Title
Comprehensive Protection for Dynamically-typed Languages: Avoiding the Pitfalls of Language-level Sandboxing

Permalink
https://escholarship.org/uc/item/2g9542cn

Author
Park, Taemin

Publication Date
2020

Peer reviewed|Thesis/dissertation
Comprehensive Protection for Dynamically-typed Languages: Avoiding the Pitfalls of Language-level Sandboxing

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Taemin Park

Dissertation Committee:
Professor Michael Franz, Chair
Professor Nikil Dutt
Professor Alexandru Nicolau

2020
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>VITA</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>x</td>
</tr>
</tbody>
</table>

## 1 Introduction
1.1 Chapter 3 .......................... 4
1.2 Chapter 4 .......................... 4

## 2 Background
2.1 Memory corruption vulnerability ........................................... 6
2.2 Code-injection attacks & their mitigation ............................. 7
2.3 Code-reuse attacks & their mitigation .................................. 7
2.4 JIT Code Injection ....................................................... 8
2.5 JIT Spraying ............................................................ 9
2.6 JIT ROP ................................................................. 9

## 3 Bytecode Corruption Attacks and Defenses
3.1 Motivation .............................................................. 10
3.2 Background ............................................................. 12
  3.2.1 Bytecode Storage .................................................. 13
  3.2.2 Data Encapsulation ............................................... 13
  3.2.3 Data Access ........................................................ 14
  3.2.4 Function Calls .................................................... 16
  3.2.5 Dangerous Interpreter Features .................................. 17
  3.2.6 Running Untrusted Scripts ........................................ 19
3.3 Threat Model And Assumptions ........................................... 20
  3.3.1 Realism ............................................................. 21
3.4 Attacking Bytecode ...................................................... 22
  3.4.1 Attack Overview .................................................... 22
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Loading a global variable through a hash map-like table.</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Loading a value through the constant table.</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Overview of our heap layout inference step.</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Blueprint buffer construction.</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>The payload created based on Strategy 1.</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>The payload created based on Strategy 4.</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Overview of our defense.</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>Run-time overhead in the Python Performance Benchmark Suite.</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Run-time overhead in the Computer Language Benchmarks Game for Lua.</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>High-level overview of our model</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Attack procedure</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Disclosing the locations of the victim JSFunction object and the JSContext object.</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Legacy design</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>The design of NoJITsu</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>Design of our script engine protection NoJITsu</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>JIT code and data separation</td>
<td>58</td>
</tr>
<tr>
<td>4.8</td>
<td>Example call graph and scores for each node</td>
<td>67</td>
</tr>
<tr>
<td>4.9</td>
<td>NoJITsu performance</td>
<td>77</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Supporting data structures in Python and Lua</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Percentage of the functions that need write permissions</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Percentage of the write instructions executed in the write window of primitive objects, sensitive objects, or both</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Default memory access permission at run time</td>
<td>73</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. Michael Franz for his support during my doctoral study. He gave me the opportunity to pursue a Ph. D. in Secure Systems and Software lab where I have honed my knowledge and experience that could not be obtained from any other places. Under Dr. Michael Franz’s guidance, I have been able to learn how to develop ideas and critical thinking, constructing the crucial building block of my future career.

I would like to thank our postdocs, Dr. Adrian Dabrowski, Dr. David Gens, Dr. Per Larsen, Dr. Stijn Volckaert, and Dr. Yeoul Na for sincere advise on research ideas and sharing their expertise. It was my privilege collaborating with them. I also want to express my gratitude to our lab colleagues and alumni, Anil Altinay, Alexios Voulimeneas, Prabhu Rajasekaran, Joseph Nash, Dokyung Song, Paul Kirth, Michel Dickerson, Matt Dees, Fabian Parzefall, Min-Yih Hsu, Chinmay Deshpande, Julian Lettner, Brian Belleville, and Mohaned Qunaibit. It was great time for me to spend five years with them, making lots of memories and helping to each other.

I want to give special thanks to my dissertation committee, Dr. Alexandru Nicolau, and Dr. Nikil Dutt for sparing their precious time and kind help.

Finally, I would love to express the biggest thanks to my parents and sister. They are alway at my side and encourage me in deepest trust. Their wholehearted support has provided me with confidence in confronting any challenges with little fear.

Portions of this dissertation have been previously published in conference proceedings. To my coauthors on these projects, thank you for your contributions on these publications.


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.
VITA

Taemin Park

EDUCATION

Doctor of Philosophy in Computer Science 2020
University of California, Irvine
Irvine, California

Master of Science in Computer Science 2012
Seoul National University
Seoul, South Korea

Bachelor of Science in Computer Engineering 2010
Hanyang University
Seoul, South Korea

RESEARCH EXPERIENCE

Graduate Research Assistant 2015–2020
University of California, Irvine
Irvine, California

Research Intern Summer 2019
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

Research Engineer 2012–2015
Korea Telecom Institute of Convergence Technology
Seoul, South Korea

Graduate Research Assistant 2010–2012
Seoul National University
Seoul, South Korea
TEACHING EXPERIENCE

Teaching Assistant  
CS 143A: Principles of Operating Systems  
University of California, Irvine  
Spring 2020

Teaching Assistant  
CS 201P: Computer Security  
University of California, Irvine  
Winter 2020

Teaching Assistant  
CS 132: Computer Networks  
University of California, Irvine  
Spring 2017

Teaching Assistant  
ICS 11: The Internet and Public Policy  
University of California, Irvine  
Winter 2017

Teaching Assistant  
Advanced Computer Networks  
Seoul National University  
Spring 2010  
Seoul, South Korea
REFEREED JOURNAL PUBLICATIONS

CLACK: Cross-Layer ACK-Aided Opportunistic Transmission in Wireless Networks
Wireless Personal Communications

Multi-hop transmission and routing with hierarchical modulation
EURASIP Journal on Wireless Communications and Networking

Social-community based DTN Routing
Journal of KIIE : Information Networking

Naval Tactical Data Network Scheduling Method for the Supersonic Anti Surface Missile Defense
Korea Computer Congress

Survey on human behavior and social interaction in Massive Multiplayer Online Role Playing Game
Korea Computer Congress

REFEREED CONFERENCE PUBLICATIONS

NoJITsu: Locking Down JavaScript Engines
26th Annual Network and Distributed System Security Symposium (NDSS)

PartiSan: Fast and Flexible Sanitization via Run-time Partitioning
21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)

Bytecode Corruption Attacks Are Real-And How To Defend Against Them
15th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)
Dynamically-typed languages have improved programming experience in software development, leading to widespread adoption in the modern software ecosystem. As dynamically-typed languages continue to evolve, their implementations inevitably become more complex and error-prone. As a result, many bugs in the language implementations are found every year, and attackers try to exploit them for code-injection or code-reuse attacks. Prior work has attempted to defend against these attacks by using technologies such as data execution prevention (DEP), software diversity, control-flow integrity (CFI), etc. However, interactive scripting environments provide attackers with a unique attack surface, capable of bypassing existing defenses.

In this dissertation, we explore a new attack vector that was thought to be non-exploitable: the bytecode interpreter attack. We propose four attack strategies to compromise the bytecode interpreter itself. We show that our attacks successfully lead to arbitrary code execution in three popular dynamically-typed languages (Python, Lua, and JavaScript). To address this new attack vector, we propose a new comprehensive mitigation for dynamically-typed languages: NoJitsu. NoJITsu protects complex, real-world scripting engines from not only the bytecode corruption but also prior code-injection and reuse attacks. The key idea be-
hind our comprehensive defense is to enable fine-grained memory access control for individual memory regions based on their roles throughout the script engine’s life-cycle. We combine automated analysis, instrumentation, compartmentalization, and Intel’s Memory-Protection Keys (MPK) to secure script engines against existing and newly synthesized attacks. Further, we thoroughly test our implementation using several real-world scenarios as well as standard benchmarks. We show that NoJITsu successfully thwarts code-reuse, code-injection, and bytecode attacks against any part of the scripting engine while offering a modest run-time overhead.
Chapter 1

Introduction

Dynamically-typed languages are now prevalent in modern computing systems. JavaScript is embedded in major web browsers such as Chrome, Safari, and Firefox, fostering rich interaction between users and web pages. Python is the third most popular language, and because of its immense library support, it is used in a wide range of areas from web applications to data analysis. Some dynamically-typed languages are designed for specific use-cases. For instance, Lua offers high extensibility and fast execution, which makes it one of the most popular languages in game development [61].

As dynamically-typed languages gain popularity, they draw the attention of sophisticated attackers. Since dynamically-typed languages are type and memory-safe, a mistake in a script is unlikely to generate a vulnerability. However, the script must be interpreted by a script engine that is written in an unsafe language such as C/C++. Therefore, an attacker can target a vulnerability in the script engine instead.

As script engines evolve, they are becoming larger and inevitably contain more bugs. Every year, hundreds of vulnerabilities are discovered in various script engines, and they are low hanging fruits for attackers. Therefore, protecting dynamically-typed languages becomes
increasingly important over time.

There have been extensive efforts to combat vulnerability exploitation for more than two decades. A primitive type of attack, such as a stack-smashing attack [4], can be thwarted through the adoption of data execution prevention (DEP) [65]. As a result, attackers have switched to code-reuse attacks, such as ret-to-libc [101] and return-oriented programming (ROP) [85] that allow the attacker to achieve arbitrary computation without needing to inject code. For defeating code-reuse attacks, many techniques have been proposed and are largely divided into two lines of research: software diversity [55, 52, 51, 31] and control-flow integrity (CFI) [3]. The software diversity techniques randomize code or data in the memory to hide the information that attackers can exploit. On the other hand, CFI ensures only legitimate control-flows in a program, disallowing a program from being diverted to an arbitrary location. In response to those defense techniques, attackers developed even more intricate technologies, and the cat and mouse game continues.

The emergence of dynamically-typed languages has introduced new attack vectors and made previous defenses ineffective. One of the most significant innovations in dynamically-typed languages is just-in-time (JIT) compilation that significantly improves the execution speed but simultaneously reintroduces the possibility for code injection. For example, JavaScript engines speculate runtime conditions, and and when a particular function or sequence of operations is repeatedly executed [23, 39], it is compiled into machine code directly executed by CPU. Generating machine code at runtime allows attackers to inject their malicious code, which breaks the security guarantee of DEP. As a countermeasure, some JavaScript engines adopted a DEP-like implementation where JIT’ed code is always executable, but only writable when it is emitted to memory [69].

Since JIT’ed code is writable for a short period, it significantly raises the bar for an attacker to inject their code. However, the scripting environment gives attackers the ability to encode malicious code in constants [15]. When those constants are sprayed over the heap, the
attacker can trigger a vulnerability to make the program counter jump to the middle of the constants where an attacker’s malicious codes reside. This attack can be mitigated by constant blinding [79] which hides constants by encoding them with a secret random variable. Since they have to be decoded whenever they are used, additional overhead is inevitable.

An attacker also can take advantage of the interaction between his script and runtime environment for information disclosure attacks. For example, JIT-ROP attacks [88] collect ROP gadgets and construct a ROP attack at runtime, circumventing load time randomization defenses. Also, more advanced attacks even exploit the multi-threading capability of dynamically-typed languages. Song et al. [90] proposed a new technique to spawn a separate thread and leverages a race condition to overwrite JIT’ed code when it becomes writable in the main thread. Frassetto et al. [38] also uses the same technique to corrupt JIT IR, which is an intermediate representation of the JIT’ed code. To defend against such attacks, they suggested isolation of JIT components either by a separate processor or a trusted execution environment, respectively.

Even though a variety of attack and defense mechanisms have been developed regarding dynamically-typed languages so far, their focus is solely confined to JavaScript and JIT components even though the basic execution unit in dynamically-typed languages is an interpreter. It is not surprising that researchers have neglected interpreters because they have a language-level sandbox, which doesn’t at first appear to provide exploitation opportunities for attackers.

In this dissertation, we explore this seldom considered area of dynamically-typed languages and invalidate the common belief that bytecode interpreters are hard to corrupt. In Chapter 3, we first demonstrate bytecode corruption attacks that change the behavior of interpreters to run arbitrary system calls or shell commands. We systematically developed four attack strategies to achieve our goal and implemented them in two popular dynamically-typed languages: Python and Lua. Further, we suggest a defense mechanism to secure
bytecode against our new attack vectors. In Chapter 4, we extend our work in Chapter 3 to JavaScript in order to generalize our bytecode attacks and make our defense more comprehensive. More specifically, we first show that our bytecode interpreter attacks are applicable in JavaScript. Our comprehensive defense protects dynamically-typed languages from not only bytecode interpreter attacks but also the existing attacks targeting JIT components by securing data in JavaScript engines. Our contributions are as follows.

1.1 Chapter 3

- We study the internals of modern bytecode interpreters, uncover several potential attack vectors, and show why bytecode corruption is challenging.

- We present an attack that enables arbitrary code execution in an interpreter by corrupting the bytecode and data caches. Our attack starts with an information disclosure step which infers the layout of the heap. Depending on the layout of the heap, we pursue one of four different attack strategies when constructing the attack payload. We implement our attack in two different languages/interpreters with different architectures: CPython and Lua, stack-based and register-based VMs, respectively.

- We propose a defense that protects the integrity of the bytecode caches and evaluate a reference implementation for both interpreters. Our evaluation shows that the suggested defense successfully prevents all four of our attack strategies.

1.2 Chapter 4

- We implement an instance of the bytecode attacks in Chapter 3 in modern JavaScript engines. Our attack works despite all existing defenses being enabled and enforced.
• We propose 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 components. We implemented our prototype in a recent version of Mozilla’s SpiderMonkey.

• We extensively evaluate the prototype of our approach 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. Our technique withstands all previously presented attacks as well as our new bytecode attacks. We further evaluate performance using standard benchmarks and practical use cases, demonstrating that it additionally offers low overhead with an average performance hit of only 5%.
Chapter 2

Background

2.1 Memory corruption vulnerability

Unsafe languages such as C and C++ have been widely used in system development due to their efficiency and flexibility. However, their strong aspects inevitably accompany manual memory management, which makes human errors unavoidable. A mistake in writing code can lead to an undefined behavior that is syntactically correct, but different from a programmer’s intension. As a result, an attacker can turn this undefined behavior into what they want to execute in the victim’s system in a process known as vulnerability exploitation. For example, the lack of bounds checking in unsafe languages allows a program to access beyond the size of a buffer. An attacker can exploit this property to read and write data out of bound where sensitive information could be positioned. Moreover, a programmer could make a mistake in dereferencing a deallocated pointer variable. This pointer variable can be used for leaking sensitive information when the associated memory area is later allocated with the sensitive data.
2.2 Code-injection attacks & their mitigation

One of the oldest memory corruption exploits is a code injection attack. A buffer overflow vulnerability allows an attacker to write beyond the boundary of a buffer in the stack by which the attacker injects his malicious code and overwrite a return address with the address of the injected code. Consequently, instead of returning to the calling function, the program jumps to the attacker’s injected code. This type of attack is easily defeated by adopting data execution prevention (DEP) [65]. DEP makes the memory stack writable not executable and thus executing injected code in the stack is no longer available.

2.3 Code-reuse attacks & their mitigation

Because of the emergence of DEP, attackers started to switch to code-reuse attacks which reuse existing code in the memory rather than injecting crafted code. One instance of code-reuse attacks is ret-to-libc [101] that tries to overwrite a return address with the address of dangerous system call wrappers in libc. A more generic and powerful approach towards code-reuse attack is return-oriented programming (ROP) [85] which is proven Turing-complete, realizing arbitrary computation. In ROP attacks, an attacker gathers code snippets ended with a return instruction. Those snippets are called ROP gadgets constituting a malicious program when they are executed in an intended sequence. When the addresses of the ROP gadgets are written to a return address in stack memory, a program will execute the ROP gadgets one by one.

Code-reuse attacks are so powerful that they have been at the forefront of the systems software research area. There are two lines of researches to defeat code-reuse attacks: software diversity [55] and control-flow integrity (CFI) [3]. Software diversity aims to hide ROP gadgets which are core ingredients of mounting ROP attacks. One of the famous and widely-used
software diversity techniques is address space layout randomization (ASLR) [76] that randomizes the base address of a program image at load time. Researchers also suggested more sophisticated schemes such as function-level randomization, instruction-level randomization, and register randomization, etc [55, 52, 31, 17] However, they are weak to information disclosure attacks [86, 92, 88]. For example, Snow et al. [88] utilizes a scripting environment to read memory layout to find enough number of ROP gadgets for meaningful attacks at runtime. The other line of research is control-flow integrity [3] which guarantees legitimate control-flow transitions, disallowing jumps to arbitrary ROP gadgets. Abadi et al. [3] first came up with this concept, and they argue that CFI based on complete control-flow graphs effectively stops ROP attacks. However, it is challenging to construct full control-flow graphs due to the imprecision of points-to analysis, and rigorous verification of legitimate jump locations causes significant overhead. To overcome those challenges, researchers have developed a number of variations of CFI by mostly relying on coarse-grained CFG [28, 74, 106, 105]. In response to various kinds of CFI, researchers have also suggested diverse techniques to bypass the coarse-grained CFI [44, 45, 22, 35] and even they proved that fine-grained CFI [96, 68] can also be bypassed [29, 21, 37].

2.4 JIT Code Injection

Early JIT-based VMs either left the JIT-compiled code writable at all times, or mapped two copies (backed by the same physical memory pages) of the JIT-compiled code into the VM’s virtual address space: one writable and one executable. In both of those cases, an attacker could simply inject code into the JIT code cache by locating it in memory and overwriting it. To prevent such attacks, all major browsers and high-performance JIT engines have now adapted Data Execution Prevention, and they generally only map one copy of the JIT code cache into virtual memory at any given time. The code cache is made writable only while
code is being generated, and the JIT engine makes the cache non-writable while executing JIT-compiled code.

2.5 JIT Spraying

The JIT spraying attacks [15] utilize the fact that JIT compilers copy constant integers unmodified into the JIT-compiled code region. An attacker can therefore embed instructions into constants used in a script to inject short code snippets into the JIT code cache. The injected code can be executed by jumping into the middle of instructions. In response, researchers proposed constant blinding [79] to mitigate the JIT spraying attack. Constant blinding basically masks constants with a random secret value, which makes attacker’s embedded code unpredictable. However, the masked value cannot be used in itself but should be decoded by instrumented codes, incurring significant overhead.

2.6 JIT ROP

Snow et al.’s JIT-ROP attack showed that code randomization for JIT-compiled code can be bypassed by disclosing a pointer to the JIT code cache, and by recursively disassembling the code starting from the instruction pointed to by the disclosed pointer [88]. This technique allows attackers to discover the locations of injected constants, which they can then jump to to execute the embedded code. Execute-No-Read [10, 32] and destructive code read defenses [94, 99] normally prevent such code-disclosure attacks, but were proven ineffective in the context of JIT VMs [89].
Chapter 3

Bytecode Corruption Attacks and Defenses

3.1 Motivation

Programs written in dynamic languages execute in a virtual machine (VM). This VM typically translates the program’s source code into bytecode, which it then executes in an interpreter. Some VMs also include a JIT compiler that can compile the source code or bytecode into machine code that can be executed directly by the CPU. The VM usually guarantees that the execution of the script is type and memory safe by lifting the burden of managing the application memory and run-time types off the programmer.

Unfortunately, most VMs are implemented in type and memory unsafe languages (specifically, C/C++) which provide direct control over memory. Consequently, memory and type safety vulnerabilities often slip into the VM itself. Malicious scripts may exploit these vulnerabilities to leak information, inject malicious code, and/or hijack control flow. The JIT-ROP attack presented by Snow et al. [88], for example, showed that a single memory corruption...
vulnerability in a JavaScript VM allowed a malicious script to achieve arbitrary code execution, bypassing the VM’s security mechanisms.

To make matters worse, dynamic code generation is naturally susceptible to various types of exploits. The memory region that contains the bytecode or machine code must be writable while the code is being generated. This weakens one of the most effective defenses against code injection, Data Execution Prevention (DEP), as the code cache does need to be both writable and executable (though not necessarily at the same time). Song et al. showed that it is possible to exploit the time window where JIT’ed code is writable to inject code stealthily [90]. While generating code, the VM also produces intermediate data such as bytecode and data constants which are used as input for subsequent code generation phases. Tampering with this intermediate data may also give an attacker arbitrary code execution capabilities, without having to directly hijack control-flow or corrupt the code section [95, 38]. Furthermore, these problems may be worse in bytecode interpreters than in JIT engines.

Contrary to JIT’ed code, bytecode does not require page-level execute permissions as it is executed by an interpreter and not by the CPU. Malicious bytecode can therefore still be injected, even if DEP is enabled.

Several recently proposed defense mechanisms mitigate code injection attacks on VMs [90, 95, 38]. Frassetto et al. proposed to move the JIT compiler and its data into a secure enclave to defend against intermediate representation (IR) code corruption attacks [38], while Microsoft added verification checksums to detect corruption of temporary code buffers in the JavaScript engine of its Edge browser [95]. However, these defenses focus solely on protecting JIT-based VMs and overlook bytecode interpreters. This is not entirely surprising because there is the belief that the potential impact of a code injection-attack on a bytecode interpreter is limited. It is assumed that injected bytecode cannot access arbitrary memory addresses or call arbitrary functions, for example. We contradict this belief by showing that bytecode injection is a realistic attack vector with potentially high impact. Specifically, we
present several attack strategies that may be pursued to achieve arbitrary code execution in a well-protected bytecode interpreter, even if that interpreter employs a language-level sandbox to disable access to dangerous APIs and to introspection features. Our attack allows scripts to perform operations or to interact with the host system in a way that normally would not be allowed by the sandboxed interpreter.

We implement our attack in the standard Python and Lua interpreters. Python and Lua are widely used to write add-ons and plugins for large applications. Bugs in these applications may allow remote attackers to execute arbitrary scripts (cf. Sect. 3.3). Attackers can also disguise malicious scripts or packages as benign software and distribute them through standard package managers and distribution channels where users may unknowingly download them [100]. By using the attack techniques presented in this chapter, scripts downloaded through such channels can perform malicious actions even if the user executes them in an interpreter with a language-level sandbox (that normally prohibits such actions).

Finally, we present a simple and effective defense against bytecode injection that can be deployed without hardware support and with limited run-time overhead.

3.2 Background

Bytecode interpreters translate input programs into bytecode that encodes instructions for a virtual stack or virtual register machine. Most virtual stack machine instructions operate on data that is stored on a stack. An integer addition instruction, for example, typically pops the top two elements off the stack, adds them together, and pushes their sum onto the stack. By contrast, instructions for a register machine operate on data that is stored in addressable registers. An integer addition instruction for a register machine could load its input operands from registers \( R1 \) and \( R2 \), add them, and store the result in register \( R3 \).

Regardless of the type of virtual machine that is being emulated, the size of the bytecode
instruction set is small compared to a typical instruction set for a physical architecture. The latest version of the x86_64 instruction set contains well over a thousand instructions, whereas the latest version of the bytecode instruction set used in CPython contains just over a hundred instructions.

3.2.1 Bytecode Storage

DEP prevents both static and JIT-compiled machine code from being executed while it is writable and vice versa. DEP is ineffective for bytecode, however, because bytecode can be executed even when it is stored on non-executable pages. To prevent bytecode from being overwritten while it is being executed, the interpreter should mark the bytecode as read-only. This is generally not possible because most interpreters store bytecode on the heap, where it typically resides on the same pages as data that must remain writable at all times.

As a consequence of this design decision, it is possible to overwrite the bytecode even while it is executing. It is also easier to discover the location of the bytecode cache than the location of machine code. While the latter requires an arbitrary memory read vulnerability [88], we show that it is possible to discover the location of the bytecode cache with a vulnerability that can only reveal the contents of the heap.

3.2.2 Data Encapsulation

Interpreters typically encapsulate all program variables, including those of primitive types, into objects. Every object has a header which identifies the type of the encapsulated value. Figure 3.1, for example, shows two objects representing an integer and a string value. While the integer object only contains one field that stores the actual integer value, the string object has multiple fields to store different properties of the string.
3.2.3 Data Access

One of the most notable differences between machine instructions and bytecode instructions is how they access program data. Machine instructions typically encode register numbers, memory addresses, and word-sized constants as instruction operands. Bytecode instructions, by contrast, refer to memory addresses and constants via an indirection layer.

Figure 3.2 illustrates this difference by showing a bytecode instruction that loads a constant onto the stack. The constant is not embedded in the instruction itself. Instead, the instruction’s operand denotes an entry in a constant table. This entry also does not contain the constant itself but instead refers to the address of the object that encapsulates the actual constant. These indirection layers limit the capabilities of an attacker who only can manipulate bytecode. Specifically, an attacker cannot load/store arbitrary constants/variables, call
Table 3.1: Supporting data structures in Python and Lua. Functions have their own constant and locals tables, but share globals tables.

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Opcodes</th>
<th>Data Structures</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>LOAD_CONST</td>
<td>Constant Table</td>
<td>Array</td>
</tr>
<tr>
<td></td>
<td>LOAD_FAST</td>
<td>Fast Locals Table</td>
<td>Array</td>
</tr>
<tr>
<td></td>
<td>STORE_FAST</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOAD_NAME</td>
<td>Name Table</td>
<td>Hash Map</td>
</tr>
<tr>
<td></td>
<td>STORE_NAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOAD_GLOBAL</td>
<td>Globals Table</td>
<td>Hash Map</td>
</tr>
<tr>
<td></td>
<td>STORE_GLOBAL</td>
<td>Builtins Table</td>
<td></td>
</tr>
<tr>
<td>Lua</td>
<td>OP_LOADK</td>
<td>Constant Table</td>
<td>Array</td>
</tr>
<tr>
<td></td>
<td>OP_MOVE</td>
<td>Locals Table</td>
<td>Array</td>
</tr>
<tr>
<td></td>
<td>OP_GETUPVAL</td>
<td>Upvalue Table</td>
<td>Array</td>
</tr>
<tr>
<td></td>
<td>OP_SETUPVAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OP_GETTABUP</td>
<td>Globals Table</td>
<td>Hash Map</td>
</tr>
<tr>
<td></td>
<td>OP_SETTABUP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

arbitrary functions, or access arbitrary memory locations without the help of built-in tables. To perform a system call, for example, there must be a table entry that refers to a function object, which in turn contains the address of a system call function. The attacker needs to manually inject this entry and the function object because they are unlikely to exist while the interpreter executes benign scripts.

Table 3.1 lists the supporting data structures and their related bytecode instructions for Python and Lua. When executing a bytecode instruction supported by an array-typed data structure, the interpreter treats the bytecode’s operand as an index into the array. The LOAD_CONST instruction, illustrated in Fig. 3.2, is one example of such an instruction.

Instructions supported by a hash map-typed data structure, such as LOAD_GLOBAL, shown in Fig. 3.1, access their target through a triple indirection. First, the interpreter uses the instruction’s operand as an index into a key table containing strings. The interpreter loads the string that the instruction points to, hashes it, and uses the hash value as an index into a hash map table (i.e., the global hash table in this case). Then, the interpreter loads the
object reference from the hash map table, and loads the data stored in this object.

### 3.2.4 Function Calls

Any function that is called from within a script has a function object associated with it. The function object contains a field indicating the type of the function (i.e., bytecode or native), as well as a pointer to the function’s executable code (for native functions), or the function’s bytecode object (which, in turn, points to the bytecode string).

To call a function from within bytecode, the caller first loads the function arguments onto the interpreter stack. Then, the caller loads the target function object from the name table (which is a hash map-like data structure). Next, the caller uses a function call bytecode instruction to prepare an argument list array, and to invoke the interpreter’s call dispatcher function.

This dispatcher function receives pointers to the target function object and the argument list as its arguments. If the target function is a native C function, the dispatcher will call that C function with a context pointer as its first argument and a pointer to the argument list as its second argument. This context pointer is stored in the function object itself, and can therefore be overwritten by an attacker.

### Calling Arbitrary C Functions

The set of C functions that can be called by overwriting or injecting function objects on the heap is limited. The reason is that C functions normally expect to find their arguments in certain registers or stack slots, as stipulated in the platform’s application binary interface standard.

However, bytecode interpreters pass arguments differently when calling C functions. Specif-
ically, the aforementioned dispatcher function passes pointers to the context and to the argument list structure as the sole arguments to any C function. The context pointer is an implementation-specific pointer that can usually be controlled by the attacker. The argument list pointer, however, cannot be controlled by the attacker. Moreover, unless the C function is aware of the calling conventions used by the interpreter, it will not correctly extract the actual arguments from the argument list.

Consequently, the set of C functions that an attacker can call by corrupting function objects only includes functions that expect less than two arguments and functions that are aware of the calling conventions used in the interpreter.

### 3.2.5 Dangerous Interpreter Features

Most bytecode interpreters are designed under the assumption that the end-user will only run benign scripts on benign inputs. These interpreters therefore implement many features that could be abused if either the script or its inputs turn out to be malicious. Recurring examples of such features include the following.

**The eval function.** First introduced in LISP, the eval function present in many interpreted languages parses a string argument as source code, translates the source code into bytecode, and then executes the bytecode. Many remote code execution vulnerabilities in scripts are caused by allowing attackers to supply the string argument to eval (e.g., CVE-2017-9807, CVE-2016-9949, and scientific literature [78]).

**Direct bytecode access.** Many scripting languages, including Python and Lua, treat functions as mutable objects with a bytecode field that contains the raw bytecode instructions for their associated function. The script can read and overwrite this bytecode field, either directly or through an API. Python scripts, for example, can access bytecode through the `_code._co_code` field that exists in every function object, whereas Lua scripts can
use the `string.dump` and `load` functions to serialize the raw bytecode instructions for a
given function object and deserialize raw bytecode instructions into a new function object
respectively.

**Dynamic script loading.** Scripting languages often allow loading and execution of addi-
tional script code stored in the file system. Python, for example, supports the `__import__`
function to load modules dynamically, whereas Lua provides the `require` function for this
same purpose. An attacker that controls the arguments to these functions may be able to
introduce malicious code into an otherwise benign script.

**Native code support.** Most bytecode interpreters including CPython and Lua support
calling native code from the interpreted bytecode through a so-called Foreign Function In-
terface (FFI). The FFI allows the language to be extended with functionality that is not
or cannot be made available within the scripting language itself. From a security perspec-
tive, the disadvantage of the FFI is that it can extend the attack surface of the interpreter.
Like the interpreter itself, functions called through the FFI are often written in C or C++,
which are neither type- nor memory-safe. Vulnerabilities in such functions therefore affect
the entire interpreter.

**Fully-featured APIs for accessing system resources.** Python and Lua both expose
APIs for creating, modifying, and deleting system resources such as files, sockets, threads,
and virtual memory mappings. The reference interpreters for both languages impose no
restrictions on how the script uses these APIs. Typically, the API invocations are only
subject to access control checks by the OS itself, and the script therefore runs with the same
privileges of its invoker.
3.2.6 Running Untrusted Scripts

If a bytecode interpreter is used to run untrusted scripts, it is often necessary to restrict or block access to the dangerous features described in Sect. 3.2.5 or even remove them altogether. Broadly speaking, there are two different approaches to restricting access to dangerous language/interpreter features.

**Language-level sandboxing.** A language-level sandbox restricts access to dangerous features by intercepting, monitoring, and (potentially) manipulating function calls within the interpreter itself. As an example, you can build a language-level sandbox for Java programs based on the Java Security Manager [73]. This Security Manager wraps calls to dangerous functions to perform fine-grained access control checks. Similarly, lua_sandbox wraps internal interpreter functions to disable script access to certain Lua packages and functions [42].

Language-level sandboxing can also be achieved through source code-level transformations. Caja, for example, transforms untrusted HTML, CSS, and JavaScript code to isolate it from the rest of a web page [48]. RestrictedPython similarly rewrites Python bytecode to restrict access to certain APIs [43].

Finally, one can just remove dangerous functionality from the interpreter altogether, which is viable if the sole purpose of the interpreter is to run untrusted scripts. An example of such a stripped-down interpreter is the Python runtime environment in Google App Engine [47], which does not support native code, does not support direct bytecode access, and does not contain certain system APIs (e.g., for writing to the file system).

The advantage of language-level sandboxes is that they can deploy fine-grained access control checks to not just the APIs for accessing system resources, but also to internal functions that can be invoked without interacting with the OS. The disadvantage is that language-level sandboxes lack a hardware-enforced boundary between the sandbox and the potentially
malicious script or program. Malicious scripts can therefore escape from such sandboxes if any part of the interpreter contains an exploitable memory vulnerability.

**Application-level sandboxing.** Application-level sandboxes restrict access to system resources by interposing on the system calls made by the interpreter. Since 2005, Linux offers the `seccomp` API for this purpose, while older sandboxes could build on the `ptrace` infrastructure.

The advantage of application-level sandboxes over language-level sandboxes is that they are protected from the interpreter by a hardware-enforced boundary (enforced through the memory paging mechanism). The disadvantages are that they can only restrict access to system APIs, and not to internal interpreter functions. Ideally, an interpreter therefore uses both language-level sandboxing and application-level sandboxing techniques when running untrusted scripts.

### 3.3 Threat Model And Assumptions

The goal of our work is to achieve arbitrary code execution through injected bytecode and data. Our threat model therefore excludes attacks that corrupt static code or that introduce illegitimate control flow in the static code (i.e., ROP attacks). Our model is consistent with related work in this area [15, 38, 90].

**Strong protection for static code.** We assume that the target system deploys state-of-the-art protection for static code. Specifically, we assume that Address Space Layout Randomization (ASLR) is enabled and applied to the stack, heap, main executable, and all shared libraries. We assume that machine code-injection attacks are impossible because Data Execution Prevention (DEP) is enforced. We assume that code-reuse attacks are mitigated by fine-grained control-flow integrity [3, 96, 72].
Memory vulnerability. We assume that the bytecode interpreter has a memory vulnerability that allows attackers to allocate a buffer on the heap, and to read/write out of the bounds of that buffer. The CVE-2016-5636 vulnerability in CPython is one example that permits this type of buffer overflow. Note that we do not assume an arbitrary read-write vulnerability.

Interpreter protection. We assume that the interpreter deploys a language-level sandbox (cf. Sect. 3.2.6) that disables all dangerous features listed in Sect. 3.2.5. Consequently, we assume that the scripts cannot access or modify bytecode directly. We further assume that there is no application-level sandbox in place. If the interpreter does use an application-level sandbox, then our attack by itself does not suffice to escape from the sandbox. It could, however, serve as a useful building block for a sandbox escape attack.

Attacker. We assume that the attacker can provide the script to be executed by the protected interpreter. The attacker-provided script does not contain any malicious features that will be blocked by the language-level sandbox. We also assume that the attacker knows the version and configuration of the interpreter, and that the attacker can run this same version locally on a machine under his/her control.

3.3.1 Realism

The assumption that an attacker can provide the script to be executed by the victim is realistic. Many large applications can be customized through Python or Lua scripts, and have App Store-like distribution systems where developers can freely share their scripts with other users of the same application.

Numerous video games [61], including the hugely popular World of Warcraft (WoW), for example, allows users to write Lua scripts to customize the game interface. Developers can upload these add-ons to dedicated fan sites, where they are downloaded by millions of
users. Another example is Python’s package manager PyPi, where rogue packages have been known to appear [100]. Packages downloaded through PyPi can subsequently be used in any Python-compliant interpreter, including interpreters with a language-level sandbox.

These script distribution systems usually lack the developer verification and malware scanning features that are commonly employed by application stores for mobile platforms. It is therefore relatively easy to disguise a malicious script as a legitimate piece of software, and to distribute it to a lot of users.

An attacker could also inject malicious script code into other (benign) scripts. We studied recent CVEs from 2014 to 2018 and found several examples of vulnerabilities that permit such script injection attacks (e.g., CVE-2017-9807, CVE-2017-7235, CVE-2017-10803, CVE-2016-9949, CVE-2015-6531, CVE-2015-5306, CVE-2015-5242, CVE-2015-3446, CVE-2014-3593, CVE-2014-2331). The CVE-2017-9807 vulnerability in OpenWebif, for example, existed because OpenWebif called Python’s eval function on the contents of an HTTP GET parameter. An attacker could exploit this vulnerability by submitting a full script as part of this parameter.

### 3.4 Attacking Bytecode

Our attack achieves arbitrary code execution in a bytecode interpreter by simultaneously overwriting the bytecode of a single function and the supporting data structures accessed by that function (e.g., the constant table). Overwriting just the bytecode generally does not suffice, because that would force us to reuse only existing constants and variables.

#### 3.4.1 Attack Overview

The attack proceeds in five steps:
Figure 3.3: Overview of our heap layout inference step. By disclosing heap contents and identifying three data structures belonging to the same target function, we can subsequently follow pointers to other data structures by calculating their offset relative to the vulnerable buffer.

**Preparation:** We load a script that contains an attacker function, at least one target function, and a blueprint buffer for the injected code. Target functions are benign functions whose bytecode and supporting data structures are easily recognizable in memory. Each target function contains a unique sequence of harmless operations that is translated into an easily identifiable bytecode string. We also use a unique and large number of constants and local variables in each target function, which allows us to recognize the function’s constant/locals tables.

The blueprint buffer contains the raw bytecode sequence the attacker wishes to inject. In most cases, we cannot inject the blueprint buffer as-is, because its bytecode attempts to load
data from data structures we cannot overwrite. We therefore rewrite the blueprint buffer in a later step to ensure that it accesses the correct data.

Note that the attack script itself looks benign to the interpreter. The script does not use introspection features, nor does it call any privileged APIs or perform privileged operations that are normally stopped by the interpreter’s security mechanisms. The code we inject, by contrast, is not benign and does violate the interpreter’s security policies, although it does so without being detected.

**Heap layout inference:** The goal of our second step is to infer the precise layout of a large portion of the heap. The layout information we infer includes the absolute positions (i.e., addresses) of the bytecode and supporting data structures of both the attacker function and at least one of the target functions.

We begin this step, which is illustrated in Fig. 3.3, by executing the attacker function. The attacker function allocates a buffer, and then leverages the buffer overflow vulnerability to read outside the bounds of that buffer, thereby leaking the contents of the heap. Based on these contents, we can determine the positions of a set of data structures relative to the vulnerable buffer. We do so as follows. First, we search for one of the target functions’ bytecode strings. These are easily recognizable since the bytecode for each target function is known and remains the same across executions of the interpreter.

Once we have identified a bytecode string for a target function, we proceed to finding its constant table. The constant table is filled with pointers, which we cannot follow at this point because that would require an arbitrary read vulnerability. Therefore, we cannot examine the contents of a constant table to determine to which function it belongs. Instead, we read the type and size fields for any potential data structure we encounter. Lua uses a bitfield to encode data structure types, so constant tables have a fixed value in the type field. CPython’s type fields are pointers to constant strings, which are always loaded at the
same address (relative to the image base). In both cases, potential constant tables are easily recognizable.

Once we have determined that a data structure is a potential constant table, we read its size field. Our attacker script ensures that each target function has a large and unique number of constants in its constant table. We do this by declaring a local variable which stores a list of constant numbers in each target function. The size value of the constant table therefore uniquely defines to which function it belongs.

Having identified the bytecode string and the constant table for a specific target function, we now attempt to find the bytecode object for that function. Again, we can recognize potential bytecode objects based on their type field. Once we identify a potential bytecode object, we can determine if it belongs to the same function as the already identified bytecode string and constant table by verifying that the distance between the bytecode string pointer value and the constant table pointer value in the bytecode object matches the distance between the data structures we identified. If these distances match, we assume that we have found the right bytecode object, and that we now know the absolute addresses of the bytecode object, bytecode string, and constant table we disclosed.

At this point, we can follow any heap pointer to an address that is higher than that of the vulnerable buffer, and we can ultimately disclose the full layout of the heap area that follows the vulnerable buffer. We expect that the attacker will also be able to find at least one code pointer on the heap, thereby identifying the base address of the interpreter’s executable code section. This is necessary to locate the C functions we wish to call in our attack. Recent work shows that this is a realistic expectation [98].

**Attack strategy selection:** Based on the heap layout information, we can select an attack strategy and inject the payload. The payload injection is subject to three constraints. First, we cannot write any data at addresses lower than that of the vulnerable buffer, because
the vulnerability we are exploiting allows buffer read/write overflows, but not underflows. Second, for the same reason of the first constraint, the payload we inject must be contiguous. Third, we must be careful when overwriting the currently executing bytecode string or any of the data structures that may be accessed by the currently executing code, since doing so might crash the interpreter.

As a result of these three constraints, it is not always possible to overwrite the bytecode string and the constant table of a target function. We have therefore devised multiple attack strategies, each targeting different data structures. We describe these strategies in Sect. 3.4.2.

**Payload construction:** We now craft the attack payload, which consists of a bytecode string, and a data structure containing references to the data the attacker wishes to use in the injected bytecode. We provide more details on the payload construction in Sect. 3.4.2.

**Execution:** At this point, we overwrite the bytecode and data structures we identified in the second step with the payload crafted in the fourth step. Finally, we transfer control to the target function we overwrote to trigger our injected code.
3.4.2 Crafting the Payload

We craft the payload based on an attacker-provided blueprint buffer. The blueprint buffer contains the raw bytecode string to be injected. The attacker must additionally provide information about the data to be used in the injected bytecode.

Figure 3.4 shows the process of creating a blueprint buffer, and converting it into an attack payload. The attacker begins by writing a blueprint function in an off-line instance of the interpreter. We assume that this off-line instance of the interpreter is controlled by the attacker, and that it provides access language introspection features (unlike the target interpreter the attacker wishes to attack).

The blueprint function has the desired attack semantics, but does not necessarily operate on the desired data. For example, if the attack should call a C function that is not normally exposed to the scripting language, then the attacker can just write a blueprint function that calls a different (accessible) function instead, dump the blueprint, and adjust the target of the function call while crafting the payload.
Rewriting the Blueprint

The attacker now rewrites the blueprint buffer into a concrete attack payload that works in the target interpreter. Depending on the inferred heap layout, the types of the disclosed data structures, and whether or not these data structures can be safely overwritten, the attacker can pursue one of four rewriting strategies. For simplicity, we explain our strategies using Python’s bytecode convention. All strategies apply to the Lua interpreter as well, however, needing only trivial modifications.

**Strategy 1: Overwriting a bytecode string and constant table**  The attacker chooses this strategy if the attacker has disclosed a constant table and the table’s entry size is larger than or equal to the number of load instructions in the blueprint buffer. In the blueprint buffer, the function object is loaded from the *name table*. The attacker therefore needs to adjust the load instruction to a load from a constant table, i.e., `LOAD CONST $id`. The attacker also needs to inject the objects with the prepared data and update the constant table so that each table entry points to the injected object. The resulting payload of Strategy 1 is shown in Fig. 3.5. `LOAD_CONST 0` loads a function argument, "/bin/sh", then `LOAD_CONST 1` loads the function object overwritten with the address of `posix_system()` function which is a wrapper function in CPython that unboxes argument objects and calls C `system()` function with the unboxed arguments. Next, we use a `FUNC_CALL` instruction to call the injected C function. We could also call the `system()` function directly because it expects just one argument (cf. Sect. 3.2.4). In both cases, we are able to launch a system shell, which is normally not allowed in the sandboxed interpreter.

**Strategy 2: Overwriting a bytecode string and hash map-like table**  If the attacker only found a hash map-like table or the target constant table size is too small to cover all the load instructions in the blueprint buffer, the attacker selects this strategy. Manipulating hash map-like structures is challenging, however, due to the multi-level indirections and the
use of a hash function (see Sect. 3.2.3). The underlying idea of this strategy is to simplify the hash map manipulation by making the key table entries point to integer objects instead of string objects. This way, the attacker can access the hash map as if it were an array-like structure.

The implementation details can vary depending on how the interpreter accesses the hash map-like table. In CPython, the interpreter maintains a dedicated key table for all hash map-like structures. The LOAD_GLOBAL instruction fetches a key from the global key table, and then uses this key as an index into the global hash table. In this architecture, the attacker can overwrite the key table so that each key table entry points to an integer object written by the attacker. Lua, on the other hand, requires two bytecode instructions to load data from a hash map, one for loading the key and one for fetching the value. Moreover, Lua does not maintain dedicated key tables for any of its hash maps. Instead, the key can be loaded from any array-like table. The attacker can therefore convert an existing array-like tables into a key table and fill it with references to integer objects.

Similar to Strategy 1, the attacker replaces the bytecode dump of previous load instructions with that of the bytecode sequence for accessing hash map-like structures as described above. The attacker then changes entries in the hash map which point to attacker’s objects.

**Strategy 3: Overwriting a bytecode string and loadable object** If the attacker is unable to update entries in any tables, he can shape his payload as a single function using this strategy. Instead of using existing tables, the attacker crafts a constant table and adjusts the bytecode and updates the data according as in Strategy 1. The attacker then prepares a bytecode object pointing to the adjusted bytecode buffer and attacker’s constant table. To be able to load this bytecode object to the interpreter’s stack (or to a register in a register-based machine), the attacker has to overwrite any loadable object with the bytecode object. To do so, the attacker prepares a unique constant object in the preparation step so that its data
The payload created based on Strategy 4.

Structure can be easily found in the heap layout inference step. The attacker then overwrites this constant object with the bytecode object, thereby the attacker’s bytecode object can be loaded on the stack through the constant table. Based on this loaded bytecode object, the attacker makes a function object which itself becomes the attacker’s payload to call the associated function. To do so, the target function’s bytecode should be overwritten with two bytecode instructions. One is to create a function object using a bytecode object loaded on the interpreter’s stack. The other is to call the function in the function object.

Strategy 4: Injecting bytecode and overwriting a bytecode object In Strategy 4, the attacker injects bytecode instead of overwriting the existing bytecode buffer. To this end, the attacker injects the bytecode on the heap and overwrites the bytecode pointer in the bytecode object with the address of the injected bytecode as shown in Fig. 3.6. Before injecting the bytecode, the attacker still needs to adjust the bytecode in the blueprint buffer and update the prepared data according to the available data structures (again, the same step as in Strategies 1 and 2).
3.5 Crafting a Defense

We designed and prototyped a defense that thwarts the presented as well as other bytecode injection and overwrite attacks. The main goal of our defense is to protect the integrity of bytecode. The design of our defense is inspired by existing defenses against code cache corruption attacks \cite{95, 58}. We propose two defense techniques: making bytecode strings read-only, and verifying bytecode targets during function calls. When combined, this effectively defeats all four of our attack strategies.

First, as shown in Fig. 3.7, we make all bytecode strings read-only so that the attacker cannot overwrite them. This specifically stops attack strategies 1 through 3, which overwrite the bytecode string of a target function. We implemented this feature by modifying the interpreter’s memory manager and parser. Normally, when the interpreter parses a source function and translates it to bytecode, it allocates and stores that bytecode on the heap. We modified the interpreter to allocate a dedicated page for each function’s bytecode string, and mark this page as read-only when the source function is fully translated.

This first defense technique prevents valid bytecode strings from being overwritten. However, it does not prevent bytecode injection attacks. Specifically, an attacker can still inject...
bytecode on the heap, and overwrite the bytecode string pointer in a bytecode object to point it to the injected bytecode instead. We implemented a second defense mechanism that prevents this type of attack. Concretely, we added a bytecode pointer verifier that checks the integrity of a function’s bytecode pointer whenever it is called.

We extended the interpreter’s parser to generate bytecode pointer checksums whenever it finalizes the translation of a source function into bytecode. We generate these checksums by calculating the hash of the concatenated value: $BytecodePointerValue||BytecodePointer - Location$.

As our hash function, we used the publicly available HighwayHash, which is an optimized version of SipHash. Both SipHash and HighwayHash are keyed hash functions. We generate a random hash key when the interpreter starts and prevent it from leaking by (i) keeping it stored in a dedicated CPU register at all times, (ii) using gcc’s -ffixed-reg option to prevent reuse or spilling of that register, and (iii) customizing the hash function so it loads the hash key from the dedicated register and so it restores the old values of all registers that we might move the key into. Our bytecode pointer verifier recalculates and verifies the checksum whenever the interpreter invokes a bytecode function. The verifier effectively prevents strategies 3 and 4, which rely on a malicious function call, because the checksum verification will fail before the attacker’s bytecode is executed.

3.6 Evaluation

We implemented our attack and defense for two commonly used bytecode interpreters: CPython 2.7.12 and Lua 5.3.2. We retrofitted a slightly altered version of a known heap buffer overflow vulnerability into CPython (CVE-2016-5636) and added a similar bug to Lua. We constructed an attack that launches a shell by calling `system("/bin/ls")`. We verified that all four of our proposed attack strategies succeed in both interpreters.
We also evaluated the run-time performance impact of our defense by running the Python Performance Benchmark Suite [77] for CPython and the Computer Language Benchmarks Game [5] for Lua. We ran these benchmarks on a GNU/Linux system running Ubuntu 14.04 LTS x64. The system has an eight-core Intel Xeon E5-2660 CPU and 64Gb of RAM. Fig. 3.8 and 3.9 show our results.

The run-time performance impact of the first part of our defense (making bytecode read-only) is generally negligible. Only `hg_startup`, `python_startup`, and `python_startup_no_site` slow down noticeably. These benchmarks measure the startup time of the interpreter, which is generally short, but do not measure the execution of any bytecode. The other benchmarks do include execution of actual bytecode.

In these other benchmarks, our checksum verification incurs run-time overheads of less than 16% on average. Since our checksum verification checks occur at every function call, the overhead is directly proportional to the number of function calls and returns. `spectralnorm`, and `binary_trees` benchmarks in Lua execute a significant number of recursive functions, which produces numerous function calls and returns, and thus high overhead.

### 3.7 Security Analysis

While our defense successfully stops all four of our proposed strategies, an attacker could still attempt to bypass it as follows:

**Pure data-injection attacks:** The current implementation of our defense only protects against bytecode overwrite and injection attacks. While this suffices to thwart all four of our proposed attack strategies, we do believe it might be possible to mount a pure data-injection attack that also achieves arbitrary code execution. In such an attack, the attacker would overwrite or inject new data to alter the behavior of a benign function without overwriting
that function’s bytecode.

To block these attacks, one can apply the same conceptual techniques we proposed in this chapter to protect all of the interpreter’s data structures. Immutable data structures such as
constant tables can be moved to read-only pages, while mutable structures can be extended with verification checksums.

**Partial state corruption attacks in multi-threaded scripts:** The bytecode interpreters we evaluated parse and translate source functions into bytecode lazily (i.e., when the function is first called). Therefore, there is a time window after the interpreter has fully initialized for which source functions may be stored in a partially translated state in writable memory. Recent work by Song et al. [90] and Frassetto et al. [38] showed that it is possible to overwrite this partially generated state in interpreters that support multi-threaded scripting languages. Our defense is, in principle, also vulnerable to such attacks. To prevent such attacks, we could offload the parsing and bytecode translation to an external process, as was done by Song et al. [90].

**Checksum forging:** We protect pointers to bytecode strings with a verification checksum to prevent attackers from forging bytecode objects pointing to bytecode strings stored on the heap. If an attacker can create a bytecode object with the correct checksum, our defense would not be able to detect that the bytecode string it points to is stored in writable memory. We prevent such attacks by using a keyed hash function, and by storing the key in a dedicated register which is never leaked. We also prevent attackers from reusing correct bytecode pointers and checksums to redirect one bytecode function to a different, legitimate bytecode function. We achieve this by using not just the bytecode pointer itself, but also the location where it is stored as input to our hash function.

**Checksum alternatives:** As an alternative to our bytecode pointer checksum, we could have used a true HMAC (as was done in Subversive-C [58]), or a MAC-AES (as was done in CCFI [64]). We opted not to use an HMAC because our input (i.e., the concatenation of the bytecode pointer and its storage location) is fixed-length. An HMAC therefore does not increase security over our scheme. We did not implement the MAC-AES scheme used
in CCFI because it requires many reserved registers, as opposed to just one register in our case.

3.8 Conclusion

We presented an attack that achieves arbitrary code execution in bytecode interpreters that deploy language-level security mechanisms to prevent unauthorized access to files, sockets, or APIs. Our attack leverages a heap-based buffer overflow vulnerability in the interpreter to leak the contents of its heap, infer the heap layout, and overwrite or inject bytecode and program data. We also presented a defense that thwarts our attack by moving all bytecode to read-only memory pages, and by adding integrity checks to all bytecode pointer dereferences.

We evaluated our attack and defense on CPython 2.7.12 and Lua 5.3.2. Our evaluation shows that our defense incurs an average run-time overhead of less than 16% over a large set of Python and Lua benchmarks.
Chapter 4

Locking Down JavaScript Engines

4.1 Motivation

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 [80, 103].

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 [87, 24]. Later JIT engines added support for W⊕X poli-
cies 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 [15, 9, 59]. Defenders quickly thwarted these attacks through the use of constant blinding [15], constant elimination and code obfuscation [25], code randomization [50], or control-flow integrity [71].

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 [88]). A number of schemes protect against such attacks by leveraging randomization and execute-only memory [32, 11, 10, 41].

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. [95] and Frassetto et al. [38] 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 [66, 90] or hardware-based trusted execution environments [38].

In this chapter, 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 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, that we have given the name NoJITsu, to defend the JIT engine against a wide variety of runtime 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.
Figure 4.1: 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.

4.2 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.

4.2.1 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 [83, 49], Meltdown [60], and Spectre [53] are orthogonal to software vulnerabilities and outside the scope of this work. Our threat model is in line with those of related work in this area [15, 9, 59, 25, 50, 71, 38].

- **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 $W \oplus X$ policy [65], and, thus, that no memory pages are ever simultaneously writable and executable. Some engines enforce $W \oplus X$ by offloading the JIT compilation to an external process [66, 90] or trusted execution environment [38], while others simply toggle the writable and executable permissions on JIT pages at run time [69].

- **Code-reuse defense.** We assume that the browser uses all code-reuse defenses that have seen widespread adoption. These defenses include, among others, ASLR [76] 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) [30] 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 uninten-
tional occurrences of instructions to modify in-register PKRU values.

4.2.2 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 4.1)\(^1\) 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 dispatch.

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

\(^1\)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.
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.

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.

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 [84]. Next, we will demonstrate that an adversary can construct successful exploits by attacking the interpreter, bypassing these proposed mitigations.

### 4.2.3 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 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 4.2. 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.
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 [14]. Weaponizing the bug takes four steps. First, the JavaScript program allocates a number of small and consecutive ArrayBuffer on the heap. We gave all of the ArrayBuffer a size of 32 bytes in our exploit. Then, the program creates Uint32Array and Uint8Array view objects for one of the allocated ArrayBuffer. 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 ArrayBuffer. 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.

After weaponizing the bug, we leak the locations of the JSContext and victim JSFunction objects as illustrated in Figure 4.3. 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.

**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 [66, 38] since we attack the interpreter which always executes before any JIT compilation is invoked.
4.3 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 [66]. 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 [15], constant elimination and code obfuscation [25], code randomization [50], or control-flow integrity [71].

4.3.1 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 4.4, 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 sophis-
ticated 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 per-
mission 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 4.5 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.

Figure 4.6 shows the overall design of our JavaScript engine protection, NoJITSu, which
Figure 4.4: Legacy design

Figure 4.5: The design of NoJITsu
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 for individual memory domains without modifying page table entries or flushing the TLB. Hence, in contrast to traditional MMU-based protection mechanisms [41, 10, 32, 11], 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.

**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
Figure 4.6: 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.

This data separation to minimize the potential performance impact (see Section 4.4.2). 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) [15] and JIT-ROP attacks (which rely on reading code) [88].

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 [63]. JIT spraying attacks using one- and two-byte constants are, however, feasible in practice [9]. Similarly, there are several existing implementations of execute-only memory,
but they only apply to statically generated code [41, 10, 32, 11, 18, 67, 27]. This leaves these implementations vulnerable to JIT-ROP attacks that only use gadgets found in the JIT code cache.

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) [67] to enforce execute-only permissions for the static code regions.

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 [90, 38]. 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.
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.

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 4.4.3).

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.

4.4 Implementation

We applied our defense to SpiderMonkey 60.0, which was released in late 2018 [70]. 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 4.4.3), which we conducted offline. For this last step, we used LLVM 8.0.0 to modify and transform SpiderMonkey’s code automatically [56]. 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.

4.4.1 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 4.5, NoJITsu provides protection against a wide variety of attacks, including arbitrary code execution attacks based on code injection and code-reuse.

4.4.2 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.
Jump Table Separation

During JIT compilation, SpiderMonkey emits jump tables such as those shown in Figure 4.7a. 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 4.7b 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.

Permission Change Routine

Listing 4.1 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 [75]). Thus, instead of immediately writing to the PKRU register, the `set_pkru` function first checks if the current PKRU value
Figure 4.7: 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.

already has the write permission. If so, the function returns without overwriting the register. If the page does not have the write 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_pkr` 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 work, however, potential mitigations orthogonal to our work are: i) to avoid spilling of registers containing PKRU values by assigning it a dedicated register [34], or
ii) to randomize the PKRU values before being spilled to stack and keep the randomization secret in a dedicated register [19, 13].

**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 4.6, for example, we assign key3 and key4 for different types.
of JavaScript objects.

```javascript
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 4.1: Permission change routine

### 4.4.3 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 [93, 57, 81]. 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.

**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.
Dynamic Object-Flow Analysis

Listing 4.2 and Listing 4.3 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.

**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 4.5.2). 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 func-
tions 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.

4.4.4 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 4.4.3. 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.

4.4.5 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 [75, 97]. 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 4.5.3).

```c
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(...);
        ...
        unitStaticTables[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 4.4: Example: Redundant calls

**Code Example**

We show a code example that can highly benefit from our optimization. In Listing 4.4, 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.

\[
Score = \begin{cases} 
1 & \text{if accessor function,} \\
\frac{\sum \text{score of called function}}{\text{number of function calls}} & \text{otherwise.}
\end{cases}
\] (4.1)
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 4.1).

![Figure 4.8: Example call graph and scores for each node. The example is based on our heuristic to determine nodes to insert permission changes.](image)

We demonstrate how we score each function in Figure 4.8, 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 4.8, 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, 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.

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.
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 4.8, 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.

4.5 Evaluation

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

4.5.1 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 4.3, the default access permissions are locked down significantly within NoJITsu for each of the components we identified in Section 4.2.2. 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.

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 4.2. We insert \texttt{set\_pkey} and \texttt{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
Table 4.1: Percentage of the functions that need write permissions

<table>
<thead>
<tr>
<th></th>
<th>Primitive obj</th>
<th>Sensitive obj</th>
<th>Both obj</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single write</td>
<td>0.09%</td>
<td>0.16%</td>
<td>0.05%</td>
<td>0.29%</td>
</tr>
<tr>
<td>Block write</td>
<td>0.04%</td>
<td>0.02%</td>
<td>0.01%</td>
<td>0.07%</td>
</tr>
<tr>
<td>Total</td>
<td>0.13%</td>
<td>0.18%</td>
<td>0.06%</td>
<td>0.36%</td>
</tr>
</tbody>
</table>

Table 4.2: Percentage of the write instructions executed in the write window of primitive objects, sensitive objects, or both

<table>
<thead>
<tr>
<th></th>
<th>Primitive obj</th>
<th>Sensitive obj</th>
<th>Both obj</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single write</td>
<td>11.11%</td>
<td>3.29%</td>
<td>1.26%</td>
<td>15.66%</td>
</tr>
<tr>
<td>Block write</td>
<td>0.68%</td>
<td>0.19%</td>
<td>0.13%</td>
<td>1.00%</td>
</tr>
<tr>
<td>Total</td>
<td>11.79%</td>
<td>3.48%</td>
<td>1.39%</td>
<td>16.66%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Primitive obj</th>
<th>Sensitive obj</th>
<th>Both obj</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single write (Opt.)</td>
<td>13.08%</td>
<td>2.86%</td>
<td>2.45%</td>
<td>18.39%</td>
</tr>
<tr>
<td>Block write (Opt.)</td>
<td>0.86%</td>
<td>1.72%</td>
<td>0.43%</td>
<td>3.01%</td>
</tr>
<tr>
<td>Total (Opt.)</td>
<td>13.94%</td>
<td>4.58%</td>
<td>2.88%</td>
<td>21.40%</td>
</tr>
</tbody>
</table>

JavaScript objects. We also considered the types of the write accesses. Table 4.1 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 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 4.2 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 4.4.5). This optimization reduced the average performance overhead from 5% to 2%, as discussed in Section 4.5.3. 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 4.2. 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
Table 4.3: Default memory access permission at run time

<table>
<thead>
<tr>
<th>Data</th>
<th>Permissions</th>
<th>SpiderMonkey 60.0.0 with NoJITsu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytecode</td>
<td>RW</td>
<td>R</td>
</tr>
<tr>
<td>Object tables</td>
<td>RW</td>
<td>R</td>
</tr>
<tr>
<td>JS Objects</td>
<td>RW</td>
<td>R</td>
</tr>
<tr>
<td>JIT IR</td>
<td>RW</td>
<td>R</td>
</tr>
<tr>
<td>JIT code</td>
<td>RX</td>
<td>X</td>
</tr>
<tr>
<td>JIT data</td>
<td>RX</td>
<td>R</td>
</tr>
</tbody>
</table>

the degree of hosting optimization according to the performance and security requirements of the system.

**Code-Injection Attacks**

While code-injection attacks are already mitigated to some extent by the existing deployed defense mechanisms [24], several advanced attacks aim at bypassing them, e.g., by injecting constants and exploiting unaligned instruction fetches [15, 9, 63]. 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 [25]. As a result, the current version of SpiderMonkey does not deploy constant blinding in the interpreter or the Baseline JIT compiler [91]. 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 4.3.1). This means that injected constants will no longer be mapped as executable at run time within NoJITsu.
**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.

**Bytecode Interpreter Attacks**

As described in detail in Section 4.2 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.

4.5.2 Coverage of Dynamic Object-Flow Analysis

As discussed in Section 4.4.3, 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.
4.5.3 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 [7] 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 4.9 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 4.9, 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.

4.6 Discussion

4.6.1 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 [46] and JavaScriptCore [6], 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 [2, 1].

**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 4.2.3). We found that in V8 and JavaScriptCore any types of JS objects remain writable, 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.

**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 [102].

4.6.2 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 [8], 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 [26, 75, 97]. 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 [10, 32, 18] in
support of that feature.

4.7 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 chapter, 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%.
Chapter 5

Related Work

Most of the existing work in this area focuses on code-injection attacks and defenses for JIT-based VMs. The security of pure bytecode interpreters has received little attention in the academic community.

5.0.1 Direct Code Injection

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 [87, 24]. Early attempts to address this issue included monitors that detected system calls originating from writable code regions [36]. However, as JIT compilers began to enforce strict W⊕X policies [24], 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. In Chapter 3, We enforce the equivalent level of security for bytecode by moving all bytecode strings to read-only memory.
Song et al. showed that direct code-injection attacks on the JIT cache were still possible by leveraging JavaScript worker threads [90]. 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 [66]. However, Microsoft recently announced shifting their focus and replacing Chakra with V8 as part of their Edge browser [12]. 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.

5.0.2 JIT Spraying

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) [15]. Since these constants appear as instruction operands in the JIT code cache, they can be executed as if they were valid instructions. This attack was initially mitigated using a defense called constant blinding [79]. Constant blinding masks embedded constants by XORing them with a random constant value before they are used. However, this defense has since been bypassed by Athanasakis et al. [9], who showed that JIT spraying is also possible with smaller constants and that applying constant blinding to smaller constants is prohibitively expensive. Similarly, Maisuradze et al. demonstrated a variant of the JIT spraying attack that uses carefully placed branch statements to inject useful code into the code cache [62]. This attack cannot be mitigated by constant blinding at all. As an alternative to constant blinding, Homescu et al. proposed to apply code randomization to JIT-compiled code [50]. With code randomization, it is still possible to spray the code cache with machine code that is embedded in constants, but the location of these constants becomes less predictable. Also, it is shown that integrating control-flow integrity into JIT code can mitigate the advanced JIT Spraying attacks [71]. NoJITSu strengthens these existing defenses by additionally
separating data constants from JIT code (see Figure 4.7). This enables us to enforce non-executable permissions for constants.

5.0.3 JIT Code Reuse

Snow et al. proposed to attack JIT engines through code-reuse attacks [88]. 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 [85]. Defenses against JIT-ROP included execute-only memory combined with randomization [32, 11, 10, 41], destructive code reads [94, 99], and cross-checking reads performed by JIT code [40]. However, some of these defenses were quickly bypassed [89, 62], 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.

5.0.4 Intermediate Data Structure Corruption

Theori proposed an attack that corrupts a temporary native code buffer in Microsoft’s JavaScript engine (Chakra) [95]. This temporary buffer is used to store machine code before the JIT compiler emits it into a non-writable memory region. Microsoft subsequently added checksum verification logic to the JavaScript engine to verify integrity of the temporary buffer. Similar to this attack, Frassetto et al. corrupt intermediate representation (IR) code which the JIT compiler temporarily produces from bytecode to apply code optimizations and to generate machine code [38]. To defend against this attack, Frassetto et al. proposed a defense called JITGuard, which moves the JIT compiler and its data into an Intel SGX
environment that is shielded from the application. JITGuard emits code to a secret region which is only known to the JIT compiler. Since the code is inaccessible to the attacker this also prevents code-reuse attacks targeting JIT’ed code. Our work bears similarity with these approaches in that we corrupt internal data structures of the VM to cause malicious code execution. Unlike the previous approaches, however, we corrupt bytecode which is considered more challenging to manipulate due to its restricted capabilities. Frassetto et al. mentioned in their discussion that corrupting bytecode is challenging and is out of scope [38]. In addition, since all of the previous defenses focus on mitigating attacks against the JIT compiler, they cannot prevent our bytecode interpreter attacks.

5.0.5 Dynamic Dispatch and Checksum Verification

Schuster et al. [82] and subsequent works [33, 58] presented whole-function code-reuse attacks that defeat strong CFI and code randomization defenses. These attacks exploit the dynamic dispatch mechanisms present in C++ and Objective-C resp. Cama et al. [20] 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 [82], and thus this attack would be prevented by existing defenses against vtable corruption or vtable reuse attacks [33, 104, 16]. In contrast, our attack deliberately overwrites the internal data of an 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 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 CPython, lua, and SpiderMonkey and leverages the fact that internal data
can be overwritten safely without restoring them. Our defense in Chapter 3 is also inspired by existing work [64, 58, 95] that uses hash checksums to verify the integrity of sensitive pointers, runtime metadata, and JIT code caches, respectively.

### 5.0.6 Memory Protection Key

There are previous works to provide secure isolation interfaces using MPK. Libmpk [75] provides a secure software abstraction to improve security and resolve technical challenges in using MPK. ERIM [97] 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) [54]. These approaches are orthogonal to our approach in Chapter 4 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.
Chapter 6

Conclusion

It is generally accepted that memory corruption attacks are hard to defend against. Moreover, the scripting environment of dynamically-typed languages widens the attack surface for the memory corruption exploitations. Although various techniques have been developed to attack dynamically-typed languages, their primary focus is on JIT components.

Our work in this dissertation explores a new attack vector to corrupt the interpreters and highlights the pitfalls of language-level sandboxing. Specifically, we corrupt bytecode interpreters and successfully execute arbitrary system calls. Our attacks bypass existing defense techniques that focus on components without hardening the interpreter.

Therefore, we propose defense mechanisms to completely lock down the script engines. Our defenses are comprehensive in that they protect the entire script engine including the JIT components and the bytecode interpreter. To implement our defenses, we utilize Intel’s Memory Protection Key (MPK) to achieve the fine-grained memory access control and low overhead of less than 5%.

As script engines evolve with new technologies, new security holes can be created. Therefore,
we need to pay attention to potential threats and anticipate the risks. Our work highlights new threats in dynamically-typed languages and suggests practical solutions to them. We hope that this dissertation sparks further exploration of the dynamically-typed languages in the pursuit of a safe scripting environment.
Bibliography


95


