Bachelor Thesis

Spectify: Transforming Regular Vulnerabilities into Spectre Vulnerabilities

by

Chris Wagenaar
(Student Number: 2695346)

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

July 16, 2023

Certified by ....
Cristiano Giuffrida
Associate Professor
First Supervisor

Certified by ....
Mathé Hertogh
PhD. Student
Daily Supervisor

Certified by ....
TBD
Member VUSec faculty
Second Reader

Spectify: Transforming Regular Vulnerabilities into Spectre Vulnerabilities

Chris Wagenaar
Vrije Universiteit Amsterdam
Amsterdam, NL
c.c.j.wagenaar@student.vu.nl

ABSTRACT

This paper aims to study whether classic software vulnerabilities transfer into the speculative domain. In doing so, it presents 5 synthetic proof-of-concepts of Spectre V1 attacks. More specifically: buffer overread, buffer overflow, use-after-free, uninitialized read, and a concurrency bug. These proof-of-concepts are used to determine which standard software vulnerabilities transfer into the speculative domain, and how they work. We will explain and evaluate the proof-of-concepts extensively. While synthetic, these proof-of-concepts introduce the reader to multiple variations of Spectre attacks, raising awareness and understanding of these vulnerabilities in a basic form. Furthermore, we present a speculative mutex-lock-bypass, opening the door for research into speculative concurrency bugs.

1 INTRODUCTION

This Bachelor thesis focuses on reproducing speculative execution based attacks. Speculative execution is a hardware optimization present on practically all modern computer architectures, and since no comprehensive mitigations exist, this means practically all computers are vulnerable to this vulnerability type. The attacks rely on an optimization that predicts which code to run in advance, before knowing with certainty that this code is supposed to be run. When this prediction is wrong, an attacker can sometimes leak unauthorized data based on traces left behind by this wrongly executed code. The attacks will be synthetic proof-of-concepts. The aim is to show the reproducibility of these attacks, as well as to gain new insights into Spectre attacks. Spectre is a subset of speculative execution vulnerabilities that involves predictions of the control flow of a program (such as branch predictions and address speculation).

Since the discovery of speculative-execution based vulnerabilities in 2018 [8], much research has gone into finding new related vulnerabilities and finding possible mitigations [3, 5, 7, 13]. In the present, several mitigations have been proposed. Some speculative execution vulnerabilities, such as Meltdown and MDS, have been completely mitigated in hardware and/or microcode [7]. Yet Spectre, the main focus of this thesis, has not been mitigated. All current Spectre mitigations only prevent certain instances of speculative execution vulnerabilities, require new hardware, or require rewriting and/or recompiling code [11, 12]. Hence, none of these mitigations are fully satisfiable. Furthermore, hardware manufacturers have reduced efforts to find proper mitigations and instead advice programmers to manually mitigate and prevent these vulnerabilities. Much research focuses on the standard Spectre bounds-check-bypass (BCB) vulnerability as presented in the original paper regarding this topic [8]. However, this thesis shows that speculative execution vulnerabilities are much more diverse than just BCB by presenting various other vulnerability proof-of-concepts.

This lack of a comprehensive and realistically implementable mitigation results in speculative execution vulnerabilities still remaining an active research field. It is important to continuously gain new insights, and use these to design, implement and test better mitigation strategies.

Many programmers are unaware that vulnerable code might be executed speculatively, leaving traces. Even if aware, many programmers reason on a software level only. Yet identifying Spectre vulnerabilities requires reasoning in accordance with extensive hardware knowledge. As a matter of fact, many standard mitigations to well-defined software vulnerabilities include regular safety checks without specific speculative execution considerations (i.e. a bounds check for a buffer overflow). Thus, these standard mitigations can often be bypassed speculatively, which creates a large risk for possible undiscovered vulnerabilities [8].

This reproductive study aims to transfer standard software vulnerabilities into the speculative domain. Hence, the research question is:

> *"Which standard software vulnerabilities transfer into the speculative domain?"*

This paper will present and explain 5 synthetic proof-of-concepts of different Spectre V1 attacks. Spectre V1 being a subset of Spectre vulnerabilities, in which the control flow prediction specifically involves branch prediction. Particularly, we will investigate the following vulnerability types: buffer overread, buffer overflow, use-after-free, uninitialized read, and a concurrency bug. These were chosen because they are standard and common vulnerability types, especially in memory unsafe languages like C/C++, and because they have received much attention in memory safety literature [2].

Contributions:

⋆ Explanation of multiple Spectre V1 proof-of-concepts of standard vulnerability types.

⋆ Easily extendable codebase [14] with library of helper functions.

⋆ Presents a speculative mutex lock bypass proof-of-concept, which opens the door to research to speculative concurrency bugs.

2 BACKGROUND

Spectre vulnerabilities occur when code is executed speculatively; a CPU optimization that involves executing certain code ahead of time. Such execution leaves traces that can sometimes be leaked. This section will first discuss microarchitectural side channels, which are used to leak the aforementioned traces. Then, we will cover the workings of Spectre vulnerabilities, also known as transient execution attacks. Finally, this section will address the interplay of Spectre with other software vulnerabilities.

2.1 Microarchitectural Side Channels

Practically all modern computer architectures make use of both memory and cache. Memory can store much more data, but accessing it takes a relatively long time. Therefore, CPUs also have cache memory. This cache is much faster than regular memory, but has a lower capacity. The cache is typically split in three layers, ordered from fast and small to slower and larger: L1, L2 and L3. The first two are usually CPU core specific, while the L3 cache is shared among all cores. Yet due to cache coherence protocols, cache values between different cores are kept consistent [11].

After executing code, the used variables often remain in cache. These traces can be used to leak data using a covert channel such as flush+reload [16]. In essence, an eavesdropper would first flush all variables of interest from cache. Then, the eavesdropper waits a certain amount of time for the victim to potentially use some of these variables. These used variables will likely remain in cache. At a later point in time, the eavesdropper reloads the variables. A fast load means that the variable was in cache and thus used by the victim. A slow load from memory suggests - but does not guarantee - that the variable was not used.

2.2 Transient Execution Attacks

When programming, especially without much low-level knowledge, it seems self-evident that the computer will exactly follow the steps provided in code. More knowledgeable programmers might be aware that the compiler makes optimizations, but that the end result will always be identical to what was initially programmed. This sensible expectation, however, does not entirely hold for optimizations such as out-of-order execution and speculative execution.

Out-of-order execution executes instructions ahead of time. The result will be temporarily stored, but only retired when all prior instructions have finished execution [8]. In this context, retired means permanently committing or storing the results of an operation. Out-of-order execution works in a way that ensures the end result will always be the same as if it was executed linearly.

One form of such out-of-order execution is speculative execution. Imagine a program has to validate a condition to know whether to take the branch of an if-statement. In this case, when the CPU has to wait on a memory load to verify the branch condition, the branch predictor will instead predict whether to take the branch or not. Branch prediction is an optimization technique that involves predicting which branch will be taken based on which branches were taken prior. This is very efficient in case the variables to verify the branch-condition need to be retrieved from memory (which is slow), while the required variables to execute the code within the branch are in cache (which is fast). Based on this prediction, it will continue executing based on the fast-loading variables in cache, until the memory load has completed and the branch condition can actually be verified. Comparable Spectre vulnerabilities make use of exception prediction or address speculation [3].

However, it is possible that a cache value is stale, or that one of the predictions turned out to be wrong. In that case, the CPU will perform a machine clear [11]. This entails that the effects of the speculatively executed code will not be retired (meaning that all speculative progress will be undone), after which the correct branch will be executed. However, this machine clear leaves traces of the speculatively executed code. The variables that were loaded from memory often remain in cache for some time. A covert channel such as flush+reload could be used to leak those variables from cache. This paper will focus on Spectre V1, which encapsulates Spectre vulnerabilities that involve a direct branch (mis)prediction.

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

Listing 1: An example of an out-of-bounds read [8].

It is not uncommon for Spectre V1 research to focus on a simple speculative buffer overread (often referred to as bounds-check-bypass or BCB), as presented in the original Spectre paper [8], also see Listing 1. This very simple proof-of-concept served as a great starting point. However, Spectre V1 vulnerabilities are much more diverse than only buffer overreads. Thus, research that only considers a speculative bounds-check-bypass can be incomplete. To move past this singular proof-of-concept, this paper presents 5 different Spectre V1 proof-of-concepts.

An important optimization that some Spectre V1 vulnerabilities rely on is speculative store-to-load forwarding. When a store instruction is encountered and a write to cache is started, the CPU will already try to continue executing instead of awaiting this store operation to finish. Therefore, the value is quickly written to the store buffer. This is also where the unretired results of speculative execution are stored. Furthermore, the store buffer is used to ensure memory consistency and coherence. Spectre vulnerabilities that involve a speculative write can read this written value from the store buffer during the speculative execution window. This way, speculative store-to-load forwarding allows for Spectre vulnerabilities with a write operation [3].

2.3 Interplay with Software Vulnerabilities

Software vulnerabilities are usually classified as a certain type. Vulnerabilities of the same type usually have a similar working, and also similar mitigations. Some common vulnerability types are:

- ⋆ **Buffer overflow**: caused by an out-of-bounds write to an array. Results in a different variable being overwritten.
- ⋆ **Buffer overread**: caused by an out-of-bounds access of an array. Results in a different value than the array being read.
- ⋆ **Integer overflow**: caused by a number being stored in a variable with too little space. Results in the higher bits being cut off, similar to a modulo operation.
- ⋆ **Memory leak**: caused by allocated memory not being freed. Results in the computer running out of memory for new variables.
- ⋆ **Race condition**: caused by multiple threads *racing* to do one or more operations on the same variable. Can result in undefined behavior, since it is not certain which thread executes the operation(s) first.
- ⋆ **Uninitialized read**: caused by an allocated variable being used that was not initialized. Results in the variable that resided at the address before allocation to be used.

⋆ **Use-after-free**: caused by a pointer being used after it was freed. Can result in undefined behavior, since the memory address might be in use by another variable by then.

The previously mentioned vulnerability types and their mitigations are well defined in the field. However, these mitigations usually do not take speculative execution into account. An if-statement used to mitigate a vulnerability can be bypassed speculatively, after which the vulnerable code will be executed regardless. Furthermore, speculative execution can result in a different control flow than the one defined architecturally. This means that vulnerabilities which are architecturally impossible or mitigated are sometimes possible during speculative execution.

Speculative execution opened the door to many research projects. One such project is Blindside, which takes BROP attacks into the speculative realm. BROP (Blind Return Oriented Programming) relies on at least one buffer overflow vulnerability in software and probes software based on different inputs. Based on whether the software crashes or not, it finds ROP gadgets. Using these, its final goal is to create a shell. The main drawbacks are that these crashes are easily noticeable, and the software has to be crash-resistant. Blindside does not have these drawbacks because the crashes occur during speculative execution, and these are simply suppressed [5].

A paper about SPEAR discusses SPEculative ARchitectural control flow hijacks. The main takeaway is that many buffer overflow mitigations do not always mitigate speculative buffer overflow vulnerabilities. As a result, an attacker can use vulnerabilities like bounds-check-bypass to alter the control flow of a program and create speculative ROP [10].

Another paper covers PACMAN, a speculative bypass of ARM's Pointer Authentication (PA). Simply put, PA assigns certain protected pointers a hash called the Pointer Authentication Code (PAC). Whenever this pointer is called, PA checks whether the pointer matches the hash. When an attacker has changed the value of the pointer, they will have to provide the correct hash to pass the PA. If this fails, the program crashes. Hence, brute forcing the hash will lead to many crashes, which is easily observed by anomaly detection tools. However, when this hash is verified speculatively, the crashes are suppressed meaning detection tools will not register crashes when the PAC is leaked by brute force. When the hash is leaked, Pointer Authentication can be bypassed [13].

3 THREAT MODEL

This paper focuses on synthetic proof-of-concept attacks for various Spectre-related vulnerabilities, hence the threat model is also synthetic. Furthermore, for any Spectre vulnerability, it is assumed that both the victim and the attacker can write and execute code on the same machine. It is also assumed that the attack takes place on a machine that supports out-of-order execution, and that this is not disabled. The victim and the attacker's code does not have to be executed on the same CPU core [12]. An example of this threat model would be the victim and attacker both operating on the same server in the cloud. However, to simplify the proof-of-concepts, the victim and the attacker run the same program and address space. This means that formally, the threat model would be defined as an In-Domain Transient Execution Attack [6].

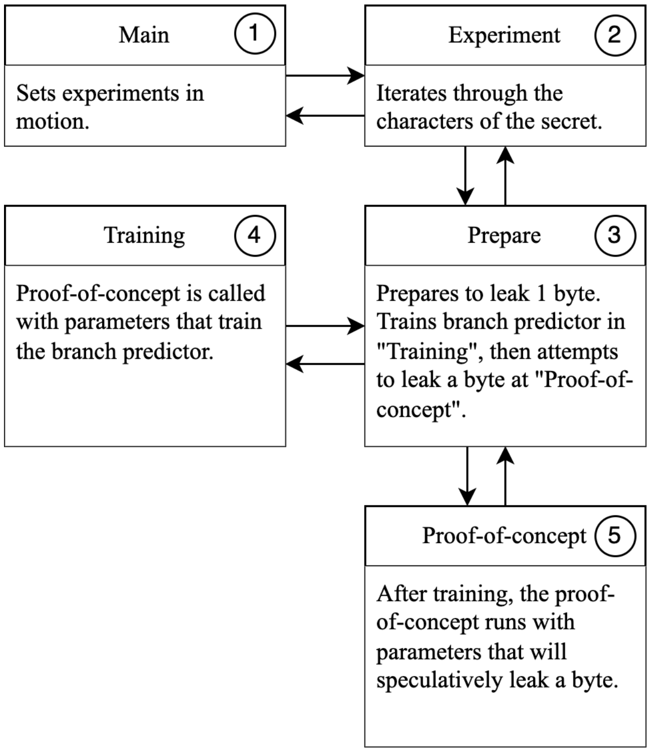
Main
1
Sets experiments in motion.
Experiment
2
Iterates through the characters of the secret.
Training
4
Proof-of-concept is called with parameters that train the branch predictor.
Prepare
3
Prepares to leak 1 byte. Trains branch predictor in "Training", then attempts to leak a byte at "Proof-of-concept".
Proof-of-concept
5
After training, the proof-of-concept runs with parameters that will speculatively leak a byte.

Figure 1: Diagram depicting the general structure of the proof-of-concepts.

4 OVERVIEW

This paper introduces 5 classic architectural vulnerabilities. Then, a standard architectural mitigation is proposed. After that, we try to find a speculative exploit of this architecturally mitigated code. These speculative exploits will be Spectre V1 proof-of-concepts, which will be used to find out which standard software vulnerabilities transfer into the speculative realm.

Each proof-of-concept makes use of a library we wrote containing a flush+reload (F+R) covert channel. The proof-of-concepts leak data by using a secret byte as an index in the F+R array, leaving traces that can later be used to deduce the value of the secret byte.

Each proof-of-concept has a similar structure, as depicted in Figure 1. The main goal is to leak a secret string containing "mysecret". The process starts in the 'Main' phase (1). This phase repeats the overall experiment multiple times. Not every leakage attempt is successful, for example, due to an OS rescheduling which makes a measurement take longer than it otherwise would have. Hence, the experiment to leak the secret string is repeated multiple times for robustness. The proof-of-concepts leak one byte at a time. Hence, an individual 'Experiment' (2) repetition involves iterating through each character of the secret string. For each character, the 'Prepare' (3) phase is entered. Before a leakage attempt is made, the proof-of-concept is executed multiple times to 'Train' (4) the branch predictor to take a favorable branch for the leakage attempt. After training, the 'Proof-of-concept' (5) vulnerability executes with the right parameters to leak a byte.

Repeating the experiment allows us to take the median loading time of the characters over multiple repetitions. Using these median values, an accurate reconstruction of the secret string can be made, even if many leakage attempts failed. After repeating this experiment multiple times, a print function in the library analyzes the data, makes a best prediction on what the leaked string was and calculates metrics such as the leakage rate.

The standard architectural software vulnerabilities that we attempted to translate into Spectre V1 counterparts are common software vulnerabilities in C/C++ code. They were picked because they are common and well-known, meaning that lots of research into them already exists. Furthermore, they were picked because we deemed these relatively easy to transfer into the speculative domain, allowing multiple proof-of-concepts to be finished within the time constraints of the project. Many more C vulnerabilities exist [2].

The individual proof-of-concept are best summarized as follows:

⋆ **Bounds-check-bypass**: Architecturally, an array can be accessed out-of-bounds. This means that an array of size X will be accessed at an index like X+1. As a result, an attacker could read out any adjacent data if they can control the index. An architectural mitigation would be an if-statement preventing out-of-bounds accesses. The speculative counterpart of a buffer overread is usually referred to as bounds-check-bypass, in which the mitigating if-statement is bypassed. In our proof-of-concept, the upper bounds value is flushed from memory, meaning that the outcome of the branch will have to be speculated. As a result, speculative execution could start executing the content of the if-statement before checking whether the access is out-of-bounds. This access will be leaked through a covert channel, allowing it to be leaked speculatively.

⋆ **Buffer overflow**: Similarly to a buffer overread, when writing out-of-bounds to an array, some other value adjacent to the array is overwritten. This way, an attacker who controls which index is written to could overwrite the value of other variables. An architectural mitigation would be an if-statement preventing this out-of-bounds write. In our proof-of-concept, this if-statement is speculatively bypassed, which results in a speculative out-of-bounds write. By aligning the array and the password to be adjacent in memory, this write overwrites the password. By overwriting the password with a value the attacker knows, the attacker can give the *correct* password and bypass the authentication.

⋆ **Use-after-free**: After a variable is freed, a new memory allocation could be assigned to the memory address of the previously freed variable. If the original variable is used after being freed, the new owner of the variable controls its value. This new owner could be an attacker, who by extension has control over some aspects of the program. The architectural solution would be to ensure variables are only freed after their last usage. Nevertheless, it is possible that code which architecturally would never cause a use-after-free, follows a speculative path of branches that does cause a use-after-free. In our proof-of-concept, the victim program allocates *var0* and *var1*. Then, either of these is freed based on the function parameter. The attacker then allocates a new variable of the same size, which gets the same address allocated as the recently freed variable. The attacker then gives this variable the index of the secret character they want to leak. The victim program checks if *var0* has been freed, and if not uses its value as an index to the secret string. However, this check can be mispredicted, leading to speculative usage of the variable that the attacker controls.

⋆ **Uninitialized read**: When a variable is allocated a certain address, its value is officially undefined. In practice, memory gets reused, so often this value is the value of a variable that has been previously deallocated. That previously deallocated variable could contain sensitive data, allowing it to be leaked. An architectural mitigation would be to initialize all allocated variables, or to only use these uninitialized variables when the memory holds no sensitive data to leak. However, a speculative branch misprediction might speculatively store this sensitive data on the stack, after which it could be leaked during the same speculative execution window. Our proof-of-concept will do exactly that. Based on the parameters, the victim function could load a secret character, let it go out of scope and then declare an initialized value. Alternatively it does not touch the secret value, and then declares an uninitialized value. Since all three variables are of the same size, they get the same address allocated. Yet because the uninitialized variable is only declared when the secret value was not touched, there is no architectural risk of an uninitialized variable leakage. However, speculative execution can predict to declare the uninitialized variable after the secret character has been loaded. This way, a speculative uninitialized read vulnerability occurs and the character can be leaked.

⋆ **Mutex-lock-bypass**: In a multithreaded program, while one thread is using a variable, a different thread can read and write to that variable. This means that one thread can influence the behavior of another thread, which is sometimes unintended. In case this is unwanted, the architectural solution to this is to use locks to prevent concurrent access to this variable. However, our proof-of-concept shows that mutex locks can be bypassed speculatively. Based on the parameters, the victim function either locks *lock0* or *lock1* and sets the respective value of *lock_var0* or *lock_var1* to 0. The attacker then tries to lock *lock0*. If they manage to do so, they will set the value of *lock_var0* to the index they want to leak. *Lock_var0* is then used as an index to the secret string, the resulting value of which is leaked into a covert channel. Architecturally, it is impossible for the attacker to get the lock taken by the victim. Nonetheless, the proof-of-concept shows that this is possible during speculative execution.

5 PROOF-OF-CONCEPTS

The proof-of-concepts presented in this paper all have a very similar structure. The main goal is to leak a secret string without architecturally returning its value. All proof-of-concepts have a main function that sets attempts to leak a byte in motion. Within such an attempt, there is a function that calls the victim function multiple times. The proof-of-concepts also make use of the same custom library containing a flush+reload covert channel and a printing function. This will be further outlined in the following two subsections. Following this, the individual proof-of-concepts will be discussed. Note that some code listings of the proof-of-concepts have been simplified for easier reading, the full code can be found in the GitHub repository [14]. Finally, some of the implementation difficulties will be addressed.

5.1 General Approach

The program of a proof-of-concept starts in the main function. In this function, a data structure is created to hold the F+R data, and a timer is started. Then, it starts cycling through the repetitions. Each repetition, an attempt is made to leak the characters of the secret string one by one by calling the preparation function with the index of each secret character.

The preparation function attempts to leak one byte from the secret string. It simulates the attacker being able to call the victim function and set the parameters. Before it can do so, it has to train the branch predictor of the victim function to speculatively take the branch(es) required for the vulnerability to work. Training the branch predictor is done by repeatedly calling the victim function with parameters that take the branch you want to be taken during the attack. However, the intuitive approach of calling the victim function with hardcoded training parameters in a loop, and then with the attack parameters afterwards does not work. This is because the branch predictor can then learn when the loop ends, and learn from previous repetitions that the branch prediction should be different when the for-loop ends (which would be the attack phase). Instead, the parameters for the training and the attack have to be put in the same array. Then, all victim function calls are made from the same loop. This loop cycles through the parameters in the parameter array. As a result, the branch predictor cannot tell the difference between a training and an attack function call.

The victim functions themselves are presented individually in later subsections. However, there are some general patterns between them. First of all, the secret to be leaked is the global variable *SECRET*. A single character from the secret is usually leaked into the flush+reload array. This entry of the F+R array is then stored into a *volatile* variable. This *volatile* attribute ensures that the variable will get saved in memory and that the compiler does not remove the instruction as an optimization. There are also the functions *cpuid()* and *flush()*. *Cpuid()* executes the assembly instruction by the same name, this serializing instruction basically blocks speculative execution until all prior instructions have been executed. *Flush()* also calls an assembly instruction by the same name, which takes a memory address and removes the cache line that contains that address from cache.

After the experiment that leaks the secret has been repeated a certain number of times, the timer is stopped. Then, the print function from the custom library is called, taking the F+R data and the measured time. This will print the results.

5.2 Custom Library

The library contains a flush+reload implementation that makes use of an array of cache-page-sized structs, being 4096 bytes. This struct is of type *cp_t*, which stands for cache page type. This relatively large size is used to avoid entries being on the same cache line, as well as to prevent the prefetcher to preload entries when iterating through the array. The F+R implementation leaks 8-bits (or 1 character) at a time, meaning the array has 256 entries to represent each possible byte value. The way the leaking of F+R works is that a byte to be leaked is used as an index of the array. This will load that entry of the array into cache. Later when the array is reloaded, the covert channel measures the time it takes to load each entry. A fast load indicates that the entry was already in cache, meaning it was used by the victim. If, for instance, only the 10th entry seems to be in cache, then that means the byte-value of the leaked byte was likely 10. Note that after initializing the F+R covert channel, the entire array has to be flushed from memory. Otherwise, many of the entries will still be in cache from the initialization, and it is no longer certain whether the victim loaded an entry or not.

The print function takes the F+R data and the measured time. First, it lists all cache hits, memory load time measurements below constant *CACHE_HIT* (usually around 100). It also does some calculations. For each index in the secret array, it gets the median loading time for all 256 values over the repetitions. The lowest of these 256 median loading times will be the estimated value of a secret character. This is done for each index in the secret array. Using this data, the F+R data and the time measurement, multiple metrics are calculated and printed. These will be discussed in section 6.

```
#define SECRET      "mysecret"
#define BUF         "public"
int     BUF_SIZE =  7;

char victim_func(int index) {
    //if (index < BUF_SIZE) {
        return BUF[index];
    //} else return '?';
}
```

Listing 2: An example of an out-of-bounds read.

5.3 Bounds-check-bypass

Bounds-check-bypass is a well-known Spectre attack, and a multitude of examples of it can be found online [1, 4, 15]. The architectural vulnerability involves accessing an entry of an array that does not exist (e.g. accessing an array of size 10 at index 11). In languages like C, this simply means that whatever data resides next to the array will be loaded. In Listing 2, *victim_func()* accesses array *BUF* but does not check if the index is within bounds. Imagine *BUF*, which spans 7 bytes, resides on addresses 0 up and until 6. Imagine *SECRET*, which spans 9 bytes, resides on addresses 7 until 14. If index 7 is used on *BUF*, then this will point to address 7 (since address 0 + offset 7 points to address 7). This will then load the first character of *SECRET*. Hence, with indices 7 until 14, string "mysecret" can be leaked.

The architectural mitigation is to add a bounds check, such as by adding the commented lines in Listing 2. This way, any access equal to or above the value of *BUF_SIZE* will return '?' (for simplicity, a lower bound has been left out).

Speculatively, this if-statement can be bypassed, see the slightly simplified proof-of-concept Listing 3. Before attempting speculative execution, it is important to train the branch predictor. Hence, *victim_func()* will be called a multitude of times with an index that is within bounds. This significantly increases the accuracy of the speculative attack, because it trains the branch predictor to take the branch.

Furthermore, it is important to flush *BUF_SIZE* before reaching the if-statement. This way, *BUF_SIZE* has to be fetched from memory in order to verify the branch condition. This can take a long time, so the CPU will start working ahead by executing instructions

```c
#define SECRET      "mysecret"
#define BUF         "public"
int     BUF_SIZE =   7;
cp_t*   flush_reload_arr;

void victim_func(int index) {

    flush(&BUF_SIZE);
    cpuid();

    if (index < BUF_SIZE) {
        unsigned char x = BUF[index];
        cp_t cp = flush_reload_arr[x];
    }
}
```

Listing 3: Architecturally safe code with a speculative bounds-check-bypass vulnerability.

speculatively. Note that both *BUF_SIZE* and *flush_reload_arr* have an alignment attribute set to the size of a cache line (64 bytes). This is to prevent than when *BUF_SIZE*'s cache line is flushed, *flush_reload_arr* might also get flushed. This attribute has been left out of the code snippet for simplicity. Also note that *cpuid()* has to be placed between the flush and the if-statement. This is a serializing instruction, preventing any out-of-order execution to occur before *flush()* finishes executing. In many cases, the trained branch predictor will then predict the branch is taken. Speculatively, the *index*-th character of *BUF* will be used as an index of the flush+reload array *flush_reload_arr*, making its value traceable in the future. Note that because the index is out-of-bounds, the accessed entry of *BUF* will actually be a character of the adjacent array *SECRET*. This is repeated for each character in *SECRET*, after which the entire experiment is repeated multiple times.

5.4 Buffer Overflow

A buffer overflow is quite similar to a bounds-check-bypass, but it involves an out-of-bounds write to an array, instead of a read. Architecturally, when writing to index 11 of an array of size 10, the value overwrites whatever value was originally stored adjacent to the array. This way, arbitrary memory can be overwritten and corrupted during speculative execution. A classic exploit involves overwriting a return address or function pointer that is later used. This way, the control flow of the program can be altered to a different function. In doing this, a speculative ROP chain can be created, allowing for arbitrary code execution. However, for simplicity we demonstrate a proof-of-concept that speculatively leaks characters from a secret string, while this is architecturally impossible.

In Listing 4, the adjacent overwritten variable is the password. In this proof-of-concept, the attacker overwrites the password to gain access to *SECRET*. In the real proof-of-concept variables *buf* and *password* have been put in a struct because this made it easier to control their adjacency, yet this has been left out of the snippet to preserve simplicity.

The architectural mitigation to this vulnerability is also to implement a bounds check. An example of this can be seen in the commented lines in Listing 4. When these lines are uncommented, any *user_id* value equal to or above *BUF_SIZE* will be ignored.

```c
#define BUF_SIZE    16
#define SECRET      "mysecret"
char    buf[BUF_SIZE];
char    password;

char victim_func(int user_id, char user_char,
                 char user_password, int secret_index) {
    password = 'x';
    //if (user_id < BUF_SIZE) {
        buf[user_id] = user_char;
    //}

    if (user_password == password) {
        return SECRET[secret_index];
    } else return '?';
}
```

Listing 4: An example of a buffer overflow vulnerability.

```c
#define BUF_SIZE    16
#define SECRET      "mysecret"
int     buf_size =  BUF_SIZE;
char    buf[BUF_SIZE];
char    password;

void victim_func(int user_id, char user_char,
                 char user_password, int secret_index,
                 cp_t* flush_reload_arr) {
    password = 'x';
    flush(&buf_size);
    cpuid();

    if (user_id < buf_size) {
        buf[user_id] = user_char;
    }

    if (user_password == password) {
        cp_t cp = flush_reload_arr[SECRET[secret_index]];
    }
}
```

Listing 5: Architecturally safe code with a speculative buffer overflow vulnerability.

However, this if–statement can also be bypassed speculatively, see Listing 5. Once again, the branch predictor has to be trained first. This is done by executing the function multiple times, with parameters that will take both branches.

During the attack phase, it is important that all variables are in cache, except *buf_size* which specifically has to be flushed from cache. This way, speculative execution will start at the first if-statement, since *buf_size* has to be fetched from memory to verify the branch condition. Since all other required variables are in cache, the rest of the code can be executed in advance, while *buf_size* is being fetched from memory. Note that the code shows *BUF_SIZE* and *buf_size*. This is because the compiler did not allow non-constant variable *buf_size* to be used to determine the size of *buf*. At the same time, a macro like *BUF_SIZE* cannot be flushed from cache, so both *BUF_SIZE* and *buf_size* are needed. Furthermore, *buf_size* has an alignment attribute to prevent *buf_size* and *buf* or *password* from residing on the same cache line. This would be problematic since *flush()* flushes an entire cache line. This attribute and a struct containing *buf* and *password* have been left out of the code listing

```c
#define SECRET      "mysecret"
#define PAGE_SIZE   4096

void* attack_func(int secret_index) {
    int* var1_dupe = malloc(PAGE_SIZE);
    *var1_dupe = secret_index;
    return var1_dupe;
}
char victim_func(int secret_index) {

    int* var1 = calloc(1, PAGE_SIZE);
    free(var1);

    int* var1_dupe = attack_func(secret_index);

    char secret_char = SECRET[*var1];
    free(var1_dupe);
    return secret_char;
}
```

Listing 6: An example of a use-after-free vulnerability.

to preserve simplicity. To initiate the overflow, *user_id* will be set to *BUF_SIZE*. Since *buf* and *password* are adjacent to each other, *buf[BUF_SIZE]* points to the address of *password*. Hence, when the write to *buf* is made, this will actually speculatively overwrite *password* with *user_char* by means of store-to-load forwarding. The attacker can then simply use the same value for *user_char* and *user_password* to bypass the second if-statement, without knowing the actual password. In doing so, the second branch is speculatively executed and a byte from *SECRET* will be leaked into the F+R array.

5.5 Use-after-free

A use-after-free vulnerability occurs when an allocated variable has been freed, but is still used afterwards. If this happens, it is possible that the previously allocated memory is allocated again for a different variable. When a value is written to this different variable, the value of the original variable is also changed. In the event that an attacker gets control of this memory, they can use this to control the variable that the victim is going to use. This way, the attacker can often change the control flow or the result of the victim program. The overwritten value can also be used to set a different vulnerability in motion. In the speculative proof-of-concept we present, the overwritten value will be used to dictate which character of the secret string will be leaked.

An example can be found in Listing 6. First, *var* is allocated memory, initialized to 0, and then freed. Afterwards, *attack_func()* requests the same amount of memory, and gets the memory address of *var* allocated. *PAGE_SIZE* has been set to the size of a memory page, because this seemed to help getting the same address allocated. The architectural mitigation is to ensure variables are only freed after their last usage. This way, a new memory allocation will never get assigned the same address.

Speculatively, it is still possible that a use-after-free vulnerability occurs in code in which it is architecturally impossible. Consider the code in Listing 7. The real proof-of-concept looks a bit complicated. The main reason for this is that the code can't contain input-dependent branches, because this would help the branch predictor identify the training phase input from the attack phase input. If an if-statement was used to determine whether to free *var0* or *var1*, then the branch predictor can take this into account in the branch history and use it to predict whether *freed[0]* is true or false. So instead, *var0* and *var1* have been put in array *var01_addresses*. Both *var0* and var1 get allocated. Depending on the value of *free_index*, either of the two gets freed. Which variable is freed is also tracked in array *freed*. Then, *attack_func()* will request memory of the same size, and will consequently get the memory that was just freed. There is an inconsequential for-loop that allocates values to variable *x*, this loop only exists to overwrite the older branch history with new entries. The branch history has limited space, and will store the most recent branch decisions. By artificially adding recent branch decisions, possible older ones that the branch predictor would otherwise use are no longer in the history. Doing so positively affects the training of the branch predictor, and ultimately the accuracy of the proof-of-concept. Afterwards, the array *freed* that tracks whether *var0* or *var1* has been freed is flushed from memory. Then, architecturally, only if *var0* has not been freed will it be used as an index for *SECRET*.

However, the branch predictor can be trained beforehand to predict that the branch should be taken. Then, *var0* will speculatively be used as an index for *SECRET*. This character will then be leaked through the flush+reload covert channel. This will be repeated for every character of *SECRET*, until the entire string is leaked.

5.6 Uninitialized Read

When a variable is allocated in C, its value is undefined until one is given. In practice, its value will be the value that resided there before allocation. As a result, it could be a value of a previously used variable that has gone out-of-scope. This way, the value of a previously used variable can be leaked. For reference, see Listing 8. This architectural uninitialized read vulnerability allocated 1 byte that contains a character of *SECRET* through *touch_secret()*. Notice that *touch_secret()* does not return any value, so at first glance this operation seems safe. However, *uninit_func()* also allocates 1 byte. Since *s* has already gone out of scope, *uninit* will be given the same memory address. Because *uninit* is not initialized, its value will be the old value of *s*.

Architecturally, this vulnerability could be mitigated by initializing *uninit* in *uninit_func()* as suggested in the comment in Listing 8. Alternatively, it could be mitigated by calling *init_func('i')* before calling *uninit_func()*, as suggested in the other comment. *init_func()* also allocates 1 byte for *init*, and therefore gets the memory that *s* occupied before. Because *init_func()* initializes the address with a value, *uninit_func()* can no longer read the value of *s*.

Speculatively, it is still possible that an uninitialized read vulnerability occurs in code in which it is architecturally impossible. See *victim_func()* in Listing 9. If *init_bool* is true, it calls *touch_secret()* and later calls *init_func()*. When *init_bool* is false, it does call the dangerous function *uninit_func()*. But since *touch_secret()* is not called it seems *SECRET* is not at risk of being leaked.

However, the branch predictor can be trained to predict calling *uninit_func()*. In the attack phase, *init_bool* is set to true. Then, *touch_secret()* will be architecturally executed. Another inconsequential for-loop is used to overwrite the branch history, to prevent the branch predictor from taking the branch that called *touch_secret()*

```c
#define PAGE_SIZE   4096
#define SECRET      "mysecret"
#define false       0
#define true        1
cp_t* flush_reload_arr;

void* attack_func(int secret_index) {
    int* var0_dupe = malloc(PAGE_SIZE);
    *var0_dupe = secret_index;
    return var0_dupe;
}
void victim_func(int free_index, int secret_index) {

    int* var0 = calloc(1, PAGE_SIZE);
    int* var1 = calloc(1, PAGE_SIZE);

    char freed[n_vars];
    freed[0] = false;
    freed[1] = false;

    if(free_index==0) {
        free(var0);
        freed[0] = true;
    }
    else {
        free(var1);
        freed[1] = true;
    }

    int* var0_dupe = attack_func(secret_index);

    for(int i = 0; i < 100; i++) {if(i%2==0)
        {volatile int x = 0;} else {volatile int x = 1;}}

    cpuid();
    flush(&freed[0]);
    cpuid();

    if(!freed[0]) {
        volatile cp_t cp;
        cp = flush_reload_arr[SECRET[*var0]];
    }

    cpuid();
    if(!freed[0]) free(var0);
    if(!freed[1]) free(var1);
    free(var0_dupe);
}
```

Listing 7: Architecturally safe code with a speculative use-after-free vulnerability.

into account later. *init_bool* has to be copied into *init_bool_copy*, a variable aligned to the size of a cache line (alignment attribute is left out of the listing as an abstraction). We are not sure why exactly, but it likely has to do with the fact that *flush()* could otherwise flush something important that would then reside on the same cache line. Since *init_bool_copy* is flushed, a trained branch predictor will speculatively execute *uninit_func()*, despite *init_bool* being true and *touch_secret()* having executed. As a result, *uninit* gets the old address of *s* allocated and leaks a character of *SECRET*.

```c
#define SECRET      "mysecret"

void touch_secret(int secret_index) {
    volatile char s = SECRET[secret_index];
}
void init_func(int val) {
    volatile char init = val;
}
char uninit_func() {
    volatile char uninit;
    //uninit = 'u';
    return uninit;
}
char victim_func(int secret_index) {
    touch_secret(secret_index);
    //init_func('i');
    return uninit_func();
}
```

Listing 8: An example of an uninitialized read vulnerability.

```c
#define PAGE_SIZE   4096
#define SECRET      "mysecret"
cp_t*   flush_reload_arr;
volatile cp_t cp;

void touch_secret(int secret_index) {
    char s = SECRET[secret_index];
}
void init_func(int val) {
    char init = val;
}
void uninit_func() {
    char uninit;
    cp = flush_reload_arr[uninit];
}
void victim_func(char init_bool, int secret_index) {
    if(init_bool) touch_secret(secret_index);

    for(int i = 0; i < 100; i++) {if(i%2==0)
        {volatile int x = 0;} else {volatile int x = 1;}}

    int init_bool_copy = init_bool;
    flush(&init_bool_copy);
    cpuid();

    if(init_bool_copy) init_func(5);
    else               uninit_func();
}
```

Listing 9: Architecturally safe code with a speculative uninitialized read vulnerability.

5.7 Mutex-lock-bypass

Locks are used in situations where other threads or pieces of code should not interfere with certain variables. In this case, the architectural vulnerability is a lack of locks. In Listing 10, *victim_func()* creates *lock_var* and uses it as an index of *SECRET*. However, before *lock_var* is used as an index, *attack_func()* overwrites its value with *secret_index*. As a result, *SECRET* can be leaked character by character.

The architectural vulnerability can be mitigated by introducing locks. An example of this can be shown by uncommenting the

```c
#define SECRET      "mysecret"
#define FLAG        PTHREAD_MUTEX_DEFAULT

void attack_func(/*pthread_mutex_t* lock_ptr,*/
                 int* lock_var_ptr, int secret_index) {
    //if(pthread_mutex_trylock(lock_ptr) != 0) return;
    *lock_var_ptr = secret_index;
    //pthread_mutex_unlock(lock_ptr);
}
char victim_func(int secret_index) {
    //pthread_mutex_t lock;
    //pthread_mutex_init(&lock, FLAG);
    int lock_var;

    //pthread_mutex_lock(&lock);
    lock_var = 0;

    attack_func(/*&lock,*/ &lock_var, secret_index);
    char s = SECRET[lock_var];
    //pthread_mutex_unlock(&lock);
    //pthread_mutex_destroy(&lock);
    return s;
}
```

Listing 10: An example of a race condition vulnerability.

comments in Listing 10. In this case, *attack_func()* is not able to lock *lock*, and hence has no control over the value of *lock_var*.

Yet this lock mitigation can be bypassed with speculative execution. Like any other code, mutex locks are subject to speculative execution and can be speculatively bypassed.

In Listing 11, *victim_func()* creates 2 locks. If *lock_index* is 0 then *lock0* will be locked, and if *lock_index* is 1 then *lock1* will be locked. It also creates *lock0_var* and *lock1_var*, these variables should only be accessed when the respective lock is acquired. In the real proof-of-concept, this is done without branches. However, the listing uses branches for the sake of readability. *Victim_func()* flushes *lock0* from memory, and *attack_func()* is called. *Attack_func()* tries to acquire the lock, overwrite *lock_val*, unlock, and then leak a character from *SECRET*. However, when it fails to acquire the lock, the function returns. This proof-of-concept shows that both *pthread_mutex_trylock()* as well as *pthread_mutex_unlock()* can be bypassed speculatively. A more realistic proof-of-concept would place the leaking of a character from *SECRET* before unlocking, but in that case it would not show that *pthread_mutex_unlock()* was speculatively bypassed too.

5.7.1 *Semaphore bypass.* The GitHub repository [14] also presents a semaphore bypass. This proof-of-concept is almost identical to the mutex-lock bypass, but the locks functions are replaced by semaphore equivalents. Think of *pthread_mutex_trylock()* being replaced with *sem_trywait()*. Due to the two proof-of-concepts being so similar, it will not be extensively discussed. The rest of this paper will also not consider it as its own proof-of-concept. In terms of evaluation, the metrics were also almost identical to the mutex-lock-bypass proof-of-concept.

5.8 Implementation Difficulties

There were multiple difficulties that could be interesting for a reader considering to reproduce these or similar proof-of-concepts. Firstly, it is important to realize that a flush operation flushes the entire

```c
#define SECRET      "mysecret"
#define FLAG PTHREAD_MUTEX_DEFAULT
cp_t* flush_reload_arr;

void attack_func(pthread_mutex_t* lock_ptr,
                 int* lock_val_ptr, int secret_index) {

    if(pthread_mutex_trylock(lock_ptr) != 0) return;
    *lock_val_ptr = secret_index;
    pthread_mutex_unlock(lock_ptr);
    cp_t cp = flush_reload_arr[SECRET[*lock_val_ptr]];
}
void victim_func(int lock_index, int secret_index) {
    pthread_mutex_t lock0, lock1;
    pthread_mutex_init(&lock0, FLAG);
    pthread_mutex_init(&lock1, FLAG);

    int lock0_var, lock1_var;

    if(lock_index == 0) {
        pthread_mutex_lock(&lock0);
        lock0_var = 0;
    }
    else {
        pthread_mutex_lock(&lock1);
        lock1_var = 0;
    }

    cpuid();
    flush(&lock0);
    cpuid();
    attack_func(&lock0, &lock0_var, secret_index);

    cpuid();

    if  (lock_index == 0) pthread_mutex_unlock(&lock0);
    else                  pthread_mutex_unlock(&lock1);
    pthread_mutex_destroy(&lock0);
    pthread_mutex_destroy(&lock1);
}
```

Listing 11: Architecturally correct and safe usage of locks. However, speculatively the locks can be bypassed.

cache line (64 bytes on the architecture used). When an individual variable is smaller than that, adjacent variables will also be flushed from memory. Another difficulty was aligning variables in certain ways on the stack (e.g. for a buffer overflow), because the compiler might see a less intuitive alignment as safer or more efficient.

A rather minor but interesting find in the bounds-check-bypass proof-of-concept (5.3) is about the addresses of the constants. Whichever of the constants *BUF* and *SECRET* is put above the other in code, and whatever size their assigned strings are, *SECRET* gets placed at a higher memory address than *BUF*. This is convenient for the buffer overread, but it is interesting that *SECRET* never gets placed before *BUF*.

In the buffer overflow proof-of-concept (5.4), for unknown reasons *flush_reload_arr* had to be a function parameter. When it was used as a global variable like in the other proof-of-concepts, the accuracy decreased significantly (to a total accuracy of approximately 66%). This might have to with *flush_reload_arr* residing on the same cache line as the flushed *buf_size*. Yet even when both variables are declared with the memory alignment of a cache page, the problem persists. This would suggest there is a different cause, but as of now we do not know why.

For the use-after-free proof-of-concept (5.5), the allocation size was set to the size of a memory page. However, we found that the size of half a memory page (2048 bytes) worked equally fine. While using the size of a memory page for *malloc()* giving different behavior seems intuitive, we cannot explain why the same holds true for half a memory page.

A very peculiar find in the uninitialized read proof-of-concept (5.6), is that the memory of *init_bool_copy* could be aligned from a value of 32 bytes or more. In many other cases (and as the theory would suggest), this value should actually be the size of a cache line (64 bytes). It is possible that, by chance, the compiler placed a variable on the cache line of *init_bool_copy* that was not essential for the proof-of-concept, but we do not know this with certainty. Another important consideration is that the *volatile cp_t* variable that the covert channel writes to has to be declared outside of *uninit_func()*. This ensures that the stack of that function remains the same as *init_func()*, which is necessary for variable *uninit* to be allocated the same address as *s*. Lastly, one some executions (loosely estimated to be 5% of the time), the uninitialized read proof-of-concept crashed with a exit status 139 (segmentation fault). We are currently not sure why this problem occurs. Furthermore, when running the executable with *gdb* the problem no longer occurs.

In terms of the pthread lock proof-of-concept (5.7), we initially forgot to compile with the *-lpthread* flag. Surprisingly, this did not result in any warnings or crashes. Instead, the trylock and unlock calls were simply ignored. An attempt was made to provide a mutex lock proof-of-concept using multiple threads. However, we were unsuccessful in doing so. The main obstacle was that anything leaked into the F+R array before calling *pthread_join()* seemed to have disappeared from cache. A mutex lock proof-of-concept based on multiple threads would certainly be a good future research project.

6 EVALUATION

All proof-of-concepts were repeated numerous times, after which the results were used to calculate metrics. When a proof-of-concept is executed to leak one byte, we define this as a character leakage attempt. When this character leakage attempt is repeated multiple times to leak the entire secret string, this is defined as a string leakage attempt. Due to noise, the entire string will be leaked multiple times. We define all repetitions of a string leakage attempt as an experiment. The experiment is repeated multiple times by a script, and in between each experiment execution the code is recompiled. This entire process is defined as a proof-of-concept analysis.

Before an experiment is starts, a timer starts. Then, the string leakage attempt is repeated multiple times. Upon completion, the timer stops. The timing function used is *clock()* from the *time.h* library. Its accuracy is defined by *CLOCKS_PER_SEC*, which was set to 1,000,000 on the used system. This means that the precision of the measurement is accurate down one-millionth of a second (or one microsecond).

Out of all calculated metrics, the most interesting were:

⋆ **Total accuracy**: the number of attempts to leak a byte that were correct.

⋆ **Leakage rate**: the total number of successful attempts to leak a byte divided by the time the proof-of-concept analysis took.

For the data presented in Table 1, the experiment was repeated 9 times, during which the string leakage attempt for each proof-of-concept was repeated 10,000 times. The secret to be leaked was set to “mysecret” (note that C strings end in a null-byte, meaning its size is 9). This means that there were 810,000 (9 * 10,000 * 9) attempts to leak a byte per experiment.

Results Intel i7-6700HQ	***total accuracy***	***leakage rate***
Bounds-check-bypass	99.9377%	1537.7041 B/s
Buffer overflow	99.8714%	1529.4918 B/s
Use-after-free	99.9452%	1515.1298 B/s
Uninitialized read	99.6161%	1514.8945 B/s
Mutex-lock-bypass	98.8337%	1522.2698 B/s

Table 1: ***Listing the accuracy, time per byte and leakage rate of the experiments. Results of I7-6700HQ***

From Table 1, it can be noticed that the accuracy and performance of the proof-of-concepts are extremely similar. The different leakage rates are especially close together. A possible explanation for this is that the overhead of the proof-of-concepts is much higher than their own execution times. This would then result in the proof-of-concepts themselves having a relatively small influence on the total measured time. In terms of accuracy, the mutex-lock-bypass sticks out for having a slightly lower accuracy than the other proof-of-concepts. A possibility is that a pthread lock has multiple branches to be bypassed speculatively, while the other proof-of-concepts only had very few. This might make it harder to bypass a lock speculatively. Another explanation is that bypassing *trylock()* and *unlock()* sometimes takes longer than the speculative window duration. However, its accuracy is very high regardless.

7 DISCUSSION

The proof-of-concepts were tested on an Intel i7-6700HQ, on a Lenovo IdeaPad Y700-15ISK running Ubuntu 20.04.6 LTS. The used compiler was *gcc* with *CFLAGS -g -O0*. A strong limitation is that there is no guarantee that these proof-of-concepts work on other architectures. However, Spectre vulnerabilities do exist on practically all modern CPUs. Hence, it can be expected that on architectures on which the proof-of-concepts do not work, only minor tweaks would be required. Think of different cache hit thresholds, more training, or different alignment of memory addresses. This is because a different machine might have different cache speeds, or a different compiler might allcoate variables on different addresses. However, it is also possible that some or all proof-of-concepts don’t work on a machine. This could be because of certain mitigations in place that prevent the particular working of a vulnerability. An example of this is SSBD (speculative store bypass disable) [9]. This mitigation should disable speculative stores, which should mitigate the speculative buffer overflow vulnerability.

The full evaluation process was also done on an Intel i5-7500 and an Intel i5-850. The i5-7500 had very similar results, which likely has due to this model only being a year and a half newer than the i7-6700HQ that the proof-of-concepts were originally designed on. This suggests that the architectures are relatively similar. The much older i5-850 performed similar on some proof-of-concepts, but the accuracies were lower for the bounds-check-bypass and particularly the mutex-lock-bypass. See Table 2 and Table 3 in Appendix A for the data. Limited testing was also done on an AMD Ryzen 7 5800h, on which the proof-of-concepts did not work at all.

Another limitation is that this paper only includes 5 proof-of-concepts. No claim is made that there are not more architectural vulnerabilities with speculative equivalents. Furthermore, these proof-of-concepts are limited by the fact that they are synthetic. A working proof-of-concept shows that vulnerabilities like the ones shown are possible. Further research would be required to find out how common these vulnerabilities are in real systems.

This paper leaves room for plenty of future research projects. As one might expect, there are more vulnerabilities that could be transferred into the speculative realm. Concurrency proof-of-concepts are especially interesting, because not much research about them has been done. Similar proof-of-concepts could also be designed of a different classification than Spectre V1. Lastly, mitigations against these and other proof-of-concepts could be presented.

8 RELATED WORK

The Spectre vulnerability was discovered by P. Kocher et al. [8]. They reported the exploit possibility of conditional branches (also known as Spectre V1), indirect branches (also known as Branch Target Injection, Spectre V2), and the Meltdown vulnerability. They demonstrated leakage of host memory inside a Linux guest VM.

Since its discovery, Spectre vulnerabilities have not been mitigated as a whole. Mitigations for specific Spectre vulnerabilities include: Hardware mitigations, microcode patches, serialization, microarchitecture buffer flushing and introducing data dependencies. However, none of these are satisfiable and/or comprehensive mitigations against Spectre as a whole [13]. One simple and comprehensive mitigation would be to disable speculative execution altogether. However, this mitigation is rarely applied due to the high performance cost [7, 8].

Since the original Spectre paper of P. Kocher et al. [8], much research has gone into different speculative execution vulnerabilities [3, 5, 7, 13]. However, we have not been able to find other papers focussing on multiple synthetic proof-of-concepts like this paper does. The original Spectre paper of P. Kocher et a. [8] already presented a very rudimentary proof-of-concept of a bounds-check-bypass (see Listing 1). The code basically entailed an if-statement checking whether the index was within bounds. Within this if-statement, this index was used on an array (the secret), the resulting value of which was used as the index for another array (the covert channel). A similar proof-of-concept of bounds-check-bypass was presented by V. Kiriansky et al. [3].

V. Kiriansky et al. [3] also presented a buffer overflow proof-of-concept. It basically entailed a bounds check on variable y, after which a value was written into *array[y]* (out-of-bounds). However, this paper coins it as Spectre V1.1: Bounds-check-bypass on stores. The paper discusses how such a speculative buffer overflow could overwrite the return address, causing speculative ROP. Among others, they propose the solution to disable store-to-load forwarding, which is required for speculative buffer overflow vulnerabilities.

We have not been able to find papers presenting speculative use-after-free proof-of-concepts. However, KASPER [7] is a speculative execution gadget scanner that does take speculative use-after-free vulnerabilities into account, as well as out-of-bounds accesses. It was used to find 1379 unmitigated gadgets in the Linux kernel.

We also have not been able to find papers presenting speculative uninitialized read vulnerabilities. The paper presenting KASPER[7] does state that uninitialized reads commonly occur during speculative execution and that an attacker could place malicious data on the stack intending for it to be speculatively read.

To the best of our knowledge, at this time there are no published works discussing the speculative bypassing of locks. However, VUSec does ongoing research into speculative concurrency bug exploitation in the operating system kernel.

This paper shows that both programmers and mitigations have to take more general and complicated Spectre V1 vulnerabilities into account other than just bounds-check-bypass. Furthermore, more research into more complex and advanced gadget scanners such as KASPER [7] is important. This way, more Spectre variations and vulnerabilities can be found and mitigated.

9 CONCLUSION

To conclude, this paper presents 5 Spectre proof-of-concepts of standard software vulnerabilities. It is generally well accepted that these vulnerability types can work in the speculative realm. Therefore, these proof-of-concepts are of a reproductive nature, instead of presenting new work.

To our current knowledge, there is no published work that discusses the speculative bypassing of pthread mutex locks or similar concurrency-related functions. This concurrency proof-of-concept opens the door to research into finding speculative concurrency bugs in real software. Finding other concurrency proof-of-concepts could also lead to fruitful research.

To answer the research question, all vulnerability types presented in this thesis did transfer into the speculative realm. For the 5 classic software vulnerabilities presented, we can conclude that they do have speculative counterparts which could very well exist in real systems. Note however that not all software vulnerability types were addressed, so a conclusive statement about all vulnerability types cannot be made.

10 ACKNOWLEDGEMENTS

This paper acknowledges the frequent and extensive assistance of daily supervisor Mathé Hertogh, who helped with technical aspects of the project. Equal acknowledgement goes to first supervisor Cristiano Giuffrida, for guiding the scientific and research aspects of this thesis. Furthermore, this paper acknowledges the assistance of the entire VUSec department during group meetings. Finally, the inspiration to try a mutex lock bypass comes from earlier VUSec research, this was not the author's original idea.

REFERENCES

[1] Andriy Berestovskyy. 2018. *spectre-meltdown*. Semihalf. https://github.com/berestovskyy/spectre-meltdown

[2] The MITRE Corporation. 2008. CWE VIEW: Weaknesses in Software Written in C. https://cwe.mitre.org/data/definitions/658.html
[3] Crispin Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *DARPA Information Survivability Conference and Exposition* 2, 119–129 vol.2. https://doi.org/10.1109/DISCEX.2000.821514
[4] Ryan Crosby. 2023. *SpectrePoC*. Unique Micro Design Pty Ltd. https://github.com/crozone/SpectrePoC
[5] Enes Goktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*. the Netherlands. Paper=https://download.vusec.net/papers/blindside_ccs20.pdfWeb=https://www.vusec.net/projects/blindsideCode=https://github.com/vusec/blindsidePress=https://bit.ly/3c4MkhU Pwnie Award for Most Innovative Research.
[6] Intel. 2022. Refined speculative execution terminology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html
[7] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2022. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In *NDSS*. the Netherlands. Paper=https://download.vusec.net/papers/kasper_ndss22.pdfSlides=https://download.vusec.net/slides/kasper_ndss22.pdfWeb=https://www.vusec.net/projects/kasperCode=https://github.com/vusec/kasperVideo=https://www.youtube.com/watch?v=v89Zt3vxrww
[8] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre Attacks: Exploiting Speculative Execution. *Commun. ACM* 63, 7 (jun 2020), 93–101. https://doi.org/10.1145/3399742
[9] Michael Larabel. 2018. Benchmarking the performance impact of speculative store bypass disable for Spectre V4 on Intel core i7. https://www.phoronix.com/review/intel-spectre-ssbd
[10] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William K. Robertson, Engin Kirda, and Anil Kurmus. 2020. Bypassing memory safety mechanisms through speculative control flow hijacks. *CoRR* abs/2003.05503 (2020). arXiv:2003.05503 https://arxiv.org/abs/2003.05503
[11] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*. the Netherlands. Paper=https://download.vusec.net/papers/fpvi-scsb_sec21.pdfWeb=https://www.vusec.net/projects/fpvi-scsbCode=https://github.com/vusec/fpvi-scsb Distinguished Paper Award, Intel Bounty Reward, Mozilla Bounty Reward, Pwnie Award Nomination for Most Innovative Research, Pwnie Award Nomination for Best Privilege Escalation Bug, Pwnie Award Nomination for Best Client-Side Bug, Pwnie Award Nomination for Epic Achievement, DCSR Paper Award, CSAW Best Paper Award Runner-up.
[12] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*. the Netherlands. Paper=https://download.vusec.net/papers/crosstalk_sp21.pdfWeb=https://www.vusec.net/projects/crosstalkCode=https://github.com/vusec/ridlPress=https://bit.ly/3frdRuV Intel Bounty Reward.
[13] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 685–698. https://doi.org/10.1145/3470496.3527429
[14] Chris Wagenaar. 2023. *Specify: Transforming Regular Vulnerabilities into Spectre Vulnerabilities*. Vrije Universiteit Amsterdam. https://github.com/CcjWagenaar/spectify
[15] Sachin Yadav. 2020. *spectre-attack*. Indian Institute of Technology. https://github.com/yadav-sachin/spectre-attack
[16] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 719–732.

Appendix A OTHER ARCHITECTURES

Results Intel i5-7500	***total accuracy***	***leakage rate***
Bounds-check-bypass	99.9623%	1892.2232 B/s
Buffer overflow	99.9622%	1894.5860 B/s
Use-after-free	99.9867%	1877.6777 B/s
Uninitialized read	99.9623%	1859.6732 B/s
Mutex-lock-bypass	99.9644%	1893.1599 B/s

Table 2: *Listing the accuracy, time per byte and leakage rate of the experiments. Results of Intel I5-7500.*

Results Intel i5-750	***total accuracy***	***leakage rate***
Bounds-check-bypass	88.8104%	1698.8801 B/s
Buffer overflow	99.3711%	1901.2421 B/s
Use-after-free	99.5799%	1879.7298 B/s
Uninitialized read	96.0949%	1818.3996 B/s
Mutex-lock-bypass	48.3289%	920.2888 B/s

Table 3: *Listing the accuracy, time per byte and leakage rate of the experiments. Results of Intel I5-850.*