT spraying attacks," in *IFIP International Information Security Conference (SEC)*, 2011.

[19] P. Chen, R. Wu, and B. Mao, "JITSafe: A framework against just-in-time spraying attacks," *IET Information Security*, vol. 7, no. 4, pp. 283–292, 2013.

[20] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 56–71.

[21] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, "Norax: Enabling execute-only memory for cots binaries on aarch64," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[22] J. Corbet, "Intel memory protection keys," https://lwn.net/Articles/643797/, 2015.

[23] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[24] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz, "It's a TRaP: Table randomization and protection against function reuse attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[25] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Pt-rand: Practical mitigation of data-only attacks against page tables." in *NDSS*, 2017.

[26] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens, "Jitsec: Just-in-time security for code injection attacks," in *Benelux Workshop on Information and System Security (WISSEC)*, 2010.

[27] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, "JITGuard: Hardening just-in-time compilers with sgx," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[28] R. Gawlik, P. Koppe, B. Kollenda, A. Pawlowski, B. Garmany, and T. Holz, "Detile: Fine-grained information leak detection in script engines," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[29] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[30] Google, "V8," https://v8.dev.

[31] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[32] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: transparent code randomization for just-in-time compilers," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[33] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

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

[35] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[36] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[37] W. Lian, H. Shacham, and S. Savage, "Too lejit to quit: Extending jit spraying to arm." in *Symposium on Network and Distributed System Security (NDSS)*. Citeseer, 2015.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018.

[39] G. Maisuradze, M. Backes, and C. Rossow, “What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses,” in *USENIX Security Symposium*, 2016.

[40] ——, “Dachshund: digging for and securing against (non-) blinded constants in jit code,” in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[41] Microsoft, “Data execution prevention (DEP),” http://support.microsoft.com/kb/875352/EN-US, 2006.

[42] Microsoft, “Out-of-process jit support,” https://github.com/Microsoft/ChakraCore/pull/1561, 2016.

[43] D. l. Mingwei Zhang, Ravi Sahita, “eXecutable-Only-Memory-Switch (XOM-Switch),” in *Black Hat Asia Briefings (Black Hat Asia)*, 2018.

[44] Mozilla, “W xor x JIT-code enabled in firefox,” https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox, 2015.

[45] ——, “Spidermonkey,” https://ftp.mozilla.org/pub/spidermonkey/prereleases/60/pre3, 2018.

[46] B. Niu and G. Tan, “RockJIT: Securing just-in-time compilation using modular control-flow integrity,” in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[47] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, “libmpk: Software abstraction for intel memory protection keys,” in *USENIX Annual Technical Conference*, 2019.

[48] PaX Team, “Address space layout randomization (aslr),” https://pax.grsecurity.net/docs/aslr.txt, 2001.

[49] saelo, “Exploiting logic bugs in javascript jit engines,” http://phrack.org/papers/jit_exploitation.html, 2019.

[50] P. D. Schubert, B. Hermann, and E. Bodden, “Phasar: An interprocedural static analysis framework for c/c++,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019.

[51] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications,” in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[52] M. Seaborn and T. Dullien, “Exploiting the dram rowhammer bug to gain kernel privileges,” in *BlackHat USA*, 2015.

[53] SecuriTeam, “Cve-2018-12387,” https://github.com/tunz/js-vuln-db/blob/master/spidermonkey/CVE-2018-12387.md, 2018.

[54] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[55] A. Sintsov, “Writing JIT-spray shellcode for fun and profit,” 2010.

[56] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[57] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, “Return to the zombie gadgets: Undermining destructive code reads via code inference attacks,” in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[58] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, “Exploiting and protecting dynamic code generation,” in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[59] A. Souchet, “Introduction to spidermonkey exploitation,” https://doar-e.github.io/blog/2018/11/19/introduction-to-spidermonkey-exploitation/#force-the-jit-of-arbitrary-gadgets-bring-your-own-gadgets, 2018.

[60] Y. Sui and J. Xue, “Svf: interprocedural static value-flow analysis in llvm,” in *International Conference on Compiler Construction (CC)*. ACM, 2016.

[61] A. Tang, S. Sethumadhavan, and S. Stolfo, “Heisenbyte: Thwarting memory disclosure attacks using destructive code reads,” in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[62] Theori, “Chakra JIT CFG bypass,” http://theori.io/research/chakra-jit-cfg-bypass, 2016.

[63] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, P. Druschel, and D. Garg, “Erim: Secure, efficient in-process isolation with memory protection keys,” in *USENIX Security Symposium*, 2019.

[64] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, “No-execute-after-read: Preventing code disclosure in commodity software,” in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[65] T. Zagallo, “A new bytecode format for javascriptcore,” https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/, 2019.

[66] P. Zero, “Virtually unlimited memory: Escaping the chrome sandbox,” https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.htmll, 2019.

[67] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, “Vtrust: Regaining trust on virtual calls.” in *NDSS*, 2016.