SAARLAND UNIVERSITY

Faculty MI – Mathematics and Computer Science
CISPA Helmholtz Center for Information Security
Research Group of Dr. Michael Schwarz

Bachelorthesis

Power-ups for Chromium: Facilitate Side-Channel Research

Jonathan Busch

First Reviewer: Dr. Michael Schwarz
Second Reviewer: Dr. Cristian-Alexandru Staicu

Saarbrücken, February 8, 2022

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 08.02.2022 [illegible]

(Datum/Date) (Unterschrift/Signature)

Abstract

In recent years, side-channel based attacks have gained more and more importance. Instead of exploiting bugs, these novel attacks use side effects caused by software or hardware to infer secret information. Low-level programming languages such as C or C++ are predominantly used to develop side-channel exploits, since they allow for controlling and measuring the microarchitectural state. However, access to these low-level functions is not available for every programming language. For instance, JavaScript lacks some crucial functions used in side-channel attacks. Instead, one must use highly laborious workarounds to realize the same attacks. This makes testing, whether a newly found side-channel attack works in JavaScript, a time-consuming task. In this work, we add low-level functions to Chromium's JavaScript engine in order to facilitate this process. This allows researchers to build Proof-of-Concept scripts to check whether a new side-channel attack is feasible in JavaScript, to avoid developing expensive workarounds. Additionally, we provide Proof-of-Concepts of well-known side-channel-based attacks such as Flush+Reload and Spectre utilizing our extended version of Chromium. We evaluate the reliability of the added power-ups of our Proof-of-Concepts by running them multiple times with randomly generated input strings. The results show high success rates for all Proof-of-Concepts, with results greater than 95.95%.

Acknowledgements

I wish to express my gratitude to my advisor Dr. Michael Schwarz for guiding me through this thesis. Thank you for always taking the time to patiently answer all the questions I had; for your constructive and helpful feedback, and your important and interesting input. Thank you for your guidance and support.

I would also like to thank Dr. Cristian-Alexandru Staicu for agreeing to review my thesis as second reviewer.

Finally, I would like to say thank you to my closest friends and family, for listening to me talk about nothing but the thesis in the last months, coping with my stressful phases, hours of proof-reading and always supporting me in the process.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

Every 11 seconds, a business is expected to be attacked by ransomware [1]. These attacks pose a great risk to businesses and individuals. The average total cost of a data breach for businesses in 2021 was $4.24m, which is an increase of 9.8% compared to the previous year [2]. This does not only affect the business's reputation and finances, but threatens the integrity of customer data. IT-Security research is trying to identify vulnerabilities in systems before criminals do, and develop mitigations. The approach of attack prevention becomes progressively more important, since in 2021, it takes a business 212 days on average to identify and contain a data breach [2].

One subfield of IT-Security research is concerned with *microarchitectural attacks*, such as *side-channel attacks (SCA)*. These attacks focus on the microarchitectural properties and optimizations of a system and use side effects to uncover secret information. Such attacks require low-level functions, e.g., a high precision timer, which is easily accessible using low-level programming languages like C or $C++$. However, these *low-level functions* are not always directly available in a browser environment. For example, even though there exists a function to retrieve a high-resolution timestamp in JavaScript, browsers typically round the returned value to introduce a certain amount of inaccuracy in order to mitigate attacks that utilize timing differences [3]. While in the case of high-precision timers, the substitution of the low-level functionality is straightforward, other low-level functions require laborious workarounds to work in a browser environment. If a new side-channel attack is discovered using a low-level programming language, testing whether this attack works in a browser is a non-trivial task.

In this work, we want to facilitate this process by adding low-level functions and code snippets we call *power-ups* to Chromium, an open-source browser project [4].

These power-ups are available to researchers through API-style calls in a newly added package called *Sca*, and can be used to implement side-channel-based attacks in a browser environment using the same low-level functionality as in low-level programming languages. Additionally, we provide JavaScript Proof-of-Concepts based on our version of Chromium for side-channel-based attacks, such as Flush+Reload [5] and Meltdown [6], to show that the added power-ups work as intended.

Finally, we evaluate the reliability of the added power-ups of our Proof-of-Concepts by running them multiple times with randomly generated input strings. The results show high success rates for all Proof-of-Concepts, with results greater than 95.95%.

The presented work is available to the public in a GitHub repository[1] including patch files for Chromium version *99.0.4832.0* and ready-to-use installers for Windows and Linux (Debian-based systems).

[1] https://github.com/Th3J0nny/Power-ups-for-Chromium

2 Background

2.1 Side-Channel Attacks

There exist numerous factors that can cause vulnerabilities in computer systems. For instance, a flaw in the design of a cryptographic algorithm can break the encryption and potentially leak secret information.

In recent years, *side-channel attacks* (SCA) have gained more and more importance. These attacks rely solely on meta-data to infer secret information. The meta-data used by side-channel attacks are caused by various properties, which can be categorized into physical and logical properties [7] (Table 2.1). Additionally, side-channel attacks have further properties that include whether an attacker is active or passive during the attack and whether the attacker has to be present locally or controls the attack from a remote location.

Physical	**Logical**
Power analysis	Differential computation analysis
Temperature variation monitoring	Network traffic analysis
Wi-Fi signal monitoring	Data-usage statistics
...	...

TABLE 2.1: A sample of side-channel attacks categorized by whether they exploit physical or logical properties.

Hardware-based side-channel attacks, such as the measurement of power-consumption of a system, require physical access and equipment, for instance, an oscilloscope. This is not required for software-based side-channels, such as Flush+Reload (see Section 2.1.1). However, due to the complexity of modern computer systems, there are many reasons for side channels to be unreliable, including noise from concurrent operations. Most side-channel attacks target cryptographic functions, passwords, and other user

data. The first theoretic description of such a software side-channel attack in 1996 uses timing differences to target cryptographic systems such as Diffie-Hellman or RSA [8]. The timing differences described in the attack are often caused by optimizations that are dependent on secret information. One of these optimizations are caches, which are used to speed up repeated access to data by storing it in a fast, temporary storage. Caches exist, e.g., in browsers and on CPUs. However, the speed-up introduced by caches causes measurable timing differences between cached and uncached data accesses. In a browser scenario, an eavesdropper could use the timing information they acquire by observing (possibly encrypted) network traffic, to determine whether the victim recently visited a website or not.
The same principles apply to CPU caches. By using high-precision timers, it is possible to measure the timing differences between *cache hits* (something was recently accessed and is therefore present in the cache) and *cache misses* (something is not in the cache and therefore was probably not accessed recently).

Cache side-channel attacks are found to be a great threat, even in allegedly secure execution environments, such as Intel SGX enclaves. In 2017, it was shown that cache-timing attacks on AES are possible, even when the code was running in an Intel SGX enclave [9]. Mitigating these attacks is a non-trivial task, especially when considering the trade-off between speed-ups through optimizations, such as caches, and the security against these side-channel attacks. This is reflected by statements regarding the prevention of side-channel attacks by manufacturers like Intel: "Intel SGX does not provide explicit protection from side-channel attacks. It is the enclave developer's responsibility to address side-channel attack concerns." [10]

2.1.1 Flush+Reload

In 2014, Yarom et al. presented a novel side-channel cache attack called *Flush+Reload* [5], which monitors timing differences of memory accesses. The attack scenario is the following: There exists an attacker process and a victim process that share some memory. For example, in the case when shared libraries are used, e.g., system libraries like *libc* [11]. Shared libraries have the same physical address in memory for all processes and are thus at the same location in the cache as well.

Additionally, a high performance timer is required to measure the access times and thereby distinguish cache hits and misses. On current Intel CPUs, access to data in the cache takes around 4-31 CPU cycles [12]. However, access to data in DRAM memory requires more than 120 cycles. Depending on the clock frequency of the CPU, a timer in the nanosecond range is required to differentiate between cache hits and misses. This precise measurement is achievable by accessing the CPUs *Time Stamp Counter (TSC)* using the x86 *rdtsc* instruction [13] right before and after a data access. This instruction is available on all modern x86 processors.
The time measurement is repeated in a loop until a cache hit is found. However, with every iteration undesired data is cached, which would make subsequent measurements ineffective. The x86 *clflush* instruction [13] removes the specified data from the cache, and therefore must be called after each timing measurement in an iteration.

By using the Flush+Reload attack it is possible for an attacker to infer secret information from vulnerable processes. For instance, the private key of the RSA cryptosystem can be leaked if it uses the square-and-multiply algorithm to efficiently compute the modular power of large natural numbers [12]. Algorithm 1 shows a vulnerable implementation of the square-and-multiply algorithm which takes three arguments and returns res:

- m: A message in binary representation
- exp: Secret exponent
- n: Modulo
- res: $m^{exp} \mod n$

Depending on the current last bit of the secret exponent, the control flow of the code changes in Line 3: If the the bit equals 1, the $multiply()$ function is called, otherwise the execution immediately continues in Line 6. Given that the functions $multiply()$ and $square()$ are shared in memory, e.g., as part of a shared library, an attacker can

Algorithm 1 Vulnerable square-and-multiply algorithm

```
1: $res \leftarrow 1$
2: while $exp > 0$ do
3:     if $exp \mod 2 == 1$ then
4:         $res \leftarrow multiply(res, x)$
5:     end if
6:     $res \leftarrow square(res)$
7:     $exp \leftarrow exp/2$
8:     $res \leftarrow res \mod n$
9:     $m \leftarrow m \mod n$
10: end while
11: return $res$
```

use Flush+Reload to determine whether these functions were called or not. This information can be used to derive the secret exponent, since for every call to $multiply()$ an attacker can note 1 as the corresponding bit in the key, and if $square()$ is called twice, 0 respectively.

2.2 Transient Execution Attacks

Side-channel attacks use unintended leakage of data, e.g., through meta-data to derive secret information. *Transient execution attacks* however, intentionally leak data caused by microarchitectural side-effects through so-called *covert channels*. These covert channels are used for the communication of two processes, which are both attacker controlled, and where data exchange is not intended or not allowed, e.g., across VMs. Transient execution attacks leak side-effects caused by CPU optimizations such as *out-of-order execution*. The CPU can rearrange instructions and execute them out of order to make efficient usage of multiple execution units, i.e., instructions are executed *transiently*. This can be useful in situations when a computationally heavy instruction, that decides the control flow, blocks an execution unit for the execution of a task, for instance in an if-statement. Another (free) execution unit, can now execute subsequent instructions speculatively in a so-called *transient execution*.

2.2.1 Spectre

In Spectre attacks, a mispredicted control or data flow is used to leak arbitrary data from the application context. That means, that these attacks can bypass software based access policies.

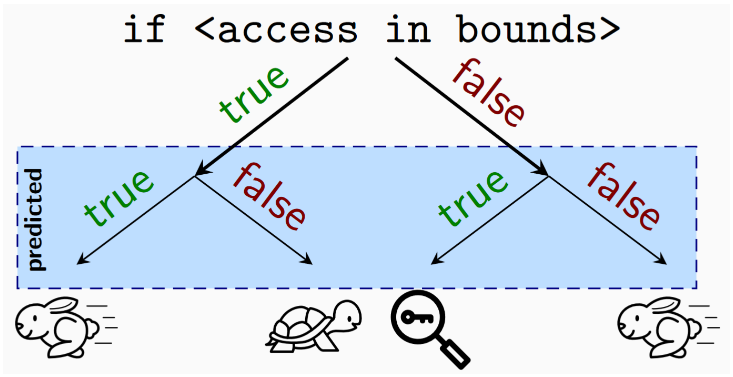

FIGURE 2.1: Speed-up of branch prediction for array in-bounds check [14]

Figure 2.1 shows the possible speed-ups of speculative execution for an array in-bounds check. If the prediction turns out to be correct after the computation of the if-statement finishes, some subsequent instructions are already executed and a speed-up is therefore achieved. Else, the instructions that have been executed out-of-bounds, are discarded and the correct instructions are executed as normal. However, if the in-bounds check results in $false$, but the prediction was $true$, it is possible, that certain memory was accessed without a permission check and with no exception raised. While the speculatively-executed instructions are discarded, there are microarchitectural side effects that are observable by an attacker. A side-channel attack can then be used to leak the data that has been accessed during the speculative execution.

There exist various types of predictors in a CPU that anticipate whether a certain branch is taken or not. In order to exploit these predictors for a transient execution attack, an attacker has to mistrain them first. In the following, we briefly describe branch predictors and mistraining strategies as described by Canella et al. [15].

Predictors

The predictor on conditional branches, such as if-statements, is the **Pattern History Table (PHT)**. It stores previous outcomes of branches in a table and uses the entries to predict the outcome of future branches. As branches often occur in loops, the accuracy of the predictions is very high.
Another type of predictor is the **Branch Target Buffer (BTB)**. It predicts the target location of indirect calls/jumps. When functions are, for instance, called from a function pointer, the target of the branch is stored in a register or the memory. The *indirect call/jump* target addresses are then predicted by the BTB based on the last branch taken.
Furthermore, the **Return Stack Buffer (RSB)** stores the caller address of a function. Since the return address has to be loaded from the memory stack, the CPU speculatively returns to the last added address on the RSB.
In addition to that, **Memory disambiguation / Store To Load (STL)** is used when a load is executed out-of-order and it speculates, whether a dependent store has been committed before or not.

Mistraining Strategies

In order for the aforementioned predictors to be utilized in attacks, they have to be mistrained. For example, mistraining can be the repeated execution of an if-statement that results in $true$. Based on that history, a predictor might assume subsequent executions to be true as well, and speculatively execute the respective instructions, even though the actual result is later found to be $false$.

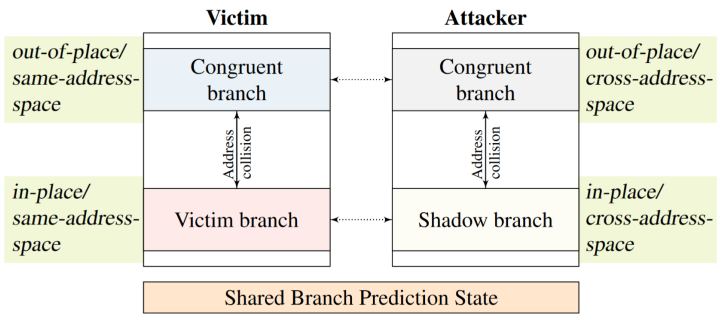
Victim
Attacker
out-of-place/ same-address-space
Congruent branch
Congruent branch
out-of-place/ cross-address-space
Address collision
Address collision
in-place/ same-address-space
Victim branch
Shadow branch
in-place/ cross-address-space
Shared Branch Prediction State

FIGURE 2.2: Overview of the predictor mistraining strategies for the Spectre attack [15]

The categorization of such mistraining strategies depends on two factors as shown in Figure 2.2. First, the mistraining can be executed in the same address space as the victim process (*same-address-space*), or in the address space of the attacker process (*cross-address-space*). Second, the victim branch itself can be used for mistraining itself (*in-place*), or an attacker can use a different branch to mistrain the victim branch which has a congruent virtual address to the victim branch. Note that for attacks on memory disambiguation, only *same-address-space in-place* mistraining is possible, since this prediction is not history-based.

The naming of Spectre attacks is based on the targeted predictor and mistraining strategy. For instance, *Spectre-PHT-SA-IP* targets the Pattern History Table (PHT) using a same-address-space (SA) and in-place (IP) mistraining strategy.

Example of Spectre-PHT

```
if (x < array1_size) {
  y = array2[array1[x] * 4096];
}
```

LISTING 2.1: Vulnerable conditional branch for Spectre-PHT attack [14]

All Spectre-PHT variants (also known as Spectre v1) target conditional branches, such as the one shown in Listing 2.1. First, the code checks whether x is in-bounds of $array1$. If that is true, the value of the base-address of $array1$ with offset x is used to determine the offset for the access of $array2$. We assume the following attack scenario: The value x is attacker controlled and $array1\_size$ and $array2$ are *not* in the cache. An attacker can now mistrain the predictor in a way that it assumes subsequent values of x to be in-bounds. If the value of x is now chosen *out-of-bounds*, it is possible, that the next line is speculatively executed. The fact that $array1\_size$ results in a cache miss makes this more likely to happen, because it takes more time to evaluate the condition. In this out-of-order execution, the computation of the $array1$ base address with the offset x results in an out-of-bounds memory access inside the victims process memory space. To make the covert channel more reliable, the value of this address is then multiplied

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

LISTING 2.2: Core of a Meltdown attack [6]

with 4096 and used as offset for the access of $array2$. This maps each possible value to another cache line and avoids the prefetcher from loading additional adjacent memory locations into the cache [6]. In this case, 4096 is chosen for memory pages with a size of 4096 bytes. For other memory page sizes this value is changed accordingly. In the meantime, the condition check could have been evaluated and the falsely out-of-bounds executed code is discarded. However, the access to $array2$ has loaded a value into the cache, since $array2$ was uncached. This value's location is dependent on the value at the aforementioned arbitrary memory address ($array1$ base + x). An attacker can now use cache attacks like Flush+Reload to determine, which index of $array2$ is now in the cache and derive the possibly secret value.

2.2.2 Meltdown

Meltdown attacks aim to *melt down* the security borders of the user space and the kernel space. As opposed to the usage of mistrained predictors triggering an out-of-order execution in Spectre attacks, Meltdown attacks rely on out-of-order executions that follow CPU exceptions. When such an exception is raised, e.g., after a *page fault*, the CPU has to retire the faulting instruction first. During this time, it is possible, that some instructions have already been executed out-of-order. As in the Spectre attacks, these instructions are discarded afterwards, but again leave microarchitectural traces that can be leaked through a covert channel. Listing 2.2 shows the core assembly code for a Meltdown attack as presented by Lipp et al. [6].

In Line 4, a byte from the kernel address stored in RCX is loaded into the lower

8-bits of the RAX register. This results in a page fault, because of the failing permission check to access kernel memory, and therefore raises an exception. However, until the exception is raised and the instruction pipeline is flushed, it is possible that the instructions in Line 5-7 are executed out-of-order. In Line 5 the contents of RAX (byte at the target kernel address) is multiplied by 4096 for the same reasons as described in the example of Spectre-PHT. If the illegal MOV instruction in Line 4 fails, the authors (Lipp et al.) [6] observed, that their implementation often returns 0. For that reason there exists a check in Line 6, whether the value in RAX is 0. If that is the case, the code jumps to Line 4 and attempts the out-of-order instruction sequence again. Else, the value in RAX (value at kernel address multiplied by 4096) is added to the address of the probe array in RBX and thereby loaded into the cache. A cache side-channel attack like Flush+Reload can now be used as a covert channel to leak the value from the cache.

Since a program is terminated when an exception occurs, the Meltdown attack can be further optimized by suppressing exceptions. This can be achieved by using CPU exception handling, Intel TSX, or by hiding the faulting instruction in another transient execution [15]. Since Meltdown allows to cross the kernel space boundary, it is possible for an attacker to read all memory contents including the entire kernel.

2.2.3 Classification of Transient Execution Attacks

The following classification of transient execution attacks is based on the work of Canella et al. [15].

The first step is to determine the cause of a transient execution. In the case of Spectre-type attacks, the root cause is the prediction of a control or data flow. On the other hand, for Meltdown-type attacks, the transient execution happens due to an exception. The type of exception further determines the type of Meltdown attack based on Intel's classification of exceptions [13], i.e., faults, traps, or aborts. In the case of page faults, the Meltdown type attacks are further categorized based on the page-table

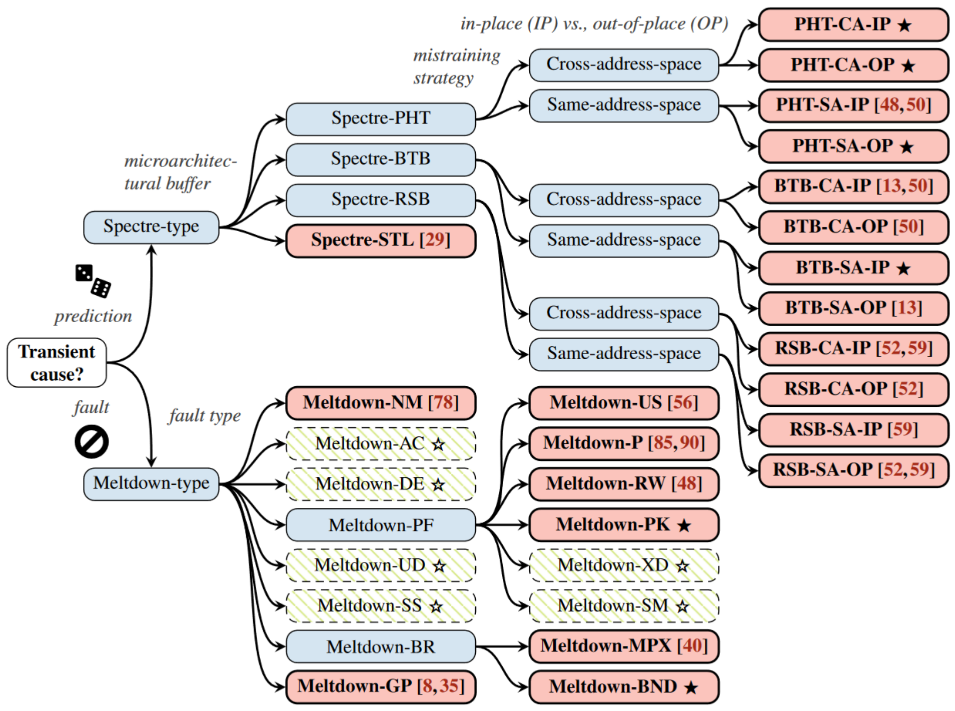

FIGURE 2.3: Classification of transient execution attacks [15]

entry protection bits and the storage locations that can be reached by the attack. However, for Spectre-type attacks, the first layer of classification is the type of predictor that is targeted (see Predictors). For all but Spectre-SLT the next two layers are split based on the mistraining strategy, i.e. whether the mistraining happens in the same-address-space or cross-address-space and whether it mistrained function is in-place or out-of-place (see Mistraining Strategies).

3 Implementation

3.1 Chromium

Chromium [4] is an open-source browser project under a permissive BSD license launched by The Chromium Projects in 2008 [16]. Many browsers, such as Chrome, Microsoft Edge and Opera, use the Chromium codebase [16][17][18], thus, research results using this work are applicable to a multitude of users. The majority of the Chromium source code is written in C++.

Blink is the rendering engine used by Chromium-based browsers and implements everything that renders content inside a browser tab, including specs of the web platform, and the embed of *V8*, the engine used to run JavaScript code [19].

In this work, we include an additional module called *sca* inside the *blink/renderer/* directory. While all added files for this work are located in the *modules/sca/* sub-directory, there are two additional files in the *bindings/* sub-directory that need to be modified in order to include the added files in the build (Appendix A).

In order to expose functions in JavaScript (as JavaScript objects), Blink uses Web Interfaces that are generally specified in the *Web Interface Definition Language (Web IDL)* [20]. The interfaces are specified in IDL files (.idl) and generate C++ bindings that the V8 JavaScript virtual machine uses to call the corresponding code in Blink. In order to implement a new Web IDL interface in Blink, we take the following steps, using the example *sca*:

- Add an IDL file as interface: *sca.idl*.
- Implement the defined functions and attributes in a C++ and header file: *sca.cc* and *sca.h*.

- Create two files (C++ and header file) to add the resulting JavaScript object to the JavaScript *Window* object: *dom_window_sca.cc* and *dom_window_sca.h*.
- Modify the files *generated_in_modules.gni* and *idl_in_modules.gni* and add the C++/header and IDL files respectively.

In this work, we implement two interfaces:

Sca contains various low-level functions and code snippets, and full C++ Proof-of-Concepts of the Spectre [14] and Meltdown [6] attacks.

Meltdown contains API-style calls to the library *libkdump*, which is part of the Meltdown Proof-of-Concept [21].

To build our version of Chromium including the power-ups, we use the following build arguments:

```
# Linux:
is_component_build = false
enable_nacl = false
blink_symbol_level = 0
symbol_level = 0
is_debug = false
target_cpu = "x64"
enable_linux_installer = true
# Windows:
is_component_build = false
enable_nacl = false
blink_symbol_level = 0
is_debug = false
target_cpu = "x64"
```

The Chromium version used in this work is based on:

- **Version**: 99.0.4832.0
- **Commit**: 9200526668cd3ea7efd0ce49a81cc376d9326113

3.2 Power-ups

In this work, we add low-level functions and code snippets to JavaScript in Chromium called *power-ups*. These power-ups facilitate research in the field of microarchitectural attacks. In the following, we provide a description of the added power-ups categorized into their respective, but not exclusive, area of usage. A full documentation of all power-ups is provided in Appendix C.

3.2.1 Memory

In JavaScript, it is not possible to retrieve the address of JavaScript objects programmatically. However, addresses of data in memory are mandatory for some attacks, e.g, when data needs to be flushed from the cache.

The power-up **bufferAddress** takes an ArrayBuffer JavaScript object as argument and returns its address as *unsigned long long*. It is desirable to retrieve the address of any JavaScript object. However, Chromium's JavaScript engine V8 is highly optimized, e.g., it uses a garbage collector, called *Oilpan*, for managing C++ memory [22]. JavaScript objects and the C++ object graph are heavily tangled and are therefore treated as one heap from the garbage collector. The garbage collector marks all reachable object it discovers, and then frees other dead objects, i.e., objects that cannot be reached in any possible execution and are therefore unused. The garbage collector not only frees dead objects, it also moves objects that survived the garbage collection [23]. Therefore, a power-up that retrieves the address of any JavaScript object is not feasible for this work, since pointers to these objects can change at any time. However, the *ArrayBuffer* object represents a generic, fixed-length raw binary data buffer. This buffer can be manipulated by, e.g., typed array objects [24], and has a fixed address in memory.

Additionally, the power-up **memmap** provides an API-style call to the Linux function *mmap* [25], which allows low-level allocation of memory and returns a memory address as well. If the mapping fails, the power-up returns 0. Values for the flag arguments *prot* and *flags* can be accessed using *Sca::FLAGNAME()*. Note: This power-up is only available on Linux systems. On other systems, the power-up returns 0.

The power-up **getPhysicalAddress** takes a virtual memory address as argument and, on success, returns the corresponding physical memory address. If run on Linux, the power-up maps the address using the pagemap file in */proc/self/pagemap*. However, this operation requires root privileges. Therefore, on failure, or if executed on Windows systems, the power-up uses the PTEditor kernel module [1] and returns the mapped physical address as *unsigned long long*. On failure, or if the the PTEditor kernel module is not installed/loaded, the power-up returns 0. Note: Chromium uses sandboxing which restricts access to some parts of the file system [26]. Therefore, if *getPhysicalAddress* is used with the PTEditor kernel module, Chromium needs to be started with the *--no-sandbox* flag.

3.2.2 Cache

Attacks, such as Flush+Reload [5] or Spectre, utilize the cache as side channel or covert channel respectively. Therefore, low-level functions, such as flush or rdtsc to change the cache state and measure timing differences, are mandatory. However, these functions are not directly available in JavaScript.

The power-up **rdtsc** uses the rdtsc instruction and returns the processors current time stamp, i.e., the number of cycles since the last reset [13], as a DOMHighResTimeStamp (double) [27]. This timestamp is used for timing attacks that require the accurate measurement of timing differences, e.g., the time it takes to access data in memory. The JavaScript function *performance.now()* returns a timestamp that is accurate to 5 microseconds, which is not sufficient to measure such timing differences [3][27]. Alternatively, it is possible to use a WebWorker [28] as counting thread. However, the provided power-up *rdtsc* saves time and keeps code clean.

[1] https://github.com/misc0110/PTEditor

The power-up **flush** takes an address as argument and flushes the data at this address from the cache. This power-up is used, e.g., in the Flush+Reload attack to remove data from the cache and observe, whether it was accessed by another process or not. Calling the flush instruction directly avoids the process of building laborious workarounds such as eviction sets as described by Oren et al. [29].

The power-up **maccess** takes an address as argument and accesses this address in memory. This power-up is used, e.g., to implicitly load data into the cache. Alternatively, it is possible to access the data, e.g., by moving it into a dummy variable.

Using the previously described power-ups, it is possible to conveniently measure the time it takes to access memory, given a memory address, and optionally flush the data from the cache afterwards:

```
let time = sca.rdtsc();
sca.maccess(address);
let delta = sca.rdtsc() - time;
sca.flush(address); // optional
```

However, according to our measurements, calling the above code introduces an overhead of about 10,000 cycles in Chromium (see Section 5). Therefore, we add the power-ups **reload** and **flushandreload**, which execute the code in Chromium using C++, and therefore avoid the overhead introduced by Chromium. Both power-ups take an address as argument and return the time it takes to access the memory at the given address. First, they use the *rdtsc* instruction to get the current timestamp of the processor. Second, the power-ups access the data at the given address using *maccess*. Next, they retrieve another timestamp using *rdtsc* and calculate the difference to the first timestamp. Finally, only the power-up **flushandreload** calls the *flush* instruction to remove the data at the given address from the cache.

Analogously, the power-up **flushdelta** is used to measure the time it takes to flush memory from the cache. This is used in attacks such as Flush+Flush [30]. First, the power-up uses the *rdtsc* instruction to get the current timestamp of the processor. Next, it flushes the given address from the cache using the *flush* instruction. Finally, the power-up retrieves another timestamp using *rdtsc* and calculates the difference to the first timestamp:

```
unsigned long time = rdtsc();
flush(address);
unsigned long delta = rdtsc() - time;
return delta;
```

Timing attacks require a threshold to distinguish cache hits from cache misses. Due to the previously described overhead, we implement the threshold calculation as the power-up **detectThreshold**, which calculates and proposes a threshold to distinguish between cache hits and cache misses. The value is calculated by accessing dummy data in the cache 1 million times without flushing (cache hits) and 1 million times with flushing (cache misses) it in between iterations. The threshold is then calculated by using the respective average values in the following statement [2]:

```
(avg_miss_time + avg_hit_time * 2) / 3
```

An analog implementation of this power-up in JavaScript is shown in Appendix B using the power-ups *flush*, *maccess*, *flushandreload*, and *reload*.

3.2.3 Transient Execution

We implement API-style power-ups that provide access to the library *libkdump* [21], which can be used for Meltdown attacks. First, the library can be initialized automatically, by calling the power-up **initAutoconfig**. Alternatively, the power-up **init** allows to add configuration parameters as arguments to initialize the library. Both power-ups return a string representation of the resulting configuration. After initialization, the power-up **read** takes an address as argument, leaks the corresponding byte in memory using Meltdown, based on the current configuration, and returns the leaked value as *int*. If the library has not been initialized when calling the power-up **read**, it initializes the library using the automatic configuration as seen in *initAutoconfig*. Finally,

[2]This is similar to existing implementations, e.g., in *libkdump* [21]

the power-up **cleanup** needs to be called after an attack is finished. Note: The above described API-style power-ups to the library *libkdump* are only available on Linux systems. On other systems, these power-ups have no effect.

Additionally, we include the power-up **meltdown**, which executes the code of the file *test.c*, which is included in the repository of the Meltdown Proof-of-Concept [31]. The power-up takes a randomly selected string from a list and uses the Meltdown bug to leak this string.

The power-up **spectre** executes the Proof-of-Concept code provided by Kocher et al. [14] and leaks the string provided as argument using the Spectre attack. On success, the provided string is returned, an undefined string otherwise. If no argument is given, the power-up uses the string “The Magic Words are Squeamish Ossifrage.” and return a string describing the result.

4 Evaluation

4.1 Proof-of-Concepts

In order to evaluate the proper functionality of the power-ups described in Section 3.2, we implement three different Proof-of-Concepts (PoCs) as HTML pages using JavaScript, that run on our version of Chromium. We evaluate these PoCs on a Linux (Debian 5.14.0-9parrot1-amd64, Parrot OS 5.0) laptop using an Intel i7-7700HQ CPU with 2.8 - 3.8GHz (Machine 1) and on a Windows 10 Pro PC (Version: 21H2, Build: 19044.1466) using an AMD Ryzen 7 5800X CPU with 3.8GHz (Machine 2).

We evaluate the correctness of our PoCs by running an attack multiple times using random input letters of a given length, and compare the leaked value against the input. Table 4.1 shows the mean success rate of strings with length 1 and length 100 over 100,000 and 1,000 iterations respectively.

The PoCs can be found in this repository: `https://github.com/Th3J0nny/Power-ups-for-Chromium`

4.1.1 Flush+Reload

The Flush+Reload PoC is based on the work of Yarom et al. [5]. The PoC encodes a string into an array (*secret*), accesses a byte of the array (load it into the cache) and then leaks the corresponding value using Flush+Reload.

When the PoC website is loaded, the *secret*, typed array (*Uint8Array*) with byte length 4096 is created. This implies, the maximum string length that can be leaked in this PoC is 4096. Next, a new *ArrayBuffer* called *array* is created and the first 256 indices are filled with additional Uint8Arrays of length 4096. Since most systems use memory

PoC	Machine	Length	Iterations	Success rate
Flush+Reload	2	1	100,000	95.95% (SD = 19.61)
Flush+Reload	2	100	1,000	99.96% (SD = 0.21)
Spectre	1	1	100,000	100%
Spectre	1	100	1,000	99.98% (SD = 0.1)
Spectre	2	1	100,000	100%
Spectre	2	100	1,000	100%
Meltdown	1	1	100,000	100%
Meltdown	1	100	1,000	100%

TABLE 4.1: Evaluation results of our PoCs. The table shows the mean success rate of strings with length 1 and length 100 over 100,000 and 1,000 iterations respectively.

pages of size 4096 bytes, this ensures, that each *Uint8Array* within *array* is mapped to one cache line.

This PoC includes two possibilities of threshold calculation. The first possibility is executed when clicking on the button "Calculate Threshold" (function *calcThreshold()*). This function calls the power-up detectThreshold and returns its value. The second button, "Calculate Threshold JS", executes the function *calcThresholdJS* which uses the power-ups bufferAddress, flush, flushandreload, maccess and reload to calculate the threshold in JavaScript as seen in Appendix B.

When the "Start Attack" button is pressed, the function *startAttack* loads the text string from the input field of the website into the *secret* array. If no string value was entered into the input field, the function uses a default string ("Leak me!"). The function now iterates over all indices *i* of the *secret* array. Next, *array* is flushed from the cache and then accessed at the position *i* of the *secret* array. Finally, the function iterates over *array* (*31 < j < 127*) and measures the access times using the power-up reload. If the measured time is below the threshold (*CACHE_HIT_THRESHOLD*), the function returns the index *j*, which is the leaked letter of the *secret* array at index *i*. If no measurement for a letter returns a value below the threshold, the character "ø" is added as placeholder.

The results in Table 4.1 show a mean success rate of 95.95% (SD = 19.61) for string length 1 with 100,000 iterations and a mean success rate of 99.96% (SD = 0.21) for string length 100 with 1,000 iterations on Machine 2. Note: We were not able to successfully execute the Flush+Reload PoC on Machine 1 (see Section 5).

4.1.2 Spectre

The Spectre PoC is based on the work of Kocher et al. and the provided PoC [14]. We use Flush+Reload as covert channel, similar to the Flush+Reload PoC described above. We test this covert channel, by adding one line in the code that accesses the desired address in memory and is then leaked in the Flush+Reload part of the attack implementation. We encode all data that is used by our power-ups (e.g., flush) into data structures using *ArrayBuffers* in order to retrieve their memory address. Additionally, we check that the *secret* array is subsequent to *array1*, i.e., we have a positive offset, as it facilitates the debugging process. In general, this PoC is analogous to the PoC described by Kocher et al. [14].

In our current Spectre PoC implementation in JavaScript, we were not able to successfully show a working Spectre attack. For more information, see Section 5. However, the full PoC provided by the authors of the Spectre attack [14] and implemented as power-up spectre can be run by pressing the "Run paper PoC" button and is used in this evaluation.

The results in Table 4.1 show a 100% success rate for both evaluation runs on Machine 2, and for string length of 1 with 100,000 iterations on Machine 1. The evaluation of string with length 100 and 1,000 iterations shows a mean success rate of 99.98% (SD = 0.1) on Machine 1.

4.1.3 Meltdown

The Meltdown PoC is based on the work of Lipp et al. [6], using the API-style power-ups based on the *libkdump* library [21]. The button "Start attack using libkdump API" calls a function that uses the power-ups initAutoconfig, bufferAddress, read, and cleanup to leak the string in the variable *EXPECTED*, which is encoded in an *Uint8Array* called *secret*. The button "Start full attack" calls a function that is based on the Meltdown PoC used by the power-up meltdown [31]. We only evaluate the function that uses the

API-style power-ups to the library *libkdump*, since this allows for more control over its configuration.

The results in Table 4.1 show a 100% success rate on Machine 1 of the Meltdown PoC.

Note: Since the CPU in Machine 2 is not vulnerable to this type of Meltdown attack, it is only checked on Machine 1.

5 Discussion

The results in Section 4 show that our power-ups can be used to implement known microarchitectural attacks, such as Flush+Reload [5], Spectre [14], and Meltdown [6] in a way that is close to the Proof-of-Concepts (PoC) as presented by the authors of the original attacks.

In our Flush+Reload PoC, we use the power-up *reload* to measure the access time to data in memory. This power-up executes the instructions *rdtsc* and *maccess* in one function call in C++ code in Chromium. We also tested a different version of the Flush+Reload PoC, in which we used the power-ups *rdtsc* and *maccess* individually in JavaScript for this task. However, we were not able to clearly distinguish cache hits and cache misses using this method. We suspect that this is because JavaScript calls to these power-up functions introduce overhead, which is critical, e.g., when executing timing critical functions, such as *rdtsc*. We use the power-ups *rdtsc* and *flush* to measure the time it takes to access data when it is cached (cache hit) and data when it is not cached (cache miss) in Chromium using JavaScript. Although it is possible to distinguish cache hits and cache misses as shown in the histogram in Figure 5.1, we notice, that the mean value of the hits is 10172.31 (SD=146.26) [1].

This value is significantly higher than the value, when the power-ups *reload* and *flushandreload* are used, i.e., when the overhead is avoided, as seen in the histogram in Figure 5.2. The mean value of cache hits in this case is 146 (SD=14.42) [2]. Additionally, we evaluate our Flush+Reload PoC using the power-ups *flush*, *maccess*, and *rdtsc* by executing the attack 10,000 times using a single random letter on Machine 2, and

[1]Data points that are greater or less than 3 standard deviations from the mean, are identified as outliers (N=12239) and discarded from the data (total number of data points: 1048576).

[2]Data points that are greater or less than 3 standard deviations from the mean, are identified as outliers (N=108) and discarded from the data (total number of data points: 1048576).

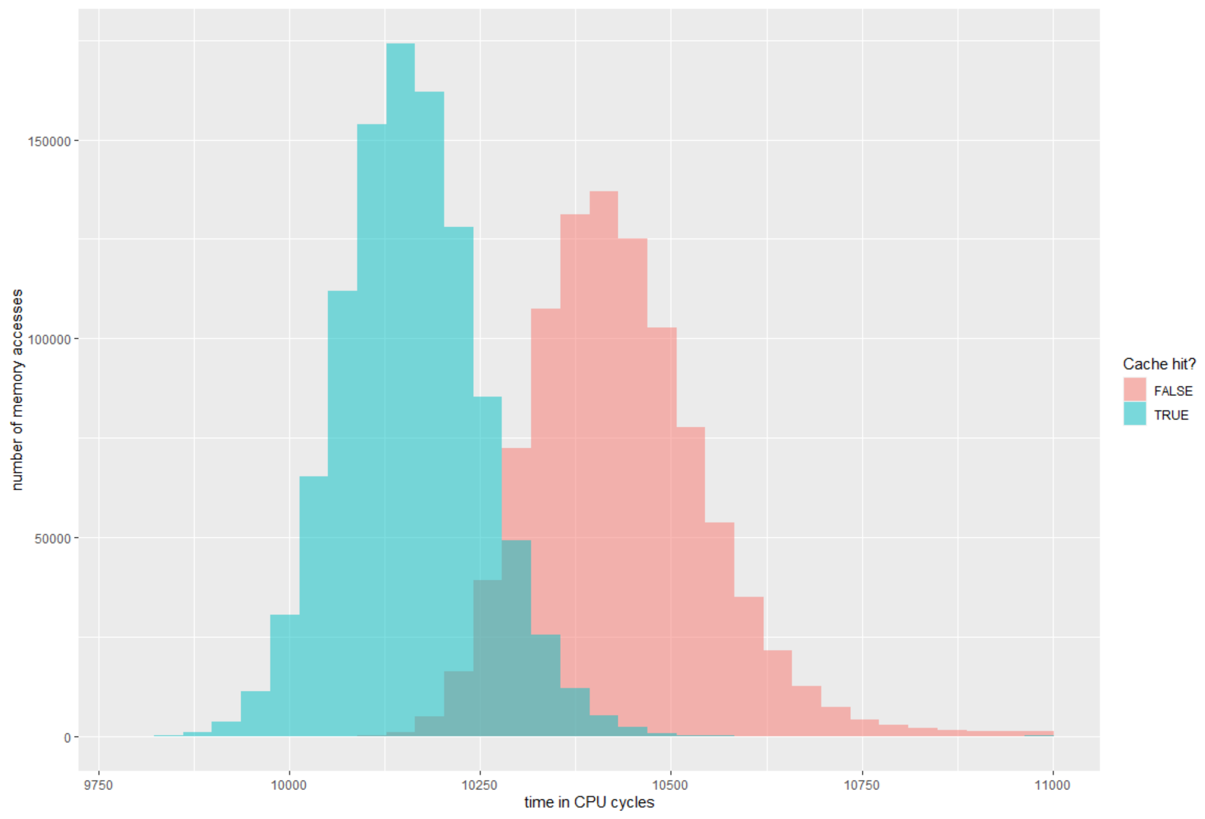

FIGURE 5.1: This histogram shows the result of 1,048,576 timing measurements of cache hits (TRUE) and cache misses (FALSE) each, using the power-ups *rdtsc* and *flush* on an AMD Ryzen 7 5800X. The x-axis shows the time it takes to access data in memory in CPU cycles. Note that this CPU model refreshes the rdtsc value every 38 CPU cycles, which explains the high bin width of this histogram [32].

repeating that experiment 50 times. The mean success rate of the individual experiments is 46.81% (SD = 5.98), which shows that reducing delay for timing operations, such as measuring the access times to data in memory, is critical for timing attacks. We interpret the aforementioned success rate as not sufficiently reliable and therefore use the power-ups *reload* and *flushandreload* in our Flush+Reload PoC, which shows a high success rate and reliability, and therefore the practicability of this attack in a browser environment.

On Machine 1 (Linux, Debian-based) we are not able to successfully execute our Flush+Reload PoC. We suspect, that this might be due to a different behaviour of the compiler on Linux systems, since we build the Chromium versions for Windows and Linux on their respective operating system, while using the same code for both versions. Therefore, identifying the reason for this problem and possibly modifying our

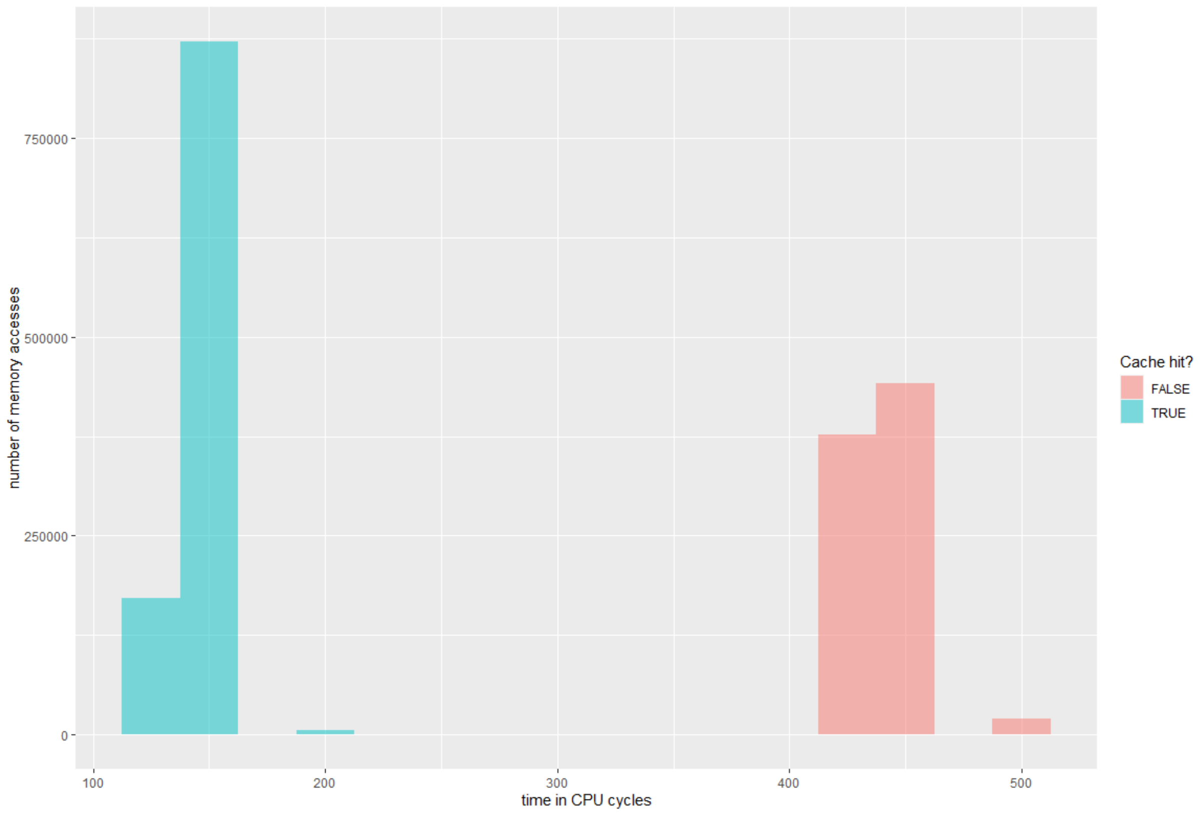

FIGURE 5.2: This histogram shows the result of 1,048,576 timing measurements of cache hits (TRUE) and cache misses (FALSE) each, using the power-ups *reload* and *flushandreload* on an AMD Ryzen 7 5800X. The x-axis shows the time it takes to access data in memory in CPU cycles. Note that this CPU model refreshes the rdtsc value every 38 CPU cycles, which explains the high bin width of this histogram [32].

version of Chromium could be subject to future work.

The description of the Spectre attack by Kocher et al. [14], includes a PoC written in C (in the following called *Spectre PoC A*), which is the basis of our implementations. We adapt the *Spectre PoC A* to work as C++ code in Chromium and add the power-up *spectre* which allows to execute *Spectre PoC A* in a browser environment (in the following called *Spectre PoC B*). The evaluation results of *Spectre PoC B* in Section 4 show that the original *Spectre PoC A* works with a high success rate in a browser environment. Additionally, we implement the attack described in *Spectre PoC A* using our low-level functions and code snippet power-ups, completely in JavaScript (in the following called *Spectre PoC C*). However, we are not able to execute a successful attack using the *Spectre PoC C*. We take the following steps to implement *Spectre PoC C*.

First, we verify that our test machines are indeed vulnerable against this attack, by running *Spectre PoC A*. Next, we test Flush+Reload as a covert channel for the Spectre attack in our *Spectre PoC C*, by accessing the targeted data before leaking, so that the data is present in the cache. We achieve results similar to the ones described in Section 4 for the Flush+Reload attack, which shows that the covert channel is working as intended. Finally, all of the aforementioned Spectre PoCs implement Spectre-PHT (also known as Spectre v1), which relies on branch misprediction, which in turn leads to speculative out-of-bounds access. Using the power-up *getByteAtAddress* we are able to successfully verify that the out-of-bounds access during the speculative execution reaches the correct memory content. However, we are not able to successfully execute a Spectre attack using *Spectre PoC C*. We suspect that this is because the speculative execution of the branch is not happening as anticipated. The target branch of the attack in the implementation is an *if*-statement that checks whether a provided value is in-bounds of the target array. In the *Spectre PoC A*, this conditional check is implemented in a way that causes a delay, which favors speculative execution. This can be achieved, e.g., by flushing the respective value used in the conditional check from the cache, or by using heavy computational tasks such as floating divisions. We suspect that the reason *Spectre PoC C* is not working correctly is that, during the conditional check, no speculative execution is performed by the CPU, even when the conditional check does not include heavy computational executions. It is possible that this happens due to the previously described overhead in Chromium, or optimizations made by the renderer Blink or the JavaScript engine V8. Therefore, a working implementation of *Spectre PoC C* could be subject to future work. Future work might also be interested in implementing other versions of the Spectre attack, as this work only focuses on Spectre-PHT (also known as Spectre v1).

6 Conclusion

The discovery and understanding of microarchitectural attacks in order to develop mitigations is a laborious task. Today's computer systems include complex optimizations, such as caches and transient executions, to overcome physical limitations. Multi-threaded systems provide the possibility of concurrent executions, which result in often unpredictable behaviour on the microarchitectural level. Since Chromium [4] uses additional optimization techniques, there exist even more factors that impede the understanding of microarchitectural attacks in a browser environment. Due to the high-level nature of programming languages used in browsers, such as JavaScript, low-level functions that allow operations, e.g., on the CPU cache, are not directly available. Therefore, we provide *power-ups* for Chromium, i.e., access to low-level functions and code snippets that can facilitate the research process of finding and validating microarchitectural attacks in a browser environment.

We evaluate these power-ups by implementing Proof-of-Concepts (PoCs) of the attacks Flush+Reload [5], Spectre [14], and Meltdown [6] in JavaScript using our version of Chromium. The evaluation of these PoCs shows a high success rate (see Table 4.1), and therefore the practicability of the power-ups for research purposes. The scope of this work includes basic low-level functions and code snippets, as well as API-style calls to the library *libkdump* [21], included in a new package called *Sca*. In future work, the collection of power-ups could be extended with additional functionalities, e.g., signal handling [33] which is used by the Meltdown attack. Additionally, the implementation of a function to retrieve the pointer of any JavaScript object can be subject to future work.

The power-ups for Chromium developed in this work show reliable functionality, and can thus be an asset to researchers looking to implement and test microarchitectural attacks in a browser environment.

References

[1] Cybersecurity Ventures Steve Morgan. Global Ransomware Damage Costs Predicted To Reach $20 Billion (USD) By 2021. https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-20-billion-usd-by-2021/. Accessed: 2022-01-10. 2019.

[2] IBM Security. Cost of a Data Breach Report. https://www.ibm.com/security/data-breach. 2021.

[3] Mozilla MDN Web Docs. performance.now(). https://developer.mozilla.org/en-US/docs/Web/API/Performance/now. Accessed: 2022-01-13.

[4] Chromium - The Chromium Project. https://www.chromium.org/Home.

[5] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[6] Moritz Lipp et al. Meltdown: Reading Kernel Memory from User Space. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[7] Raphael Spreitzer et al. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. In: *IEEE Communications Surveys Tutorials* 20.1 (2018), pp. 465–488. DOI: 10.1109/COMST.2017.2779824.

[8] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[9] Johannes Götzfried et al. Cache Attacks on Intel SGX. In: EuroSec'17. Belgrade, Serbia: Association for Computing Machinery, 2017. ISBN: 9781450349352. DOI: 10.1145/3065913.3065915. URL: https://doi.org/10.1145/3065913.3065915.

[10] Can Intel® Software Guard Extensions (Intel® SGX) Provide Input/Output (I/O) Key Security? https://www.intel.com/content/www/us/en/support/articles/000088404/software/intel-security-products.html. Accessed: 2022-01-07.

[11] The GNU C Library (glibc). https://www.gnu.org/software/libc/. Accessed: 2022-01-10.

[12] Dr. Michael Schwarz. Side-Channel Attacks and Defenses. Lecture slides. Winter Term 2020/21.

[13] Intel. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3 (3A, 3B 3C): System Programming Guide.

[14] Paul Kocher et al. Spectre Attacks: Exploiting Speculative Execution. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[15] Claudio Canella et al. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[16] Ben Goodger. Welcome to Chromium. https://blog.chromium.org/2008/09/welcome-to-chromium_02.html.

[17] Microsoft Edge Team. Microsoft Edge and Chromium Open Source: Our Intent. https://github.com/MicrosoftEdge/MSEdge/blob/14b03579d54acc5c294f846ad23f688a7eb8c124/README.md.

[18] Opera Software. The vision behind Opera 15 and beyond. https://blogs.opera.com/desktop/2013/07/the-vision-behind-opera-15-and-beyond/. Accessed: 2022-01-13.

[19] haraken@. How Blink works. http://bit.ly/how-blink-works. Accessed: 2022-01-13.

[20] The Chromium Projects. Web IDL interfaces. https://www.chromium.org/developers/web-idl-interfaces. Accessed: 2022-01-15.

[21] IAIK. libkdump. https://github.com/IAIK/Meltdown/tree/master/libkdump.

[22] Michael Lippautz Anton Bikineev Omer Katz. High-performance garbage collection for C++. https://v8.dev/blog/high-performance-cpp-gc. Accessed: 2022-01-15.

[23] Peter Marshall. Trash talk: the Orinoco garbage collector. https://v8.dev/blog/trash-talk. Accessed: 2022-01-18.

[24] Mozilla MDN Web Docs. ArrayBuffer. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer. Accessed: 2022-01-18.

[25] Andries Brouwer. mmap(2) - Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html. Accessed: 2022-01-20.

[26] Linux Sandboxing. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/linux/sandboxing.md. Accessed: 2022-01-22.

[27] Mozilla MDN Web Docs. DOMHighResTimeStamp. https://developer.mozilla.org/en-US/docs/Web/API/DOMHighResTimeStamp. Accessed: 2022-01-15.

[28] Mozilla MDN Web Docs. Web Workers API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API. Accessed: 2022-01-13.

[29] Yossef Oren et al. 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*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, 1406–1418. ISBN: 9781450338325. DOI: 10.1145/2810103.2813708. URL: https://doi.org/10.1145/2810103.2813708.

[30] Daniel Gruss et al. Flush+Flush: A Fast and Stealthy Cache Attack. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez. Cham: Springer International Publishing, 2016, pp. 279–299. ISBN: 978-3-319-40667-1.

[31] IAIK. Meltdown Proof-of-Concept. https://github.com/IAIK/meltdown.

[32] Moritz Lipp et al. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. Taipei, Taiwan: Association for Computing Machinery, 2020, 813–825. ISBN: 9781450367509. DOI: 10.1145/3320269.3384746. URL: https://doi.org/10.1145/3320269.3384746.

[33] Andries Brouwer. signal(7) - Linux manual page. https://man7.org/linux/man-pages/man7/signal.7.html. Accessed: 2022-01-25.

A Directory tree

```
├─ bindings
│  ├─ generated_in_modules.gni
│  └─ idl_in_modules.gni
├─ modules
│  ├─ sca
│  │  ├─ BUILD.gn
│  │  ├─ dom_window_meltdown.cc
│  │  ├─ dom_window_meltdown.h
│  │  ├─ dom_window_sca.cc
│  │  ├─ dom_window_sca.h
│  │  ├─ idls.gni
│  │  ├─ libkdump.cc
│  │  ├─ libkdump.h
│  │  ├─ meltdown.cc
│  │  ├─ meltdown.h
│  │  ├─ meltdown.idl
│  │  ├─ ptedit_header.h
│  │  ├─ sca.cc
│  │  ├─ sca.h
│  │  ├─ sca.idl
│  │  ├─ window_meltdown.idl
│  │  └─ window_sca.idl
│  └─ ...
└─ ...
```

The directory tree contains a list of all files and directories in $src/third\_party/blink/renderer/$ that were added in the scope of this work. Files with color blue where modified, and files and the directory in green where newly added.

B Threshold calculation in JavaScript

```javascript
let hit_sum = 0;
let miss_sum = 0;
sca.flush(addr);
for (let i = 0; i < n; i++) {
    const delta = sca.flushandreload(addr);
    miss_sum += delta;
}
sca.maccess(buff_addr);
for (let i = 0; i < n; i++) {
    const delta = sca.reload(addr);
    hit_sum += delta;
}
let avg_miss_time = miss_sum / n;
let avg_hit_time = hit_sum / n;
CACHE_HIT_THRESHOLD = (avg_miss_time + avg_hit_time * 2) / 3;
```

LISTING B.1: The code snippet shows the threshold calculation in order to distinguish between cache hits and cache misses in JavaScript using power-ups. The code takes an address *addr* and uses the power-ups *reload* and *flushandreload* to sum up all measured timing differences in *n* iterations. Finally, the average value of hits and misses (*miss_sum* / *n* and *hit_sum* / *n*) is used to calculate the threshold.

C Power-ups documentation

In the following, we provide a description of the added power-ups in alphabetical order and with their respective class in the format
returnType ClassName::functionName(ArgumentType args...).

void Meltdown::cleanup()

The power-up *cleanup* provides an API-style call to the *libkdump_cleanup* function included in the library *libkdump* [21]. Note: This power-up is only available on Linux systems. On other systems, the power-up does nothing.

String Meltdown::configToString()

The power-up *configToString* returns a string representation of the current configuration of the library *libkdump* [21]. Note: This power-up is only available on Linux systems. On other systems, the power-up returns an empty string.

String Meltdown::init(unsigned long cacheMissThreshold, String faultHandling, int measurements, int acceptAfter, int loadThreads, String loadType, int retries, unsigned long long physicalOffset)

The power-up *init* provides an API-style call to the *libkdump_init* function included in the library *libkdump* [21] using the given configuration parameters, and returns a string representation of the resulting configuration. Note: This power-up is only available on Linux systems. On other systems, the power-up returns an empty string.

String Meltdown::initAutoconfig()

The power-up *initAutoconfig* provides an API-style call to the *libkdump_init* function included in the library *libkdump* [21] using the function *libkdump_get_autoconfig*, and returns a string representation of the resulting configuration. Note: This power-up is only available on Linux systems. On other systems, the power-up returns an empty string.

int Meltdown::read(unsigned long long addr)

The power-up *read* provides an API-style call to the *libkdump_read* function included in the library *libkdump* [21]. The power-up takes an address as argument, leaks the corresponding byte in memory using Meltdown [6], based on the current configuration and returns the leaked value as *int*. If the configuration has not yet been initialized, the power-up initializes and uses the automatic configuration as seen in *initAutoconfig*. Note: This power-up is only available on Linux systems. On other systems, the power-up returns 0.

unsigned long long Sca::bufferAddress(DOMArrayBuffer* buffer)

In JavaScript, it is not possible to retrieve the address of JavaScript objects programmatically. The power-up *bufferAddress* takes an ArrayBuffer JavaScript object as argument and returns its address. The returned address is used, e.g., to flush the object from the cache. It would be desirable to retrieve the address of any JavaScript object, however, Chromium's JavaScript engine V8 is highly optimized, e.g., it uses a garbage collector, called *Oilpan*, for managing C++ memory [22]. JavaScript objects and the C++ object graph are heavily tangled and are therefore treated as one heap from the garbage collector. The garbage collector marks all reachable object it discovers, and then frees other dead objects, i.e., objects that cannot be reached in any possible execution and are therefore unused. The garbage collector not only frees dead objects, it also moves objects that survived the garbage collection [23]. Therefore, a power-up that retrieves the address of any JavaScript object is not feasible for this work, since pointers for these objects can change at any time. However, the *ArrayBuffer* object represents a generic, fixed-length raw binary data buffer and can be manipulated by, e.g., typed array objects [24] and has a fixed address in memory.

unsigned long Sca::detectThreshold()

The power-up *detectThreshold* calculates and proposes a threshold to distinguish between cache hits and cache misses, and returns the value. The value is calculated by accessing dummy data in the cache 1 million times without flushing (cache hits) and 1 million times with flushing (cache misses) it in between iterations. The threshold is then calculated by using the respective average values in the following statement [1]:

```
(avg_miss_time + avg_hit_time * 2) / 3
```

The resulting value is used, e.g., in cache timing attacks, such as Flush+Reload [5].

void Sca::flush(unsigned long long addr)

The power-up *flush* takes an address as argument and flushes the data at this address from the cache. This power-up is used, e.g., in the Flush+Reload attack [5] to remove data from the cache and observe, whether it was accessed by another process or not. Calling the flush instruction directly facilitates the process of building laborious workarounds such as eviction sets as described by Oren et al. [29].

unsigned long long Sca::flushandreload(unsigned long long address)

The power-up *flushandreload* takes an address as argument and returns the time it takes to access the memory at the given address. Additionally, after the time measurement, the power-up calls the *flush* instruction to remove the data at the given address from the cache. First, it uses the *rdtsc* instruction to get the current timestamp of the processor. Second, the power-up access the data at the given address using *maccess*. Next, the power-up retrieves another timestamp using *rdtsc* and calculates the difference to the first timestamp. Finally, the power-up calls the *flush* instruction:

[1]This is similar to existing implementations, such as seen in *libkdump* [21]

```
unsigned long time = rdtsc();
maccess(address);
unsigned long delta = rdtsc() - time;
flush(address);
return delta;
```

This power-up is used, e.g., in cache timing attacks, such as Flush+Reload [5]. Although the same functionality can be achieved using the JavaScript power-ups *rdtsc*, *maccess* and *flush*, the power-up *flushandreload* provides more accurate results, since JavaScript might introduce an overhead of about 10,000 cycles, according to our own measurements (see Section 5).

unsigned long long Sca::flushdelta(unsigned long long address)

The power-up *flushdelta* takes an address as argument and returns the time it takes to flush the memory at the given address from the cache. First, it uses the *rdtsc* instruction to get the current timestamp of the processor. Next, the power-up flushes at the given address from the cache using the *flush* instruction. Finally, the power-up retrieves another timestamp using *rdtsc* and calculates the difference to the first timestamp:

```
unsigned long time = rdtsc();
flush(address);
unsigned long delta = rdtsc() - time;
return delta;
```

This power-up is used, e.g., in cache timing attacks, such as Flush+Flush [30]. Although the same functionality can be achieved using the JavaScript power-ups *rdtsc* and *flush*, the power-up *flushdelta* provides more accurate results, since JavaScript might introduce an overhead of about 10,000 cycles, according to our own measurements (see Section 5).

unsigned long Sca::getByteAtAddress(unsigned long long address)

The power-up *getByteAtAddress* takes an address as argument and returns the corresponding byte in memory. As it is not possible to retrieve data from a specific memory address in JavaScript directly, this power-up is meant to facilitate the process of testing and verification of an attack written in JavaScript.

unsigned long long Sca::getPhysicalAddress(unsigned long long vaddr)

The power-up *getPhysicalAddress* takes a virtual memory address as argument and, on success, returns the corresponding physical memory address. If run on Linux, the power-up tries to map the address using the pagemap file in */proc/self/pagemap*. However, this operation requires root privileges. Therefore, on failure or if executed on Windows systems, the power-up uses the PTEditor kernel module [2] and returns the mapped physical address as *unsigned long long*. On failure or if the the PTEditor kernel module is not installed/loaded, the power-up returns 0. Note: Chromium uses sandboxing which restricts access to some parts of the file system [26]. Therefore, if *getPhysicalAddress* is used with the PTEditor kernel module, Chromium needs to be started with the *–no-sandbox* flag.

void Sca::maccess(unsigned long long addr)

The power-up *maccess* takes an address as argument and accesses this address in memory. This power-up is used, e.g., to implicitly load data into the cache. Alternatively, it is possible to access the data, e.g., by moving it into a dummy variable. However, since the power-up *maccess* uses a single line of assembly code, it has a possible performance advantage and is therefore more suitable for operations that rely on accurate timing.

String Sca::meltdown()

The power-up *meltdown* executes the code of the file *test.c*, which is included in the repository of the Meltdown Proof-of-Concept [31]. The power-up takes a randomly

[2]`https://github.com/misc0110/PTEditor`

selected string from a list and uses the Meltdown bug [6] to leak the string again. The returned string includes information about the expected and actual result of the attack.

long Sca::memmap(unsigned long long addr, unsigned long length, long prot, long flags, long fd, unsigned long offset)

The power-up *memmap* provides an API-style call to the Linux function *mmap* as documented on the Linux manual page [25]. If the mapping fails, the power-up returns 0. Values for the flag arguments *prot* and *flags* can be accessed using *Sca::FLAGNAME()*. Note: This power-up is only available on Linux systems. On other systems, the power-up returns 0.

long Sca::memunmap(unsigned long long addr, unsigned long length)

The power-up *memunmap* provides an API-style call to the Linux function *munmap* as documented on the Linux manual page [25]. On success, the power-up returns 0. Note: This power-up is only available on Linux systems. On other systems, the power-up returns -1.

DOMHighResTimeStamp Sca::rdtsc()

The power-up *rdtsc* uses the rdtsc instruction and returns the processors current time stamp, i.e., the number of cycles since the last reset [13], as a DOMHighResTimeStamp (*double*) [27]. This timestamp is used for timing attacks that require the accurate measurement of timing differences, e.g., the time it takes to access data in memory. The JavaScript function *performance.now()* returns a timestamp which is accurate to 5 microseconds, which is not sufficient to measure such timing differences [3][27]. Alternatively, it is possible to use a counting WebWorker [28] as counting thread. However, the provided power-up *rdtsc* saves time and keeps code clean.

unsigned long long Sca::reload(unsigned long long address)

The power-up *reload* takes an address as argument and returns the time it takes to access the memory at the given address. First, it uses the *rdtsc* instruction to get the current timestamp of the processor. Next, the power-up accesses the data at the given address using *maccess*. Finally, the power-up retrieves another timestamp using *rdtsc* and calculates the difference to the first timestamp:

```
unsigned long time = rdtsc();
maccess(address);
unsigned long delta = rdtsc() - time;
return delta;
```

This power-up is used, e.g., in cache timing attacks, such as Flush+Reload [5]. Although the same functionality can be achieved using the JavaScript power-ups *rdtsc* and *maccess*, the power-up *reload* provides more accurate results, since JavaScript might introduce an overhead of about 10,000 cycles, according to our own measurements (see Section 5).

String Sca::spectre()

This power-up executes the Proof-of-Concept code as shown in the paper by Kocher et al. [14]. On success, the code returns the string “The Magic Words are Squeamish Ossifrage.”.

String Sca::spectre(String sec)

This power-up executes the Proof-of-Concept code as shown in the paper by Kocher et al. [14] and leaks the string provided as argument using the Spectre attack. On success, the provided string is returned, an undefined string otherwise.