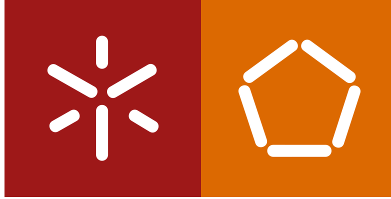
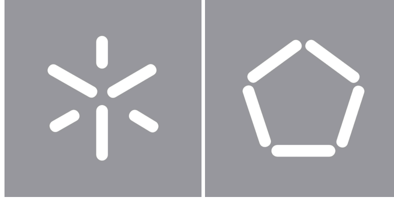


University of Minho
School of Engineering

Ana Luísa Lira Tomé Carneiro

Speculative Execution Resilient Cryptography

october 2023



University of Minho
School of Engineering

Ana Luísa Lira Tomé Carneiro

Speculative Execution Resilient Cryptography

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
José Carlos Bacelar Ferreira Junqueira Almeida

october 2023

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:

CC BY

https://creativecommons.org/licenses/by/4.0/

Acknowledgements

I would like to express my gratitude to all those who have supported and guided me throughout this research. This dissertation project was only possible because of the the help and encouragement of the following individuals and institutions.

To my supervisor José Carlos Bacelar Almeida for his support, expertise, and guidance. His knowledge and wisdom played a vital role in shaping the direction of this project. To professor Bjorn De Sutter from Ghent University for his assistance and skilfulness in following up this project while I was in my Erasmus exchange program. Furthermore, I wish to express my appreciation to Ghent University to allow the usage of their remote machines that played an integral role in facilitating my research.

Finally, I am grateful to my friends and fellow colleagues who provided moral support, and moments of relief from the hardships of academic life. To my family for their love and encouragement that sustained me during these years. Their friendship and union made this journey memorable.

I am fortunate to have had such a strong network of support and I offer my sincerest appreciation for your contributions to my academic and personal growth.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, october 2023

Ana Luísa Lira Tomé Carneiro

Abstract

Spectre attacks pose a significant threat to modern computer systems, exploiting speculative execution to leak sensitive information from a program. Since speculative execution is present in modern CPU it is of high priority to protect programs against these spectre attacks without reducing performance.

This study presents a comprehensive comparison of mitigation strategies employed by different tools to counteract the effects of spectre attacks. However, this paper focuses its analyses on the type system from the Jasmin framework and on the Blade tool. The type system uses three main primitives that work together to protect vulnerable variables from leaking. On the other hand, the Blade tool allows for an automatic mitigation strategy that implements a Min-Cut algorithm to a graph representing the different outputs of a program in order to find the minimal cut points that stop speculative execution from leaking vulnerable variables.

Aside from these strategies, this study also presents in detail the oo7 tool that identifies spectre vulnerable patterns which are used to easily identify if a program is vulnerable to spectre attacks.

Through an in-depth analysis of their techniques, performance implications, and applicability, this research evaluates the suitability of both strategies to protect cryptography functions against spectre attacks by protecting the Blake2b hash function. In the end, this comparative analysis between these two mitigation strategies determines in which scenario or purpose which technique should be used.

Keywords Spectre attacks, Jasmin, type system, Blade, hash Blake2b, oo7, speculative execution

Resumo

Os ataques *spectre* representam uma ameaça significativa para os todos os computadores modernos, já que exploram a execução especulativa para obter informações sensíveis de um programa. Uma vez que a execução especulativa está presente em todos os CPUs modernos, é de alta prioridade proteger programas contra estes ataques sem reduzir a performance de um programa.

Este estudo apresenta uma comparação compreensível das estratégias de mitigação realizadas por diferentes ferramentas para combater os efeitos dos ataques *spectre*. No entanto, este artigo concentra a sua análise no *type system* da *framework* Jasmin e na ferramenta Blade. O *type system* utiliza três primitivas principais que trabalham juntas para proteger variáveis vulneráveis de ser lidas por entidades externas. Por outro lado, a ferramenta Blade fornece uma estratégia automática de mitigação que implementa o algoritmo Min-Cut num gráfico que representa os diferentes outputs de um programa, a fim de encontrar os cortes mínimos que impeçam a execução especulativa de ler variáveis vulneráveis.

Além destas estratégias, este estudo também apresenta em detalhe a ferramenta oo7, que identifica padrões vulneráveis a ataques *spectre* que são usados para identificar facilmente se um programa é vulnerável a ataques *spectre*.

Através de uma análise aprofundada das técnicas, implicações de performance e aplicabilidade, esta pesquisa avalia o quão adequadas são as estratégias em proteger funções de criptografia, através da proteção da função de *hash*, Blake2b. No final, esta análise comparativa irá determina em qual cenário ou propósito cada estratégia de mitigação deve ser usada.

Palavras-chave ataques *spectre*, Jasmin, *type system*, Blade, *hash* Blake2b, oo7, execução especulativa

Contents

I Dissertation Chapters 1

1 Introduction 2

1.1 Contextualisation . . . 2
1.2 Motivation . . . 4
1.3 Purpose . . . 5
1.4 Expected Results . . . 6

2 State of the Art 7

2.1 Spectre-type Attacks . . . 7
2.1.1 Spectre-PHT . . . 7
2.1.2 Spectre-BTB . . . 8
2.1.3 Spectre-RSB . . . 8
2.1.4 Spectre-STL . . . 9
2.2 Defences against Spectre Attacks . . . 9
2.2.1 Pitchfork . . . 9
2.2.2 Kaibyo . . . 10
2.2.3 oo7 . . . 11
2.2.4 SpecuSym . . . 14
2.2.5 Venkman . . . 15
2.2.6 Blade . . . 16
2.3 The Jasmin type system . . . 19
2.3.1 Jasmin . . . 19
2.3.2 Type System . . . 25

3 Use case 29

3.1 Research Overview 29
3.2 Blake2 Library 30
3.2.1 Blake 30
3.2.2 Blake2 Algorithm 31

4 Spectre Analysis **33**
4.1 Detecting Vulnerabilities 33
4.1.1 Reference Code 33
4.1.2 Jasmin Code 42
4.2 Mitigating Vulnerabilities 44
4.2.1 Reference Code 44
4.2.2 Jasmin Code 49

5 Evaluation and Comparison of Results **52**
5.1 Comparison criteria 52
5.1.1 Availability and Installation Process 52
5.1.2 Level of Knowledge 53
5.1.3 Mitigation customisation 54
5.1.4 Mitigation Strategy 54
5.1.5 Efficiency and Performance 55
5.1.6 Security 56
5.2 Comparative analysis 57

6 Conclusions and future work **58**
6.1 Conclusion 58
6.2 Prospect for future work 59
6.2.1 Analyse the addition of SLH instructions 59
6.2.2 Analyse the WebAssembly version of the library 60
6.2.3 Analyse other tools and libraries 60

II Appendices **63**

A Listings **64**
A.1 Blake2b - Reference Code 64

A.2 Blake2b - Reference Code - Blade 70
A.3 Blake2b - Jasmin Code 74

B Tooling **84**

B.1 oo7 - Dockerfile 84
B.2 Blade - Dockerfile 85
B.3 Blade - Makefile to add fence instructions 87
B.4 Blade - Measure Performance - Reference Code 88
B.4.1 Main 88
B.4.2 Blake2b 89

List of Figures

1 oo7 Framework - vulnerability detection module. Adapted from Wang et al. [2021] . . . 12
2 Subset of the def-use graph. Adapted from Vassena et al. [2020] 18
3 Jasmin declaration of variables 20
4 Jasmin Instructions. Taken from Oliveira [2022] 23
5 BAP output for test program 35
6 Part of the assembly for the test program 36
7 oo7 Report for the test program 37
8 oo7 Memory Consumption 39
9 BAP report for the Blake2b library 40
10 Vulnerability in the `blake2b_final` function 41
11 Vulnerability in the `blake2b_update` function 42
12 Output from `-checkSCT` flag 43
13 Mitigation for spectre vulnerabilities using the lfence approach by the Blade tool 46
14 Mitigation for spectre vulnerabilities found in `blake2b_update` function 47
15 Vulnerable code snippets from the `blake2b_final` function 48
16 Mitigation for spectre vulnerabilities found in `blake2b_final` function 49
17 Output from `-checkSCT` flag after the mitigation 51

List of Tables

1 Access data from a pointer in Jasmin 21

2 Arithmetic operators overview. Taken from Oliveira [2022] 22

3 Bitwise and shift operators overview. Taken from Oliveira [2022] 23

4 Parameters for the Blake2 algorithm 32

5 Performance measurements for Jasmin Blake2b version 55

6 Performance measurements for Blade Blake2b version 56

7 Summary of the criteria results 57

Part I

Dissertation Chapters

Chapter 1

Introduction

This chapter introduces the main concepts and topics on the study to be developed. It starts by contextualising fundamental ideas about speculative execution and spectre attacks. Next, it explores the motivation behind this study, followed by a description of the objectives in each stage. Finally, it ponders over the expected results and conclusions from this thesis.

1.1 Contextualisation

In the last decades, the evolution of CPUs (*Central Processing Unit*) was driven by many investigators and engineers to improve performance. Numerous physical advancements have been found to enhance processing power and improve performance, including minimising processing technology and increasing clock frequencies. However, physical improvements were hampered by physical limitations hence many vendors of CPUs shifted their focus to increasing the number of cores in each CPU and optimising the instruction pipeline. Therefore, modern CPUs are massively parallelised allowing hardware to perform operations for subsequent instructions ahead of time or even out-of-order.

In out-of-order execution, complex instructions are first split up into micro-operations which are then processed by the CPU in sequential order, provided by the instruction stream, but dispatching them in parallel. Consequently, if an operand required for a micro-operation is available then the CPU will process this micro-operation even if previous ones in the instruction stream have not finished yet. To keep track of the states of these operations, CPUs use the Reorder Buffer (ROB) that allows them to discard micro-operation results when, for example, an exception is thrown. When all micro-operations are complete then the results are committed into the architecture state, freeing the space in the ROB buffer. When a micro-operation whose result is never committed to the architecture state, then the instruction is called transient.

Although out-of-order execution utilises the CPUs execution unit as much as possible and, thus, im-

proves the overall performance, it also has its downsizes. When operations have a dependency on a previous instruction that has not been executed yet, then the instruction pipeline stalls until the previous instruction finishes. Hence, to keep the pipeline running at all times, it is fundamental to predict the control flow and data dependencies. This approach is implemented in CPU through **speculative execution**, where a processor can make a prediction as to the path that the program will follow and speculatively execute instructions along that path. If the prediction is correct then the CPU commits the results into the architecture state and if the prediction is wrong then the CPU discards the changes made. This mechanism is more advantageous than idling when waiting for an instruction to finish because the prediction is made according to previous path outcomes, so the probability of a prediction being correct is high.

Another downsize of out-of-order execution is the existence of transient executions, since they reflect unauthorised computations that were never supposed to happen, due to exceptions, mispredictions, or interrupt requests. When an execution is discarded from the ROB, it may leave traces in microarchitectural covert channels, such as cache, that could later be recovered by an attacker. This is the foundation for transient attacks, where an attacker can exploit out-of-order and speculative execution to access private data from the victim's code.

According to Canella et al. [2019] there are two types of transient attacks, Spectre and Meltdown. Spectre attacks exploit transient execution following control or data flow misprediction and Meltdown attacks rely on transient execution following a CPU exception. Essentially, Meltdown exploits the CPU capacity to implement out-of-order execution, while Spectre attacks rely on the use of speculative execution to steer the victim into transiently computing on private memory locations. In this study, we will focus our attention on spectre attacks.

Spectre attacks are built upon three phases as stated by Kocher et al. [2019]. In the first phase, the attacker starts by mistraining the microarchitectural branch predictor to cause intentional misspeculation of a particular victim's branch. In the second phase, the processor executes an instruction that was mispredicted allowing to transfer confidential data to the cache, to then realise that it occur an erroneous speculative prediction, discarding the outcome of the execution. In the last phase, the attacker uses Flush+Reload or Evict+Reload attack to recover the confidential data that was transferred to the cache, by timing the access to memory addresses in monitored cache lines.

Note that the Flush+Reload and Evict+Reload are microarchitectural side-channel attacks that enable the attacker to monitor cache lines by measuring the time that it takes to perform a memory access. If the access is quick then the victim accessed the monitored cache line, otherwise, the access will be slow. The main difference between these attacks is that Flush+Reload starts by flushing all cache lines using

`clflush`, while Evict+Reload starts the attack by evicting all cache lines.

Following Canella et al. [2019], there are four types of spectre attacks that have been categorized based on the root cause that triggered the misprediction and the method used to mistrain the branch predictor. One type is the Spectre-PHT or Variant 1 which exploits the Pattern History Table (PHT) to mispredict the direction of conditional branches. This attack also poisons the Branch History Buffer (BHB) that accumulates the behaviour of the last N branches, because PHT uses BHB to predict which direction to take when arriving at a conditional branch. Even though our focus in this study will be the Spectre-PHT attacks, the other three types of spectre attacks are the Spectre-BTB or Variant 2 that poisons the Branch Target Buffer (BTB) to allow the attacker to mispredict an indirect jump to other snippets of code in the victim's memory (gadgets), the Spectre-RSB which exploits the Return Stack Buffer (RSB), allowing the attacker to overwrite return addresses and finally, the Spectre-STL that exploits memory disambiguation for predicting Store-to-Load (STL) data dependencies which requires a memory load to not be executed before all preceding stores, that write on the same location, have completed.

1.2 Motivation

Following the discussion of the previous section, speculative execution and transient computations violate security mechanisms and assumptions allowing for CPUs to be vulnerable to spectre attacks. These attacks represent a serious threat to current systems since vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM, affecting billions of devices. The work developed by Canella et al. [2019] concluded that microprocessors from these vendors are vulnerable to spectre-PHT attacks. For this reason, it is essential to think of new methods to mitigate Spectre-PHT attacks[1] as they impose a significant hazard to modern CPUs.

Considering that speculative execution and transient computations were implemented in CPUs in order to improve their performance, it is important that implementing solutions to mitigate spectre attacks does not hinder processing. According to Kocher et al. [2019], one way to easily mitigate vulnerable speculative executions and, thus, mitigate spectre attacks is to ensure that conditional branches are executed sequentially by adding `lfence` instructions on the two outcomes of every conditional branch. This approach was recommended by Intel and AMD and guarantees that the speculative execution will be disabled between the `lfence` instructions. However, excessive usage of such a mechanism imposes a considerable toll on performance, making this method not feasible to mitigate spectre attacks.

[1] Throughout the rest of this chapter the Spectre-PHT attacks are going to be referred to as spectre attacks

Since current devices have spectre vulnerabilities, these attacks can jeopardise many programs that run on microprocessors. These programs and implementations may access or store in memory confidential and private data, like secret keys, certificates, and other security elements that if compromised could lead to other attacks and security breaches. Common implementations that deal with such sensitive data are cryptography libraries that are often used to support many implementations for other programs. Given how widespread these attacks and associated vulnerabilities are, it is of key importance to verify if cryptography libraries can withstand such attacks.

1.3 Purpose

Taking into account that solutions to mitigate spectre attacks should not hinder CPU performance and that cryptography libraries, which deal with secret data, are vulnerable to these attacks, it is vital to find solutions that tackle these issues. This was the basis for researchers at the Max Planck Institute to develop a type system Shivakumar et al. [2022] incorporated into the Jasmin framework Almeida et al. [2017] that efficiently allows programmers to write cryptography implementations that are protected against spectre attacks.

Jasmin is a framework that according to Almeida et al. [2017] allows programmers to develop high-speed, high-assurance, and high-security cryptography software. This framework is structured around the Jasmin programming language which supports high-level features, like loops and procedure calls, leading to easily verifiable code and assembly-level instructions to give programmers control over the generated code. This framework also implements a formally verified compiler that was designed to achieve predictability and deliver efficient code by transforming Jasmin code into assembly programs.

Under the Jasmin framework, was created a type system Shivakumar et al. [2022] that protects cryptography Jasmin programs and libraries from spectre attacks by ensuring speculative constant-time, that is, guaranteeing the constant-time policy even during speculative execution. This policy mandates that control flow and memory access be independent of secret data, so it does not leak sensitive data to side-channels, hence, if a program is speculative constant-time its leakage does not depend on secrets, for any branch prediction and unsafe memory accesses.

Due to the spectre vulnerabilities found in many cryptography libraries and the potential risk of sensitive information leakage, this study aims to focus on protecting a cryptography library from such attacks by using mitigation techniques. As the study will involve comparing and analysing the performance and security of the several mitigation techniques, the selected library must already be implemented in Jasmin

and vulnerable to spectre attacks. For this purpose, the chosen library will be Blake2b which implements the cryptography hash function blake2b. In the following chapters, this library and the implementation of this hash function will be discussed in greater detail.

In summary, the main purpose of this study is to verify if the cryptography library Blake2b can resist spectre attacks using mitigation techniques such as the type system. Essentially, this study will analyse the implementation of several mitigation techniques and, in the end, compare them to the type system technique to evaluate if the Jasmin implementation has better performance results than the others. To pursue this goal, this study is organised into three different phases as described below.

The **first phase** is detailed in Chapter 2 and starts by exploring spectre-type attacks and possible mitigation tools and techniques, and by analysing the Jasmin specifications as well as protocols implemented with it. This phase also examines the type system and its typing rules, and characteristics.

The **second phase** is demonstrated in Chapter 4 and begins by finding spectre vulnerabilities in the Blake2b library by using a spectre detection tool. Subsequently, we apply the type system and the other mitigation techniques to this library to mitigate these attacks so that the library can now withstand spectre attacks.

The last and **third phase** shown in Chapter 5 compares and analysis the performance and security results from the mitigation techniques we apply in the previous phase to evaluate if the type system has better results than the other techniques.

1.4 Expected Results

At the end of this study, it is expected to acquire an organised and overall understanding of spectre attacks and ways to mitigate them. It is also expected to gain a detailed view of the Jasmin framework and its type system, and it is anticipated that this study analysis several mitigation techniques, including the type system, by implementing them in the Blake2b library. Finally, it is expected to examine the results of the assorted implementations by comparing them to the results of the type system.

In essence, this study aims to determine which countermeasures should be implemented in the Blake2b library to withstand spectre attacks while ensuring that the performance and efficiency of the library are not compromised.

Chapter 2

State of the Art

This chapter discusses the foundations and theoretical concepts for the rest of the study. It starts by describing spectre-type attacks by giving an overview on each type of attack, then introduces various defences to protect software and devices against these attacks and, finally, explores the Jasmin language including its security type system.

2.1 Spectre-type Attacks

As was discussed in the previous section, spectre attacks showed that branch or data misprediction events might leave secret-dependent traces in the CPU's microarchitectural state. Following Canella et al. [2019] there are 4 types of spectre attacks which are categorised according to the root cause of the misprediction and the method used to exploit these attacks based in how we mistrain the branch predictor.

The following sections describes each type of spectre attack, however spectre-PHT will be discussed in more detailed than the other 3 attacks, as this study will focus on this type of attack.

2.1.1 Spectre-PHT

The Spectre-PHT or Variant 1 exploits conditional branch misprediction to allow an attacker to read arbitrary secret memory from another process, by triggering a speculative execution of a sensitive operation that would not be executed in normal program flow. As an example given by Kocher et al. [2019], the following code snippet is vulnerable to Spectre-PHT attacks (variant 1).

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

The listing begins by checking if the variable `x` is within the bounds of `array1`, which is essential for security purposes. However, after repeatedly supplying valid values of `x`, the PHT will predict this branch

as true. Thus, when the attacker finally supplies an out-of-bounds value of x, the PHT will mispredict this branch as true. When the execution of the condition instruction finishes, the CPU realises that the PHT mispredicted the branch and rolls back the changes made. Nevertheless, microarchitectural changes in the cache are never discarded and, consequently, the attacker can access the value of array1[x] that is confidential. To complete the attack, the attacker measures the time that takes to access the location of y in the cache, using microarchitectural side-channel attacks. This reveals the value of array1[x], which was also cached along with y.

Another example of a spectre-PHT vulnerability can be seen in the snippet below (variant 1.1). Although the code does not store the value of array2[array1[x] * 4096], the value of x is still used as an index into the array1, which may contain sensitive information. An attacker can use a spectre-PHT attack to leak the value of array2[array1[x] * 4096] and then use it to access the corresponding value of array1[x], by timing the access to the cache.

```
1  if (x < array1_size){
2      array2[array1[x] * 4096] = 0}
```

2.1.2 Spectre-BTB

The Spectre-BTB or Variant 2 exploits indirect branch misprediction to allow an attacker to speculatively execute code that should not be executed in normal program flow. An indirect branch is a type of program control instruction that uses a memory variable, that stores the function address, to allow the program to jump to the next context, contrary to a direct branch that uses the function address for this jump. A program might be vulnerable to spectre-BTB attacks as the execution of indirect branches are delayed due to cache misses when obtaining the function address stored in the variable, allowing speculative execution to predict the jump to improve performance. Therefore, it is possible that a misprediction occurs, letting the program to execute a code snippet that might store vulnerable data in the cache and leaking sensitive information through side channels.

2.1.3 Spectre-RSB

The Spectre-RSB exploits the Return Stack Buffer (RSB) to cause speculative execution of a *Spectre gadget* that reads and exposes sensitive data by manipulate the software stack to create a mismatch between this stack and the RSB. The software stack stores the return addresses after a call and the RSB is a per-core microarchitectural buffer used to predict return addresses by pushing these addresses from

a call instruction to the hardware stack. When the return is encountered, the RSB uses the top of the buffer to predict the return address, supporting speculation with very high accuracy. If the RSB predicts a return address to a code snippet that writes vulnerable data to a microarchitectural side channel, then this creates a mismatch between the prediction and the software stack, making the program vulnerable to spectre-RSB attacks.

2.1.4 Spectre-STL

Speculation in modern CPUs is not restricted to control flow but also includes predicting dependencies in the data flow. The Spectre-STL attack exploits Store-to-Load (or RAW) dependencies which refer to an instruction that is dependent on a result that has not yet been calculated or retrieved. As presented in Horn [2018], the Spectre-STL attack start by mistraining the CPU's memory disambiguator, which is a set of techniques that predicts which loads are not dependent on previous stores and, therefore, can be executed speculatively. The mistraining will enable the disambiguator to predict the execution of a load that in reality dependent on a previous store and, consequently, letting the program to read secret data into covert channels, like cache. When the prediction is verified, the load and all succeeding instructions are discarded and re-executed. Finally, the attacker uses a microarchitectural side-channel attack to probe the cache and, thus, access the secret data.

2.2 Defences against Spectre Attacks

As mentioned in the previously spectre attacks violate fundamental assumptions about architectural abstractions, allowing attackers to steal sensitive data. Consequently, implementing defences against spectre attacks is crucial to protect programs and devices from getting their sensitive data exposed to the outside world. In this section, we discuss several countermeasures that reason microarchitectural details such as speculative execution and side-channel attacks. Taking into account the systematisation found in Cauligi et al. [2022] this section introduces a diverse group of verification and mitigation tools, frameworks, and compile-based defences that grants protection against these attacks.

2.2.1 Pitchfork

As presented in Section 2.1, side-channels such as cache could serve as an intermediate carrier through which private data could inadvertently be disclosed to observers by timing the cache visiting latency. One way to mitigate this issue would be to enforce constant-time execution for all operations

that deal with secret data, however, this mitigation may still get compromised by speculative execution. Therefore, to mitigate timing side-channels vulnerabilities it is necessary to maintain constant-time even in speculative execution so that no secret data is leaked through side-channels. To mitigate programs against side-channels attacks during speculative execution, that is, against spectre attacks, it is necessary to develop speculative constant-time programs. A program is speculative constant-time, when it neither branch on secrets nor access memory based on secret data during speculative execution.

On that account, Cauligi et al. [2020] lays foundations for constant-time programming in the presence of speculative execution. It presents operational semantics and a formal new definition of speculative constant-time (SCT) that enables us to discover violations of the constant-time property in programs.

As presented in the paper, given an attacker directive `d` the execution of a program goes from state `C` to state `C'`, producing an observation, that is a leakage. Hence, a program satisfies SCT iff every initial state satisfies SCT under any directives. To build the operational semantics to implement this definition it is necessary that semantics model conditional branching, memory operations, fence instructions, indirect and direct jumps, and function calls. In the end, this semantics captures a variety of existing spectre variants because it shows that these attacks violate the SCT definition by producing observations depending on secrets.

In order to detect SCT violations and data leaks in real cryptography code, it was created the Pitchfork analysis tool. This tool first generates a set of directives representing various worst-case attackers and then checks for secret leakage by symbolically executing the program under each directive. To generate the directives, Pitchfork maintains a limit size buffer that determines the depth of the speculation (speculation bound). For conditional branches and memory operations, Pitchfork constructs directives for all possible outcomes until the buffer is full, that is, the size of the buffer matches the speculation bound. Pitchfork manages to expose attacks based on Spectre-PHT, Spectre-BTB, and Spectre-STL.

In summary, according to Cauligi et al. [2022] Pitchfork is a low-level approach to detect several spectre vulnerabilities in programs by using directives to track the various outcomes of a program and by reasoning about speculation fences and speculation window.

2.2.2 Kaibyo

Just like Pitchfork, de León et al. [2022] uses semantics that takes into account speculation to reliably identify code patterns vulnerable to spectre attacks. However, unlike the semantics used in Pitchfork that describes how valid a program is interpreted, Kaibyo uses axiomatic semantics that defines which executions are valid. To create these semantics Kaibyo uses CAT Alglave et al. [2016], a relational language

used to develop a set of models that describe speculative behaviour and their effect enabling us to capture a variety of attacks in a simple, concise and unified manner.

Following the CAT approach, the paper defines the semantics of a program axiomatically in terms of consistent executions, that is, the execution does not violate the software isolation property by leaking the secret address. To construe the definition of consistent executions, the behaviours of a program are represented by graphs where nodes model occurrences of instructions (events) and edges model dependencies between events. In the end, the graph forms a group of candidate executions that define possible behaviours of a program. Nevertheless, only certain behaviours can occur in practice. To define which behaviours can befall, it is necessary to maintain certain event properties such as control and data flow for the specific program. Therefore, the CAT constraints the candidate executions by inserting different assertions each of them describing concrete properties. With this final graph, we can determine by analysing each candidate execution whether the software isolation property is violated. An attacker breaks the software isolation property when it reads a secret from an address outside its sandbox boundary by tricking the victim into leaking sensitive information.

To analyse the graph created by the CAT model and to verify if a program violates the software isolation property, the paper uses the analysis framework Kaibyo. Kaibyo takes as input a program written in assembly, the CAT model that samples the program properties, an unrolling bound to limit the number of iterations in each loop, and an address that should not violate the software isolation property. Then, it generates a formula based on the CAT model and the unrolling bound which is satisfied iff all the candidate executions are consistent.

In summary, Kaibyo only provides security guarantees up to a given bound and according to Cauligi et al. [2022] is a high-level approach to detect violations of the software isolation property by using CAT memory models to verify if programs are vulnerable to Spectre-PHT and Spectre-STL attacks. Kaibyo also reasons about speculation fences and speculation window by checking over the whole program via a CAT model.

2.2.3 oo7

While the last two approaches use semantics to model program behaviours, Wang et al. [2021] focuses on identifying code patterns on the program binaries that are vulnerable to spectre attacks and patches them with minimal performance overhead. This approach uses the static analysis oo7 to detect potentially vulnerable code in the program's binary and then introduces fence instructions at selected program points to prevent speculative execution and thereby protect the code from spectre attacks. The oo7 tool contains

two main modules, a vulnerability detection module that detects spectre vulnerabilities and a code repair module that mitigates the vulnerabilities found.

The vulnerability detection module as shown in Figure 1, is supported by three technologies: forced execution, taint analysis, and a vulnerability checker. The forced execution constructs a graph that represents all possible control flow edges of a branch, that is, forces the program to execute along all possible outcomes of a branch to simulate different speculative executions and consequently expose the several behaviours of a program. The taint analysis tracks the data and instructions that can be controlled by the attacker. In the taint propagation engine every instruction or data that imports information from un-trusted channels is considered a tainted object. This will be propagated along the path of that instruction or data by applying several taint propagation rules. Finally, the vulnerability checker detects whether the current state satisfies the condition for spectre vulnerabilities, by setting up the Speculative Execution Window (SEW) which will determine the analysis limit of a tainted branch. When the vulnerability checker evaluates instructions inside that branch as vulnerable or not, then the SEW is decremented by one, and so on until the SEW limit is reached.

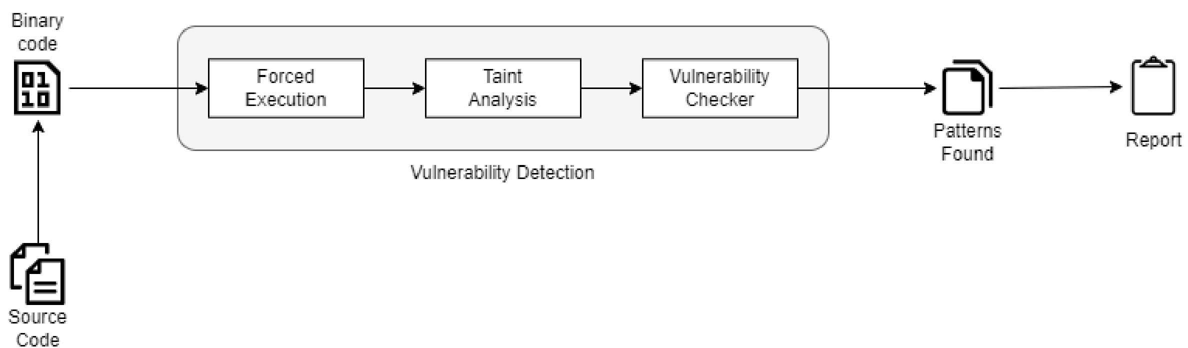


Figure 1: oo7 Framework - vulnerability detection module. Adapted from Wang et al. [2021]

This detection module is implemented using interfaces from a binary analysis platform called BAP. According to Brumley et al. [2011], BAP provides a toolkit for implementing automated binary analysis by including analyses, standard and microexecution interpreter, and a symbolic executor. BAP features its own domain-specific language, Primus Lisp, that is used for implementing analyses, specific verification conditions, model functions, and even an interface with the SMT solver. The microexectution interpreter called Primus will use a personalised code written in Primus Lisp called recipe to then perform the forced execution and the taint analysis as a part of the vulnerability checker. The recipe written in Primus Lisp, in this case, is called spectre and has all the rules to detect spectre-vulnerable patterns in a binary code.

By using interfaces from the BAP toolkit, this detection module starts with creating a graph of all possible outcomes of a program. When the forced execution evaluates a call instruction, the taint analysis

engine checks if the destination belongs to the taint source list which is a set of APIs that can import data from un-trusted sources. Then, after propagating the tainted objects along the graph edges, the vulnerability checker checks if there are any potentially vulnerable code patterns as follows.

The pattern `<TB, RS, LS>` is vulnerable to spectre-PHT attacks and begins with a tainted branch (TB), then reads a variable using a malicious input (read secret - RS) and finishes with memory accesses that is dependent on a secret leading to a cache state change (leak secret - LS). For example, the code from variant 1 found in Section 2.1.1 has this pattern. The condition evaluates a variable `x` that is imported from an un-trusted source making this a tainted branch. Then, it reads a secret variable, `array1[x]`, because it accesses a secret address due to the malicious variable `x` and, finally, reads the variable `y` that is dependent on the secret `array1[x]`.

The pattern `<TB, SW>` is also vulnerable to spectre-PHT attacks and begins with a tainted branch (TB) and then writes data to a private address that is dependent on a malicious variable (speculative write - SW). For example, the code from variant 1.1 found in Section 2.1.1 has this pattern. The condition evaluates a variable `x` that is imported from an un-trusted source making this a tainted branch. It then writes data into `array2[array1[x] * 4096]`, that is dependent on an malicious variable `x`. The attacker can then access the corresponding value of `array1[x]`, by using microarchitecture side-channel attacks.

Following the figure 1, after the detection module finds all the vulnerable patterns in the program, the BAP interface produces a "Pattern Found" report. This report has all the vulnerable patterns found in the program's binary code and helps the oo7 framework to create its own "Report", which details all the BAP findings including a more complete view of the analysis by mentioning all tainted branches and all possible vulnerable patterns found.

The code repair module takes the assembly and disassembly code from the program and the locations of the vulnerable code patterns in the vulnerability detection module to modify the assembly code by inserting memory fence instructions in the appropriate locations. In particular, it inserts memory fences following the TB instructions and immediately before the execution of RS and SW for the vulnerable patterns `<TB, RS, LB>` and `<TB, SW>`, respectively. This solution efficiently inserts a small but necessary number of fences at targeted points in the code without changing the underlying operational system, thus, indicating the practicality of this solution.

In summary, oo7 provides not only a detection tool for spectre-PHT vulnerabilities but also mitigates them with a small overhead. According to Cauligi et al. [2022], oo7 does not use semantics to detect nor mitigate these vulnerabilities but instead uses data flow analysis and reasons about speculation window for that propose.

2.2.4 SpecuSym

As mentioned in Section 2.2.1, the constant-time policy is largely used in real-world cryptographic libraries to eliminate leaks via side-channels, hence developing speculative constant-time programs protects them against spectre attacks.

The study Guo et al. [2020] proposes a new symbolic execution-based method, SpecuSym, for precisely detecting cache timing leaks introduced by speculative execution. The SpecuSym explores the program and models speculative behaviour at conditional branches and accumulates the cache side effects along those paths. After this dynamic execution, SpecuSym conducts a constraint-solving based cache behaviour analysis to generate the leak witnesses, that is, the amount of leakage found along those paths.

Symbolic execution is a systematic program testing and analysis method where interpreting an event, $e := (l_b > inst > l_a)$ stands for the execution of an instruction, $inst$, where l_b and l_a denote the state, $\mathtt{s}$, of a program before and after the instruction, respectively. Hence, a program execution explores a sequence of events along program paths.

To describe this new symbolic execution of SpecuSym, the SpecuSym algorithm starts by checking whether the event of symbolic state $\mathtt{s}$, which is received by the algorithm, is a conditional branch or memory access. If it is a branch, the algorithm checks if the branch predicate relies on memory access to speculative explore each branch outcome and its condition. After this, the state becomes a *speculative* state, s', since it will mimic the speculation of a mispredicted branch. Then, the SpecuSym explores recursively s' and assesses the effects of the memory accesses on the cache. It then constructs the leak constrains and solves this constraint to verify if a solution exists. If the solution does not exist then SpecuSym claims there is no leakage at the event in that state. Finally, when the execution of s' finishes, we use accumulated cache data from s' to update the cache data of state $\mathtt{s}$ and continue the normal execution of $\mathtt{s}$.

SpecuSym models speculative behaviour by transferring the cache information between symbolic states and manages to create a leakage witness to determine the exposure of sensitive data from the program that was analysed. However, in order to accomplish this goal, SpecuSym establishes a threshold for the speculation window, thus, this tool can only analyse a program's speculative execution to a certain point.

In summary, SpecuSym provides a detection tool for spectre-PHT attacks by analysing the leakage model of possible mispredictions, according to Cauligi et al. [2022]. The SpecuSym uses the speculation window to limit the recursion of the algorithm as it is described previously.

2.2.5 Venkman

As described in Section 2.1, spectre attacks can poison and target different hardware and software elements. Even though this study mainly focuses on spectre-PHT attacks that target the Pattern History Table (PHT), Venkman as explained in Shen et al. [2019], tries to mitigate spectre attacks that poison entries in the Branch Target Buffer (BTB) and the Return Stack Buffer (RSB). For example, in the code below, an attacker could mistrain the BTB so that in line 1 the program could speculatively jump to line 5, bypassing the load fence placed in line 4 which protects the load at line 5 from speculative execution and hence, spectre attacks.

```
1 (*func_ptr)()
2
3 if (x < array1_size){
4     load_fence();
5     y = array2[array1[x] * 4096];
6 }
```

On that account, Venkman transforms all code running on the system so that any BTB or RSB entry does not bypass fence or other instructions inserted by the compiler to protect load instructions from spectre attacks. To accomplish this, Venkman transforms the program so that instructions are grouped into bundles and ensures that branches can only target the first instruction in the bundle. By doing this, Venkman guarantees that as long as load instructions and the instructions that protect them are within the same bundle, attackers cannot execute a load without first executing the protecting statement. To sum up, attackers can only insert the initial address of a bundle into the BTB and RSB.

The Venkman tool supports two types of code: binary code and LLVM code. The binary code is used to verify if the native code has been transformed as Venkman requires because it comes with Typed Assemble Language annotations which can help the verifier to efficiently prove that the native code conforms to Venkman's requirements. The LLVM code represents programs in a virtual instruction set that makes program analysis and transformation efficient and accurate by organising a program as a set of instructions.

Venkman starts by generating the native code from an LLVM executable for the potential victim and for an unprotected program. The code for the potential victim is passed through a set of transformations that add instructions to mitigate spectre attacks, like fence or SFI instructions. This code and the unprotected program are then transformed so that all valid targets of a control-flow transfer have an identical alignment in the virtual address space. It also transforms all branches to ensure that all entries added to the BTB

and the RSB are properly aligned. By doing this, it groups the program's instructions into bundles. This combination between alignment and protective statements ensures that instructions that need protection against spectre attacks are in the same bundle as instructions providing the protection and, therefore, mistraining the BTB and RSB cannot cause execution to bypass the instructions proving protection. Finally, the binary verifier checks whether the BTB and RSB have been applied correctly before allowing the code to execute.

In summary, Venkman provides a mitigation tool for spectre-BTB, spectre-PHT, and spectre-RSB attacks. According to Cauligi et al. [2022], Venkman combines a structure compilation of a program to not allow the training of BTB and RSB to bypass the instructions proving protection.

2.2.6 Blade

The study Vassena et al. [2020], introduces a new approach to automatically and efficiently eliminate speculative leaks from constant-time cryptography code written in C or in WebAssembly. It suffices to cut the data flow from expressions that could speculatively introduce secrets (sources) to those that leak them through the cache (sinks). To accomplish this, it was created an automatic push button tool, Blade, which eliminates potential speculative leaks using three main contributions.

First, it is necessary to formalise the semantics that translates high-level commands to low-level machine instructions enabling source-level reasoning about the absence of speculative-based execution leaks. For this was created an abstract primitive called `protect` that stops speculation for a given variable. For example, $x :=$ `protect(e)` ensures that the value of e is assigned to x only after e has been assigned its stable, non-speculative value. With this new primitive, Blade, manages to stop speculation along a particular data path without eliminating all speculation, like what happens with the introduction of fences that could incur a high-performance cost.

Furthermore, was created a static type system that types each expression as either transient (expressions that may contain speculative secrets) or stable (expressions that do not contain speculative secrets). The system prohibits speculative leaks by requiring that all sink expressions be stable. A sink expression, as mentioned above, is an instruction that reads data from a variable, thus, loads the data to the cache that can be leaked secret data through side-channels attacks.

Finally, was built an algorithm that finds potential speculative leaks and automatically synthesises a minimal number of protecting statements to ensure that the program is speculatively constant-time. To this end, was created a def-use graph that captures the data flow between program expressions. If the path from transient sources to stable sinks is found in the graph it indicates a potential speculative leak in

the program.

Blade addresses spectre vulnerabilities by incorporating a minimal number of protecting statements in the program's assembly code. The protecting statements could be fence or Speculative Load Hardening (SLH) instructions. The lfence instruction is a speculation barrier that stops speculation over this instruction, that is statements after the fence will not be executed until all statements up to the fence are executed. SLH instructions stall speculative load instructions in condition blocks by inserting artificial data-dependencies between loaded addresses and the value of the condition, which ensures that the load is not executed before the branch condition is resolved, according to Vassena et al. [2020]. By efficiently applying these protecting statements to the assembly code, it allows for SLH instructions to be applied in a more selective manner by only applying them to individual load instructions whose result flows to an instruction that might leak and prevents the usage of fences from being more restrictive than necessary which could incur in a high-performance cost.

To illustrate the Blade process let's use the example below, taken from Vassena et al. [2020], that contains a speculation vulnerability. When reading the variables x and y, an attacker could give an out-of-bounds input i_1 e i_2 that when executed speculatively could enable secret data to be stored in those two variables. This private data then flows to variable z and finally leaks through the data cache when reading $b[z]$.

```
1 x = a[i1]
2 y = a[i2]
3 z = x + y
4 w = b[z]
```

To eliminate this vulnerability and to protect this program against spectre attacks, Blade builds a def-use graph whose edges capture the data dependencies between the expressions and variables of a program. For example, the node for expression $a[i1]$ should be connected to the node for variable x indicating that expression $a[i1]$ is used to compute the variable x. To track how transient values (values that might contain secret data) propagate through the def-use graph, Blade uses a special node T which represents the source of transient values. This node T is connected with all other nodes that represent transient expressions, which for the given example are the expressions $a[i1]$ and $a[i2]$ because they depend on outside input that may be malicious. Lastly, to detect insecure uses of transient values, the graph is extended with a special node S which represents the sink of stable (non-transient) values of a program. This node S is connected with all other nodes that need to be stable expressions, for the given example the variable z. The graph in Figure 2 represents a subset of the def-use graph for the given

example. As it shows, the graph contains a path from node T to node S indicating that transient data flows through data dependencies into a stable expression and, thus, a program may be leaky.

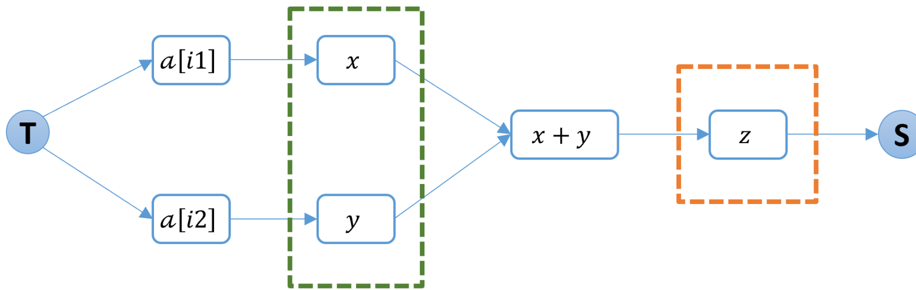


Figure 2: Subset of the def-use graph. Adapted from Vassena et al. [2020]

After creating the graph, Blade will find a cut-set (set of variables), such that removing them from the graph will eliminate all paths from node T to node S. Each cut-set defines a way to repair the program because for all variables in the set we will add the protect primitive, mitigating the spectre vulnerabilities and, hence, protecting the program against spectre attacks. For the given example, there are two sets of variables that stops the flow of transient data from T to S, as shown in Figure 2 in orange and green. However, Blade mitigates spectre attacks by minimising the usage of protect primitives, therefore, for the example shown in the figure the minimal usage of the protect primitive comes from removing the variable z from the graph. In the end, to protect the given example with minimal usage of protecting primitives, the Blade algorithm adds one protect to variable z as shown in the following code.

```
1 x = a[i1]
2 y = a[i2]
3 z = protect(x + y)
4 w = b[z]
```

Next, after patching the program, Blade uses the type system to test whether the program is secure, that is if it satisfies a semantic security condition. This semantic condition ensures that transient expressions are not used in positions that may leak data and are not written to stable variables. If a program does not guarantee these conditions then it is rejected. For this evaluation, the type system generates a set of constraints under an initial environment, then constructs the def-use graph from the constraint and finds a proper cut-set as mentioned above. Finally, computes the final environment which types the variables in the cut-set as stable, therefore, confirming that the patch program is secure.

In summary, according to Cauligi et al. [2022], Blade is a medium-level approach to mitigate spectre-PHT by using an automated data flow analysis. The tool first identifies all sources and sinks. Then, it finds

the cut points using the Max-flow/Min-Cut algorithm and either insert fences at the cut points or applies SLH to all of the loads which feed the cut point in the graph. Therefore, it reasons about speculation fences and models out-of-order execution when creating the def-use graph and when defining the minimal cut-set to protect the program.

2.3 The Jasmin type system

The last section described six different tools to mitigate and/or detect spectre attacks and vulnerabilities by using different levels of semantics. While Pitchfork uses low-level semantics, others like Kaibyo use high-level semantics to detect spectre vulnerabilities. On one hand, tools such as oo7, SpecuSym, and Venkman are not associated with formal semantics, while, Blade uses medium-level semantics to mitigate spectre attacks. Similar to Blade, Jasmin also uses medium-level semantics to detect spectre-PHT and spectre-STL vulnerabilities, but instead of being an automated tool to mitigate these vulnerabilities, Jasmin is a framework for developing high-speed and high-assurance cryptography code, giving the programmer the power to write cryptography functions and algorithms that are protected against spectre attacks.

This section describes the Jasmin framework and its properties, details some of Jasmin's specifications, and specifies the type system implemented in the Jasmin framework which mitigates spectre attacks in code developed in Jasmin.

2.3.1 Jasmin

As stated by Almeida et al. [2017], cryptography software is pervasive in software systems since it is often their most critical part, forming the backbone of their security mechanisms. For this reason, cryptography implementations must satisfy multiple properties. Some of those properties are efficiency by implying minimal overhead for software performance, protection against side-channels attacks which ensures that a program does not leak sensitive data through side-channels (also known as constant-time security), and functional correctness which are mathematical specifications that cryptography components should follow allowing to semantically detect bugs and, consequently, preventing security breaches. To implement cryptography code that satisfies these properties, Almeida et al. [2017] proposes the Jasmin framework which allows programmers to develop high-speed, high-assurance, and high-security cryptography software.

To fulfil the properties stated above, the Jasmin framework needs to be implemented as an assembly-level language in order to establish a minimal overhead performance and needs to implement high-level

abstractions that allow the framework to ensure side-channel security and functional correctness. These requirements are achieved by using the Easycrypt machine-assisted verification tool to build proof of functional correctness and side-channel security.

Therefore, to accomplish these assembly-level and high-level conditions, the Jasmin programming language is designed to support high-level features that are easy to verify and secure and assembly-level instructions enabling programmers to precisely anticipate and shape the generated assembly code in order to achieve optimal efficiency.

Language Specification

The two main properties that Jasmin aims to achieve are predictability by providing the highest levels of control to programmers over the generated code and verifiability by including several features that allow a streamlined formal verification. Therefore, the Jasmin language was created to provide a one-to-one mapping between Jasmin high-level instructions and assembly-level instructions, thus, building a uniform syntax that unifies machine instructions and high-level structures and collections to enhance predictability and verifiability.

The following sections will discuss some of Jasmin's language features. This is important because the next chapters will showcase code snippets of the Jasmin implementation for the Blake2 library, therefore, providing an overview of this language will make it easier to understand upcoming chapters.

Variables To declare variables in Jasmin, we start by specifying the storage class of the variable, followed by the corresponding type and variable name, as shown in the figure below.

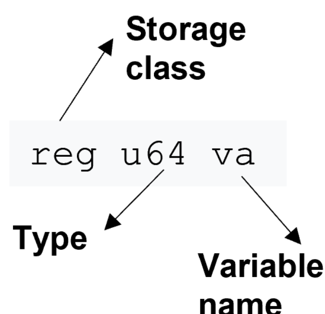


Figure 3: Jasmin declaration of variables

According to Oliveira [2022], there are four storage classes in Jasmin: **stack**, which allows variables to be allocated in the program's stack frame enabling the compiler to control its address, **reg**, that allocates

variables to a register chosen by the compiler granting responsibility to the programmer to make sure there are enough registers to allocate variables, **inline**, which initialises a variable with a statically know value, and **global**, which is similar to inline but the value will be placed in the `.data` section of the assembly file.

There are three categories of basic types in Jasmin. The **word** category can be used with all classes previously mentioned and hold the types `u8,u16,u32,u64,u128` and `u256`, allowing to declare the number of bits that the variable holds. For example, in Figure 3 the variable va will hold 64 bits of data. The **boolean** category can be used only with the class reg and provides a mechanism for handling arithmetic flags and the category **integer** specifies an unbounded integer that should be statically known and can only be used with the class inline.

The Jasmin language also allows programmers to declare arrays, supporting the declaration of reg and stack arrays, for all word types. When accessing reg arrays the index should be statically known, while accessing stack arrays the index can also be a run-time value. For instance, to declare an array in Jasmin we can use the statement `stack u64[25]` as it declares a stack array with 25 elements with 64 bits of data each.

Finally, Jasmin allows the programmer to declare a register pointer as `reg u64 aux`, which corresponds to a register variable `aux` that points to a memory region with `u64` corresponding to the number of bits that are necessary to hold a pointer in 64-bit code, thus, all external pointers must be declared as `u64`. Since pointers point to a memory region, there is an array-like notation to access information from the pointer. By default, the accesses are made in 64-bit words, hence, each value is 8 bytes in size. The Table 1 shows some examples to access information from a pointer in Jasmin.

Jasmin	Description
$[ptr]$	Accesses the first $u64$ value pointed by ptr.
$[ptr + 8]$	Accesses the second $u64$ value pointed by ptr.
$(\mathrm{u8})[ptr + 1]$	Accesses the second byte pointed by ptr.
$(\mathrm{u256})[ptr]$	Accesses the first $u256$ value that is pointed by ptr.

Table 1: Access data from a pointer in Jasmin

Operators and Instructions As stated before, each instruction is usually mapped into one assembly instruction and, thus, each operator (`+, -, *, =` and /) must be declared with a compatible storage class and type for the corresponding assembly instruction. For instance, if an assembly instruction requires

that each variable is in the register, then the Jasmin operator needs to receive the corresponding variables declared using the reg storage class. It is the responsibility of the programmer to be aware of such requirements and restrictions when writing Jasmin code.

For example, for the operator = the programmer is allowed to perform a copy from register to register, stack to register, and vice-versa, but it is not allowed to copy a value from a stack variable to another stack variable. To perform this copy, the programmer needs to first copy the stack variable to a register variable and then copy this register variable to the stack variable, hence performing two assembly instructions and two Jasmin instructions. Table 2 shows several arithmetic operators as implemented in Jasmin and its corresponding assembly instruction for the types `u8,u16,u32,u64`.

Jasmin	Assembly	Notes
$a+=1$	inc a	Addition by 1 is compiled into inc.
$a+=b$	add b, a	Addition.
cf, $a+=b$	add b, a	Same as previous, but carry flag can be used.
cf, $a+=b$+cf	adc b, a	Addition with carry.
$a=-a$	neg a	Two's complement negation.
$a-=b$	sub b, a	Subtraction.
cf, $a-=b$	sub b, a	Same as previous, but carry flag can be used.
cf, $a-=b$−cf	sbb b, a	Subtraction with borrow.
$a=b+c+d$	lea d(b, c), a	Addition using lea. Displacement d can be 0.
$a=b*s+c+d;$	lea d(b,c,s), a	Same as previous. Scale factor can be 2, 4, or 8.
$a=b*s+d;$	lea d(, b, s), a	Same as previous. But c is omitted.
$h,a=a*b$	mul b	Unsigned multiply. a allocated in rax. h in rdx.
$a*=b$	imul b, a	Signed multiply.
$a=b*i$	imul i, b, a	Signed multiply. i is an immediate value.
$a=a/b$	div b	Unsigned division. a in rax. rdx is 0.

Table 2: Arithmetic operators overview. Taken from Oliveira [2022]

Following Oliveira [2022], Jasmin also allows the programmer to perform operations in the context of bitwise and shift instructions, as shown in the Table 3 for the types `u8,u16,u32,u64`.

Finally, there are several assembly instructions, such as BSWAP or ROL, which do not correspond to any Jasmin operators previously mentioned. Therefore, Jasmin enables programmers to perform a direct call to these instructions by declaring these instructions with the prefix # and suffixed with the size, indicat-

Jasmin	Assembly	Notes
a$^{\wedge} =$ b	xor b, a	Logical exclusive OR.
a $\|= b$	or b, a	Logical inclusive OR.
a & $=$ b	and b, a	Logical AND.
a $=$!a	not a	One's complement negation.
a $=$!b&c	andn c, b, a	Logical AND NOT.
a $<<= i$	shl i, a	Shift logical left.
a $>>= i$	shr i, a	Shift logical right.
a $>> s = i$	sar i, a	Shift arithmetic right.

Table 3: Bitwise and shift operators overview. Taken from Oliveira [2022]

ing the types of inputs. For example, the instruction BSWAP can be declared as `b = #BSWAP_64(b)`, meaning that the instruction will reverse the byte order of the register 64-bit variable `b`. Figure 4 shows the Jasmin instructions that are accepted for the types `u8,u16,u32,u64`.

ADC, ADCX, ADD, ADOX, AND, ANDN, BSWAP, BT, CMOVcc, CMP, CQO, DEC, DIV, IDIV, IMUL, IMULr, IMULri, INC, LEA, MOV, MOVD, MOVSX, MOVZX, MUL, MULX, NEG, NOT, OR, RCL, RCR, ROL, ROR, SAL, SAR, SBB, SETcc, SHL, SHLD, SHR, SHRD, SUB, TEST, XOR

Figure 4: Jasmin Instructions. Taken from Oliveira [2022]

Control-Flow As referred in this section, Jasmin achieves predictability and verifiability, by using the following high-level and control-flow structures: `if`,`for` and `while`. These structures use conditions to determine whether a certain branch should be executed or not. There are two types of conditions in Jasmin: the run-time conditions that depend on run-time values which can be determined during the execution of a program, like `a < 4`, and statically known conditions that do not depend on run-time values. For run-time conditions, they can be grouped into boolean conditions which use arithmetic flags (`reg bool` variables), and word conditions that require two operands of the same type with one being a run-time value and the other being a statically known value.

The `if` statement can be used with or without the `else` clause and is written in Jasmin as

```
if(condition) {then_block} else {else_block}
```

. This statement can use with both run-time and statically known conditions.

The `for` statement allows to specify loops that are unrolled during compilation for better performance, thus, its conditions can only depend on statically known values. In Jasmin, the `for` statement can be described as

```
for i = initial_value to end_value {loop_block}
```

when the structure indicates that the iterator `i` is incremented by 1, and as `for i = initial_value downto end_value {loop_block}` if the iterator is decremented by 1.

At last, the statement `while` can be specified as

```
while (condition) {loop_block}
```

when we want to first verify the condition and then, if true, execute the `loop_block` or as `while {loop_block} (condition) {it_block}` if we want to validate the condition at the end of each `loop_block` and, if the condition is true, execute the `it_block`. For both definitions, the `condition` can depend on run-time values, however, conditions that are statically known are not useful in this context considering that Jasmin does not implement break statements the `while` loop would continuously be executed. In comparison to the `for` statement, the `while` loop is preserved during compilation and is never unrolled.

Functions In Jasmin, it is supported three types of functions: **inline**, which are inlined at the caller and can be seen as an extended macro mechanism, **local**, which allows a programmer to write functions whose code is shared, and **export**, that can be called from an external code and implement the System V calling convention.

The inline functions can be declared as:

```
inline fn function_name (arguments) -> return_types { function_body }
```

These functions can receive `reg`, `stack` and `inline` variables or arrays and then return the same storage classes. This type of function is mainly used to execute portion of the computation done by the main program or other inline function and can include control-flow structures and function calls to other inline and local functions.

Local functions are declared as:

```
fn function_name (arguments) -> return_types {function_body}
```

They receive as input only register variables with any word type and as output any type that is allowed in the input list. This type is used to share code between other functions and it is allowed to include function calls to other inline or local functions.

Lastly, the export function is declared as:

```
export fn function_name (arguments) -> return_types {function_body}
```

They can receive up to 6 registers as arguments with types `u8,u16,u32,u64` and 8 registers with types `u128,u256` and can only return one register variable or not return anything. This type can be used in external programs, but can not call other export functions or define functions within the export function. This type does not receive or return register arrays, but it can receive a register pointer to a memory address.

Semantics

The behaviour of Jasmin programs can be described as a relation between initial and final states. This semantics defines a partial function, that is, for every initial state there is at most one final state which reflects that Jasmin programs have a deterministic behaviour making them predictable. A detailed account of the semantic rules of the Jasmin language can be found in Almeida et al. [2017] (figure 4).

2.3.2 Type System

cryptography algorithms and libraries are a common and important part of current applications. As stated in the previous section, cryptography code needs to be implemented by using a high-speed and high-assurance framework, like Jasmin. However, it is also important that these cryptography algorithms be protected against security breaches since they deal with private and confidential data like secret keys and certificates that should not be leaked to outside entities.

For this reason, it was implemented, under the Jasmin framework, a type system Shivakumar et al. [2022] that protects cryptography implementations against spectre-PHT attacks by ensuring speculative constant-time. The speculative constant-time policy makes sure that control-flow and memory accesses be independent of secret data during speculative execution, that is, a program is speculative constant time if its leakage does not depend on secrets, for any execution of a program even during speculative execution.

Jasmin's type system is comprised of three primitives that must be added to Jasmin code by the programmer. When these primitives are added correctly and efficiently, they work together to protect any Jasmin code that may be vulnerable to spectre attacks.

At the assembly level, these primitives verify whether the CPU is correctly executing a branch instruction speculatively by evaluating if the branch condition was wrongly evaluated as true. If it was, the primitives discard the outcome of the branch instruction to prevent it from being stored in the cache and potentially leaked.

Language Implementation

Since the type system protects cryptography code from spectre attacks, it needs to satisfy the same properties as cryptography software, so it does not hinder the performance, execution, and security of these programs. Therefore, the type system is composed of three different primitives, as presented by Shivakumar et al. [2022], that guarantee speculative constant-time, minimal overhead, maintain normal execution of a program, and protect it against spectre attacks[1].

The following example given by Shivakumar et al. [2022] leaks secret data speculatively via a branch prediction, similar to the example shown in Section 2.1.1. If the program receives as input an out-of-bounds value of `i` and if the branch predictor incorrectly guesses the condition `i < 10` as true it is possible, under speculative execution, that a secret value at position `p[i]` is loaded into `x`, making `x` a transient variable. A variable is transient when its value speculative depends on secrets and therefore, can be easily leaked as it is shown in line 4.

```
1 if (i < 10) {
2     x = p[i];
3 }
4 w[x] = 0;
```

To protect this example, the programmer uses the primitive, `init_msf()`, to set the misspeculation flag, `ms`, to 0, ensuring that the code is entered in normal execution mode. After a branch instruction conditioned on `b`, the programmer needs to use the primitive `set_msf(b, ms)` which updates the misspeculation flag `ms`. In this primitive, if the branch is executed in normal mode then `ms` is equal to 0, but if the branch is being speculatively executed then `ms` is set to 1. Finally, the programmer needs to use the primitive `protect(x,ms)` to protect the transient variable `x` from being leaked during speculative execution. This primitive masks the variable `x` according to the flag `ms`, i.e., the value of `x` remains unchanged in a normal execution mode or it is set to -1 if the `ms` flag is 1. Thereby, whenever the value of `x` is -1, the variable is discarded, thus, the speculative value of `x` is never leaked. These primitives assure that the result of the load instruction at line 2 is only registered in a normal execution mode maintaining

[1] Throughout the rest of this section the spectre-PHT attacks are going to be referred to as spectre attacks

a minimal overhead, the correct output of the program, and protecting the code against spectre attacks. The ensuing example represents the protected version of the example given above.

```
1 ms = init_msf();
2 b = i < 10;
3 if (b) {
4     ms = set_msf(b,ms);
5     x = p[i];
6     x = protect(x,ms);
7 }
8 w[x] = 0;
```

Since the type system uses primitives that need to be manually implemented in the program, it is the responsibility of the programmer to first determine the transient variables that need to be protected by using the `protect` primitive and then decide where to implement the `set_msf` primitive. By manually implementing these primitives into the cryptography code, the programmer manages to protect it against spectre attacks by enforcing the speculative constant time policy before compiling the programs.

In order to integrate the type system into the Jasmin framework, the Jasmin language was extended with the three primitives and security annotations to allow the framework compiler to produce valid assembly instructions. The assembly code below shows the compiled assembly version of the protected example given above.

```
1  example:
2    lfence
3    movq   $0, %rcx
4    cmpq   $10, %rdi
5    jnb    Lexample$1
6    movq   $-1, %rax
7    cmovnb   %rax, %rcx
8    movq   (%rsi,%rdi), %rax
9    orq    %rcx, %rax
10 Lexample$1:
11   ret
```

The assembly code begins by compiling the primitive `init_msf()` into lines 2 and 3. It starts by assigning the register `%rcx`, which corresponds to `msf`, to value 0 as shown in line 3. The Jasmin framework then compiles the branch condition `b` and the primitive `set_msf(c,msf)` into lines 6 and

7. If the program speculatively predicts the execution of line 6 while line 4 is being executed, the register %rax is assigned to value -1. In the meantime, if the result of the comparison at line 4 determines a misprediction, the result will carry the flag CF equal to 0. At line 7, the register %rcx is assigned to -1, only if the CF flag is equal to 0, that is, the variable msf is set to -1 if the condition b is false, meaning that the program is running at speculative mode. Finally, the compiler translates the instruction x = p[i] into its assembly code at line 8 and protects the register %rax, which corresponds to variable x, by masking its value with the value of %rcx at line 9. The masking is obtained by using the instruction orq which maintains the value of %rax only if %rcx is 0, i.e. when the program is running in normal mode.

Semantics

Sequential semantics of the Jasmin framework reasons about non-speculative execution of programs. It is, therefore, suitable to reason about functional correctness, but not about leakage under speculative execution. Hence, to model the semantics of program behaviours within the type system, the type system defines speculative constant-time as equality of observations, which represents the knowledge gained by the attacker after an instruction is executed.

A program is speculative constant-time iff, for every directive that details the attack method to exploit speculative execution, a program is executed at two initial states under each directive and produces two observations that are equal to each other. Thus, an attacker does not gain any different information no matter the initial state of the program and the directive in which the program is executed.

Chapter 3

Use case

This chapter aims to give a detailed overview on the use case used for this study. It starts by giving a comprehensive description on the several phases of this research and explains the algorithm behind the Blake2b library to be used in the study.

3.1 Research Overview

The goal for this study is to verify if the cryptography library Blake2b can resist spectre attacks using the type system from the Jasmin framework and the Blade tool as mitigation techniques and in the end do a comparative analysis between the type system and the Blade tool.

Since the study aims to compare other mitigation techniques to the type system it is important to choose a tool that mitigates code written in a different language than Jasmin, has a different algorithm than the type system and that could easily be compared against the type system by having similar features. Therefore, the Blade tool was chosen, considering that is still is being maintain by the developers, has a different mitigating algorithm than the type system, and mitigates WebAssembly and C code. On the other hand, similar to Blade, Jasmin and the type system also uses medium-level semantics to detect spectre-PHT and spectre-STL vulnerabilities. In addition, it is a quite known and established mitigating tool for spectre vulnerabilities and could be a great way to compare the benefits and disadvantage of the Jasmin's type system.

As detailed in section 2.2.6, the Blade tool effectively addresses spectre vulnerabilities in cryptography code written C and WebAssembly. Given the distinct nature of these two programming languages in terms of variable allocation and memory access, the Blade tool mitigates these vulnerabilities differently, even when the two codes perform identical functions. However, this study will focus its attention in mitigating spectre vulnerabilities in the Blake2b library written in C. This emphasis stems from the fact that the entirety of this study has been directed towards mitigating spectre vulnerabilities identified in both the

Jasmin code and the C implementation of the Blake2b library.

In order to develop a more complete research, this study also focus on detecting spectre vulnerabilities in the Blake2b library. Since the type system mitigates cryptography code written in Jasmin and the Blade tool patches code written in C, this study also aims to detect spectre vulnerabilities in the reference code of the Blake2b library written in C and in the Jasmin version of this library. For this, the study will use the detection module of the oo7 tool to detect spectre vulnerabilities in the reference code and Jasmin features to detect these vulnerabilities in the Jasmin code of the Blake2b library. The oo7 was chosen to detect spectre vulnerabilities in the reference code, as it is a tool that is still being maintained by the developers, and it is easy to install, use and understand its algorithm. This tool could also be a good option to compare its mitigation strategy against the type system and the Blade tool by using its detection module, however this module is currently unavailable to the public.

To develop a concise and complete research, this study will start by detecting the spectre vulnerabilities in the reference and Jasmin code of the Blake2b library to demonstrate how the library is vulnerable to spectre attacks. It then, uses the Blade and the type system to mitigate the spectre vulnerabilities in the C and Jasmin code, respectively. Finally, the study will evaluate and compare the outcomes of mitigating the spectre vulnerabilities achieved through the usage of the Blade tool and the Jasmin framework. To carry out this assessment and comparison, it is essential to define criteria for comparison and methods for analysing these criteria. Using this criteria, the research will determine which tool is ideal for which purpose.

To note that the code used for the reference and the Jasmin code of the Blake2b library can be found in A.1 and in A.3, respectively.

3.2 Blake2 Library

As stated in the previous section, the Blake2b library will be used to be mitigated against spectre attacks by the Blade tool and the type system from Jasmin. This section aims to give a complete overview of the Blake2b library and its implementation of the Blake hash function, so that the study presented in the next chapters becomes easier to understand.

3.2.1 Blake

Hash functions are mathematical algorithms that convert input data of arbitrary size into fixed-size output values, called hash values or message digests. Hash functions are widely used in computer science

and information security to provide a fast and efficient way to verify data integrity and detect any changes or tampering in a message or file. These functions are also essential for secure password storage and user authentication, as they allow passwords to be stored in a hashed form, rather than in plain text, which can protect against unauthorised access or data breaches.

Therefore, it is essential for hash functions to maintain several properties to ensure the security and reliability of digital data and communications. When hash functions fail to maintain these properties, they become vulnerable to attacks and, hence unreliable to use. This was what happened to three of the most popular and widely used hash algorithms, MDS, and SHA-1, as it was discovered that they were vulnerable to attacks due to the absence of the collision resistance property. This was the catalyst for the SHA-3 competition, which was a global contest organised by NIST to choose a new cryptography hash algorithm that would become the standard for federal government applications.

The BLAKE hash function was designed as a candidate for the SHA-3 competition and was created in 2012. BLAKE is based on a new design strategy that combines the best features of previous hash functions, such as SHA-2, while also improving upon them in terms of speed, security, and flexibility. BLAKE supports hash output sizes of 224, 256, 384, and 512 bits, and it is known for its high level of resistance against various types of attacks, including collision attacks, preimage attack, and differential attacks. Overall, BLAKE is a well-regarded and widely-used hash function that has proven to be both efficient and secure.

According to Aumasson et al. [2013], after the SHA-3 competition and after an extensive cryptanalysis of BLAKE's security and efficiency properties, the authors of BLAKE introduce BLAKE2, an improved version of the BLAKE hash function. It was designed to be fast, secure, and highly flexible and it has quickly gained popularity among developers and security experts due to its performance and versatility.

3.2.2 Blake2 Algorithm

The Blake2 hash function comes in two different versions. The Blake2b version is optimised for 64-bit platforms and can produce a digest of any size between 1 and 64 bytes. On the other hand, the Blake2s version is optimised for 8 to 32-bit platforms and can produce a digest between 1 and 32 bytes. Starting from this section, the Blake2b version will be referred to as simply Blake2, as this study focuses on this version.

The Blake2 algorithm that is going to be used, receives as input three general parameters as shown in Table 4. With these parameters, the algorithm will perform three functions, `init`, `update`, `final`.

The purpose of the `init` function is to ensure that the hash function state is correctly initialised and

Parameter	Variable Type	Optional	Description
Message	Bytes	No	Message to be converted to an hash value
Key	Bytes	Yes	A random string in bytes that can be used to modify the hash function's behaviour
Digest length	Integer	No	Size of the hash value. In this case, the maximum value is 64.

Table 4: Parameters for the Blake2 algorithm

ready to compute the hash of the input data. It achieves this by initialising every structure and variable and applying the key parameter, if one is provided, to the hash. The `update` function is responsible for calling the compression function, which is the core of the algorithm and it creates the hash from the input data by using the initialisation, round function, and finalisation stage, following Rajaram and Mathi [2012]. Finally, the `final` function applies the compression function to remaining data that has not been processed, updates the length of the data, and applies any finalisation flags that have been set. These steps ensure that the hash computation is correctly finalised, which is critical for maintaining the security and integrity of the hash computation.

The Blake2 algorithm begins by padding the message to form a sequence of N 16-word blocks. For each block message, the algorithm will perform the compress function to create the hash value of that block. In the end, the algorithm will return the sum of all hash values.

Chapter 4

Spectre Analysis

This chapter provides an analysis of the spectre vulnerabilities discovered in the reference code of the Blake2b library using the oo7 vulnerability detection tool. It also investigates the spectre vulnerabilities detected in the Jasmin code of the same library and demonstrates the application of mitigation techniques such as the type system and Blade tool to address the vulnerabilities identified in both the reference and Jasmin code for the Blake2b library.

4.1 Detecting Vulnerabilities

The following section outlines the process of detecting spectre vulnerabilities in the Blake2b library. The oo7 tool, as explained in section 3.1, will be used to identify spectre vulnerabilities in the library's reference code written in C. In addition, the Jasmin framework will be employed to detect violations of the speculative constant-time policy in the Jasmin code, which ensures that the code is susceptible of spectre attacks, as described in section 2.3.2.

4.1.1 Reference Code

The oo7 tool works by finding potentially vulnerable patterns in the binary code of a program. Despite the explanation provided in section 2.2.3 regarding the two primary modules within the oo7 framework, namely one for detecting spectre vulnerabilities and another for mitigating these vulnerabilities, it is important to note that the mitigation module is currently unavailable to the public as stated in section 3.1. For that reason, the oo7 tool will be used to detect spectre vulnerabilities in the reference code from the Blake2b library shown in A.1.

The oo7 tool is publicly available on their Github page[1] for anyone to install and use. In order to streamline the installation process, a Dockerfile was created (as shown in B.1) that includes all the nec-

[1] https://github.com/winter2020/oo7

essary commands to fully install the tool within a docker container. To access greater processing power and memory than a typical computer, the tool was installed on a remote machine named `glorfindel`, which was made available by professors at Ghent University. This machine is composed of 8 processors from Intel (Intel(R) Core(TM) i7-10700 CPU) with each having 8 CPU cores and the total amount of system memory is 125GB.

Output Analysis

To test the tool and to better understand how this tool works before running the reference code of Blake2b, it was used the following *test* code:

```
1  /* toy example from Spectre paper */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <stdint.h>
5  #include <time.h>
6
7  unsigned int array1_size = 16;
8  uint8_t array1[16];
9  uint8_t array2[256 * 512];
10 uint8_t temp = 0;
11
12 struct timespec time_start, time_end;
13 size_t time_diff;
14
15 void victim_fun(int idx) {
16     if (idx < array1_size) {
17         temp &= array2[array1[idx] * 512];
18     }
19 }
20
21 int main(int argn, char* args[]) {
22     int source;
23
24     FILE *file = fopen("temp.txt", "r");
25
```

```
26        if (file == NULL) {
27            printf("No file!");
28            return 0;
29        }
30        source = fgetc(file);              //taint source
31        victim_fun(source);
32        return 0;
33    }
```

To detect spectre vulnerabilities in the provided C code using the oo7 tool, it is necessary to generate its binary code using the "objdump" command. This binary code can then be used along with the recipe "spectre" that has all the rules written in Primus Lips to detect spectre vulnerable patterns in the `bap test --recipe=spectre` command to execute the oo7 tool.

The oo7 framework generates two distinct outputs. One output is generated by BAP through the "grep spectre-path incidents" command. Since this is an independent tool that was added to the oo7 framework to detect vulnerable patterns, it provides its own report aside from the oo7 output. The other output presents the report of the oo7 tool, which can be viewed by executing the command `/incidents_profile.py incidents test.asm`. Here, "test.asm" refers to the assembly code corresponding to *test* C code . This final output represents the comprehensive report generated by the oo7 tool.

As mentioned in 2.2.3, a program is susceptible to spectre attacks if a pattern like <TB, RS, LS> exists within the code. The TB denotes a tainted branch, that is a branch that is dependent on a variable from un-trusted source, then the RS means that the program reads a variable using the un-trusted input, and finally the LS indicates there is a memory access that is dependent on a secret which leads to a leakage. The output obtained from BAP for the provided code is `(spectre-path (1:63u#2761 (9 (S3 (cond 6ec) (load 6fb) (last 70f)))))`, which corresponds to the <TB, RS, LS> pattern, as depicted in Figure 5.

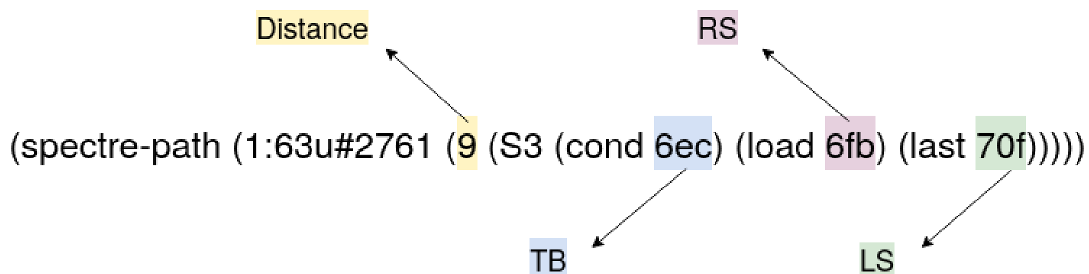


Figure 5: BAP output for test program

The figure shows the vulnerable pattern in the assembly code along with the distance between the TB and the LS instruction. In this case, the instruction 6ec in the assembly code represents the tainted branch TB, the 6fb represents the RS instruction and the LS is display in the instruction 70f. These three instructions together form the pattern `<TB, RS, LS>` which means that the program is vulnerable to spectre attacks. The Figure 6 highlights part of the assembly code that is spectre vulnerable, as stated by the output of BAP.

```
void victim_fun(int idx) {
 6da:     55                       push   %rbp
 6db:     48 89 e5                 mov    %rsp,%rbp
 6de:     89 7d fc                 mov    %edi,-0x4(%rbp)
    if (idx < array1_size) {
 6e1:     8b 55 fc                 mov    -0x4(%rbp),%edx
 6e4:     8b 05 26 09 20 00        mov    0x200926(%rip),%eax        # 201010 <array1_size>
 6ea:     39 c2                    cmp    %eax,%edx
 6ec:     73 34                    jae    722 <victim_fun+0x48>
        temp &= array2[array1[idx] * 512];
 6ee:     8b 45 fc                 mov    -0x4(%rbp),%eax
 6f1:     48 63 d0                 movslq %eax,%rdx
 6f4:     48 8d 05 45 09 20 00     lea    0x200945(%rip),%rax        # 201040 <array1>
 6fb:     0f b6 04 02              movzbl (%rdx,%rax,1),%eax
 6ff:     0f b6 c0                 movzbl %al,%eax
 702:     c1 e0 09                 shl    $0x9,%eax
 705:     48 63 d0                 movslq %eax,%rdx
 708:     48 8d 05 71 09 20 00     lea    0x200971(%rip),%rax        # 201080 <array2>
 70f:     0f b6 14 02              movzbl (%rdx,%rax,1),%edx
 713:     0f b6 05 07 09 20 00     movzbl 0x200907(%rip),%eax        # 201021 <temp>
 71a:     21 d0                    and    %edx,%eax
 71c:     88 05 ff 08 20 00        mov    %al,0x2008ff(%rip)        # 201021 <temp>
    }
}
 722:     90                       nop
 723:     5d                       pop    %rbp
 724:     c3                       retq
```

Figure 6: Part of the assembly for the test program

From the figure it can be seen that the instruction 6ec corresponds to the "if" statement in the function `victim_fun`. This "if" is dependent on a un-trusted variable since the `idx` is loaded from un-trusted source. The instructions 6fb corresponds to the load `array1[idx]`, where the program reads a variable that is dependent on `idx`. Finally, the instruction 70f represents a memory access that leads to the store of `array1[idx]` in the cache. This value could then be leaked by using a microarchitectural side-channel attack.

Similar to the BAP output, the oo7 tool also identifies the instructions that form a vulnerable pattern. The oo7 generates a text file that provides a complete overview of the analysis, including all the tainted instructions, tainted branches (TB), and vulnerable reads (RS), regardless of whether they are part of a vulnerable pattern or not. This makes the oo7 output more comprehensive and detailed compared to the BAP report.

Figure 7: oo7 Report for the test program

Figure 7 displays the complete oo7 report obtained from running the *test* code. The orange section presents the number and assembly locations of TB, which are *jump* instructions influenced by variables from un-trusted sources. However, not all of these instructions pose a threat to the program. Only the tainted branches that form a `<TB, RS, LS>` pattern are considered hazardous. The blue area represents the vulnerable reads that, in combination with the TB identified in orange, may be susceptible to spectre attacks. The first ID corresponds to the associated tainted branch, the second ID represents the RS instruction, and the final number denotes the distance between the TB and RS. The red section highlights the vulnerable pattern also identified in the BAP output. It starts by identifying the hazardous tainted branch in orange, followed by the RS instruction, and concludes with the LS instruction, thus constituting the `<TB, RS, LS>` pattern. Lastly, the green section lists various instructions that acquire information from un-trusted sources.

Blake2b Output

As mentioned earlier, the oo7 framework was employed to identify spectre vulnerabilities in the reference code (A.1) of the Blake2b library. Due to the code's complexity, the analysis with the framework requires a significant amount of time to execute. However, after several hours of running the tool on the `glorfindel` machine, the process was terminated because it was consuming an excessive amount of memory, approximately 125 GB.

To analyse the memory usage of the oo7 tool during the execution of the reference code, a graph was generated to illustrate its memory consumption over time on the `glorfindel` machine before it was terminated. Figure 8 presents the graph depicting the memory usage in gigabytes (GB) at 15-minute intervals for a duration of fourteen and a half hours.

This graph reveals a significant increase in memory consumption approximately three hours and forty-five minutes into the execution of the tool. Subsequently, from that point until the termination of the execution, memory consumption continued to increase steadily. This graph pattern could indicate a memory leak or loss, resulting in excessive memory consumption and hindering the complete execution of the spectre analysis.

Valgrind, a powerful debugging and profiling tool widely used by developers, was employed to investigate potential memory leaks and losses in the oo7 tool. The tool's capability to identify memory leaks, improper memory usage, and other programming errors in software makes it an essential resource in this context. The summary below provides an overview of the memory leak findings obtained from Valgrind after several hours of running the oo7 tool.

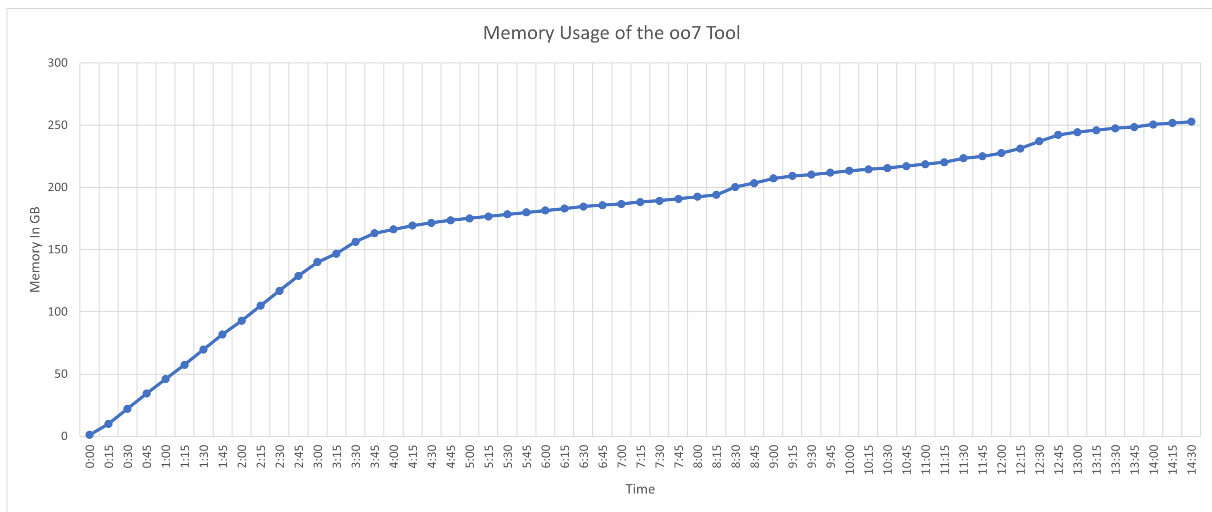


Figure 8: oo7 Memory Consumption

```
1 LEAK SUMMARY:
2 definitely lost: 37 bytes in 1 blocks
3 indirectly lost: 0 bytes in 0 blocks
4     possibly lost: 2,704,315,633 bytes in 28 blocks
5 still reachable: 47,296,423 bytes in 4,371 blocks
6     suppressed: 0 bytes in 0 blocks
7 Reachable blocks (those to which a pointer was found) are not shown.
```

As shown in the summary there are plenty of bytes that are lost, therefore it can be concluded that the oo7 tool sufferers from a memory leak that is responsible for the incorrect execution of the tool while running a more complex code like the Blake2b code.

Despite the leakage issue hindering the generation of an analysis report by the oo7 tool, it is still possible to obtain the BAP report after running the tool for a few hours. This discrepancy may be attributed to the leakage problem specifically affecting the generation of the text file required for the oo7 tool to produce its own report. However, since the vulnerability checker is executed by the BAP toolkit, it enables the generation of the BAP report ("Patter Found" in figure 1). Therefore, although the oo7 tool cannot provide a conclusive analysis, the BAP report can still be obtained, revealing all the spectre vulnerable patterns present in the code. Figure 9 shows the BAP output taken from running the reference code with the oo7 tool.

After analysing the output, it was concluded that the two highlighted patterns in pink, as shown in Figure 9, were the primary vulnerable points in the Blake2b code. The other patterns exhibited some level of correlation with these main patterns, indicating that if the two main vulnerabilities were addressed,

```
(spectre-path (0xFED:63u#66997 (23 (S3 (cond 1a09) (load 1aff) (last 1ca2)))))
(spectre-path (0x65:63u#12804 (21 (S3 (cond 1c79) (load 1aff) (last 1ca2)))))
(spectre-path (0xFF5:63u#67081 (12 (S3 (cond 1b00) (load 1b11) (last 1b15)))))
(spectre-path (0xFE5:63u#66910 (11 (S3 (cond 1b4e) (load 1b70) (last 1b8a)))))
(spectre-path (0xFE5:63u#66910 (18 (S3 (cond 1b9d) (load 1b70) (last 1b8a)))))
(spectre-path (0x12:63u#10333 (15 (S3 (cond 1af4) (load 1ac7) (last 1ae4)))))
(spectre-path (0x14:63u#10546 (22 (S3 (cond 19d6) (load 1aff) (last 19f4)))))
(spectre-path (0x35:63u#11746 (56 (S3 (cond 18ec) (load 1aff) (last 19f4)))))
(spectre-path (0x14:63u#10546 (24 (S3 (cond 19cf) (load 1aff) (last 19f4)))))
(spectre-path (0x35:63u#11746 (54 (S3 (cond 18f3) (load 1aff) (last 19f4)))))
(spectre-path (0x65:63u#12804 (48 (S3 (cond 1941) (load 1aff) (last 19f4)))))
(spectre-path (0x14:63u#10546 (52 (S3 (cond 18fa) (load 1aff) (last 19f4)))))
```

Figure 9: BAP report for the Blake2b library

the remaining ones could be easily patched. For instance, in the first pattern, the TB corresponds to the return jump in the `blake2b_init` function, the RS corresponds to the return in the `blake2b_update` function, and the LS corresponds to the call to the `blake2b_final` function. As the highlighted patterns are linked to vulnerabilities within the `blake2b_final` and `blake2b_update` functions, addressing these vulnerabilities will automatically resolve the first pattern that was identified. The remaining patterns show possible vulnerabilities within these functions, while the two main patterns specifically indicate the vulnerable sections within these functions that are susceptible to spectre attacks.

The first main vulnerability that appears in the BAP output is related to the `blake2b_final` function. The listing below shows the code snippet of the function that is vulnerable to spectre attacks. This snippet is responsible to filled with 0 the last bytes of the input buffer until the size of the buffer becomes 128.

```
1 while (ctx->c < 128)
2     ctx->b[ctx->c++] = 0;
```

According to the analysis done by BAP, as shown in Figure 10, the previous snippet shows a `<TB, RS, LS>` pattern making the program vulnerable to spectre attacks. BAP concluded that line 1 is a TB since an attacker could access the memory address of the pointer `ctx`, which is a structure that has all the information needed to produce the hash of an input message. The manipulation of this pointer could then change the `ctx->c` memory address to another that is not valid, by changing the value of `ctx->c` in the RS instruction `ctx->c++`, leading to a change of value in a memory space that is private. This value is then stored in cache since the LS instruction `ctx->b[ctx->c++]` uses the previous one to compute its values, which allows the data in `ctx->b` to be overwritten by 0 in an invalid memory address.

The last main vulnerability that appears in the BAP output is related to the `blake2b_update` function.

```
(spectre-path (0xFE5:63u#66910 (18 (S3 (cond 1b9d) (load 1b70) (last 1b8a)))))

while (ctx->c < 128)
   1b6a:    eb 22                     jmp    1b8e <blake2b_final+0x8d>
      ctx->b[ctx->c++] = 0;
   1b6c:    48 8b 45 e8               mov    -0x18(%rbp),%rax
   1b70:    48 8b 80 d0 00 00 00      mov    0xd0(%rax),%rax
   1b77:    48 8d 48 01               lea    0x1(%rax),%rcx
   1b7b:    48 8b 55 e8               mov    -0x18(%rbp),%rdx
   1b7f:    48 89 8a d0 00 00 00      mov    %rcx,0xd0(%rdx)
   1b86:    48 8b 55 e8               mov    -0x18(%rbp),%rdx
   1b8a:    c6 04 02 00               movb   $0x0,(%rdx,%rax,1)
   while (ctx->c < 128)
   1b8e:    48 8b 45 e8               mov    -0x18(%rbp),%rax
   1b92:    48 8b 80 d0 00 00 00      mov    0xd0(%rax),%rax
   1b99:    48 83 f8 7f               cmp    $0x7f,%rax
   1b9d:    76 cd                     jbe    1b6c <blake2b_final+0x6b>
```

Figure 10: Vulnerability in the `blake2b_final` function

The listing below shows the code snippet of the function that is vulnerable to spectre attacks. This snippet is responsible for calling the `blake2b_compress`, which creates the hash for each block of the input message.

```
1  for (i = 0; i < inlen; i++) {
2      if (ctx->c == 128) {
3       ctx->t[0] += ctx->c;
4       if (ctx->t[0] < ctx->c)
5        ctx->t[1]++;
6       blake2b_compress(ctx, 0);
7       ctx->c = 0;
8      }
9      ctx->b[ctx->c++] = ((const uint8_t *) in)[i];
10 }
```

According to the BAP analysis, as shown in Figure 11, the snippet shows a `<TB, RS, LS>` pattern making the program spectre vulnerable. In line 1, if the attacker could control the variable `inlen` it would make this "for" a tainted branch since the variable `inlen` comes from an un-trusted source. Similar to what happened in the `blake2b_final` function, if the attacker could control the `ctx` pointer it could then change the `ctx->c` memory address to another that is not valid, by changing the value of `ctx->c` in the RS instruction `ctx->c++`. This allows the LS instruction in line 9 to overwrite in a private address of `ctx->b[ctx->c++]` a value, that could be leaked, since the `ctx->c++` was stored in cached.

```
(spectre-path (0x12:63u#10333 (15 (S3 (cond 1af4) (load 1ac7) (last 1ae4)))))

ctx->b[ctx->c++] = ((const uint8_t *) in)[i];
   1ab7:   48 8b 55 e0              mov    -0x20(%rbp),%rdx
   1abb:   48 8b 45 f8              mov    -0x8(%rbp),%rax
   1abf:   48 8d 34 02              lea    (%rdx,%rax,1),%rsi
   1ac3:   48 8b 45 e8              mov    -0x18(%rbp),%rax
   1ac7:   48 8b 80 d0 00 00 00     mov    0xd0(%rax),%rax
   1ace:   48 8d 48 01              lea    0x1(%rax),%rcx
   1ad2:   48 8b 55 e8              mov    -0x18(%rbp),%rdx
   1ad6:   48 89 8a d0 00 00 00     mov    %rcx,0xd0(%rdx)
   1add:   0f b6 0e                 movzbl (%rsi),%ecx
   1ae0:   48 8b 55 e8              mov    -0x18(%rbp),%rdx
   1ae4:   88 0c 02                 mov    %cl,(%rdx,%rax,1)
   for (i = 0; i < inlen; i++) {
   1ae7:   48 83 45 f8 01           addq   $0x1,-0x8(%rbp)
   1aec:   48 8b 45 f8              mov    -0x8(%rbp),%rax
   1af0:   48 3b 45 d8              cmp    -0x28(%rbp),%rax
   1af4:   0f 82 31 ff ff ff        jb     1a2b <blake2b_update+0x21>
   }
```

Figure 11: Vulnerability in the `blake2b_update` function

In summary, despite the memory consumption issues encountered while using the oo7 tool to analyse more complex code, the BAP analysis within the oo7 framework determined that the reference code of the Blake2b library is susceptible to spectre attacks. Consequently, it is crucial to address these vulnerabilities by employing mitigation techniques to safeguard the library against such attacks.

4.1.2 Jasmin Code

The Jasmin framework incorporates various flags to verify compliance in Jasmin code. The `-checkSCT` flag specifically examines violations of the speculative constant-time property (SCT), as described in section 2.3.2. This flag ensures that control flow and memory accesses remain independent of secret data during speculative execution. Thus, if a code violates this property, it indicates vulnerability to spectre attacks. To identify vulnerabilities in the Jasmin code, the following command will be utilised: `jasminc -checkSCT Blake2b.jazz`.

To detect spectre vulnerabilities in the Jasmin Blake2b library it will be used the code presented in A.3. This code represents a fullstack Jasmin implementation of the reference code depicted in A.1. In this implementation, external entities can solely use the `blake2b` function to create the hash for an input message. It is impossible for the programmer to use the `update` or `init` function in its own. The output of the command shown above can be seen in Figure 12.

The type system distinguishes three types of variables: transient, public, and secret variables. Public

```
"Blake2b.jazz", line 441 (1-43):
speculative constant type checker: keylen has type (n: public, s: secret) but should be at most (n: public, s: public)
```

Figure 12: Output from `-checkSCT` flag

variables are independent of un-trusted data and are accessible to all. Therefore, any variable used in conditions must be public to ensure that the program does not violate the SCT property. Private variables, on the other hand, are inaccessible to anyone and should not be used in branch conditions. Lastly, transient variables depend on un-trusted data but are employed in branch conditions.

The output distinguishes public variables as `(n:public, s:public)`. This means that a variable is considered public in both normal and speculative execution. Private variables are identified as `(n:private, s:private)`, indicating that they remain private in both modes of execution. Transient variables are denoted as `(n:public, s:secret)`. In normal execution, these variables are public but may depend on secrets under speculative execution, since they depend on input values.

The output displayed in 12 indicates that in line 441 of the Jasmin code, there is a variable named `keylen` that should be public but is instead identified as transient. The code snippet below illustrates the vulnerable section of code, with line 441 corresponding to line 9 in the provided listing. In this snippet, the `init` function employs various input variables, including `keylen`. The usage of `keylen` within a branch condition in this function renders it as transient, as it is accessible to external entities but it is treated as private during speculative execution of the branch condition. If the `keylen` variable is considered private in speculative mode, it becomes susceptible to manipulation through spectre attacks. In such scenarios, attackers can exploit this vulnerability to influence the branch condition and potentially gain unauthorised access to private data. Consequently, it is crucial to fortify this specific section of the code against spectre attacks by leveraging the primitives offered by the type system.

```
1  export fn blake2b(reg u64 in inlen, reg u64 out outlen , reg u64 key
       keylen) {
2
3      reg u64 outlenCpy;
4      stack u64[8] h;
5      stack u64[2] t;
6
7      outlenCpy = outlen;
8
9      h,t = init(outlenCpy, keylen, key, inlen);
```

```
10
11      h = update(in, inlen, h, t);
12
13      final(out, outlenCpy, h);
14 }
```

4.2 Mitigating Vulnerabilities

The subsequent section provides a comprehensive overview of the mitigation process used to address spectre vulnerabilities in the Blake2b library. To mitigate these vulnerabilities in the reference code written in C, it will be employed the tool Blade 2.2.6. Furthermore, the type system from the Jasmin framework will be used to solve potential violations of the speculative constant-time policy in the Jasmin code. These approaches ensure that both programs become resilient against spectre attacks.

4.2.1 Reference Code

The Blade tool, as it was presented in 2.2.6, automatically mitigates spectre attacks in C code by efficiently adding lfences or Speculative Load Hardening (SLH) instructions to its assembly code. Thus, this tool will be used to mitigate the spectre vulnerabilities found in the reference code from the Blake2b library. The first section presents a test code that is going to be used in order to better understand the behaviour of the blade tool, while the second section will apply this tool to the reference code from the Blake2b library.

Following section 2.2.6, the Blade tool could implement into the program's assembly code two types of protecting statements: fence and Speculative Load Hardening (SLH) instructions. This study will focus its attention on the addition of fence instructions since the usage of SLH instructions could require more code insertions into more locations which might end up decreasing the program's performance in comparison with the addition of fence instructions, according to Vassena et al. [2020].

At its essence, the Blade repair algorithm primarily deals with spectre v1 attacks that stem from PHT mispredictions. Nevertheless, this tool has been enhanced with supplementary mitigation methods to combat spectre v1.1 as well. In the context of this research, spectre v1.1 is identified as Spectre-PHT attacks as it is discussed in section 2.1.1. For the purposes of this study, the algorithm chosen is the one designed to counter spectre v1.1., since this particular algorithm effectively addresses both spectre v1 and v1.1, encompassing the realm of Spectre-PHT attacks in its entirety.

The Blade tool is publicly available on their Github page[2] for anyone to install and use. In order to streamline the installation process, a Dockerfile was created (as shown in B.2) that includes all the necessary commands to fully install the tool within a docker container. In order to maintain the same analysis environment as the one use for the oo7 tool, the Blade tool was also installed on the remote machine glorfindel.

Test Code

To test the tool and to better understand how this tool works before running the reference code of Blake2b, it was used a similar code from the 4.1.1 as shown below.

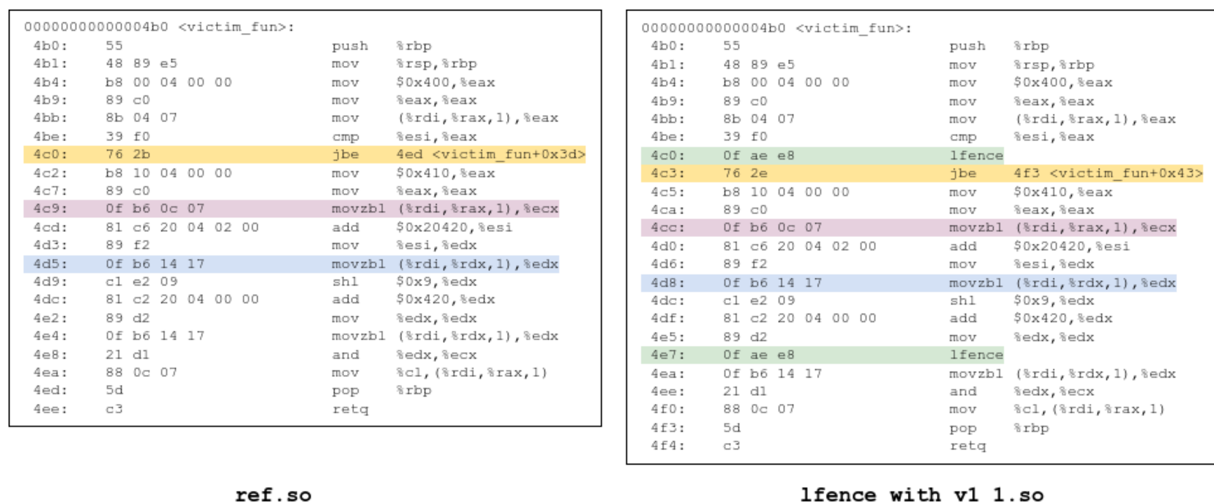
```
1  #include <stdint.h>
2
3  unsigned int array1_size = 16;
4  uint8_t array1[16];
5  uint8_t array2[256 * 512];
6  uint8_t temp = 0;
7
8
9  void victim_fun(int idx) {
10     if (idx < array1_size) {
11         temp &= array2[array1[idx] * 512];
12     }
13 }
14
15 int main(int argn, char* args[]) {
16
17     victim_fun(argn);
18     return 0;
19 }
```

As discussed in section 4.1.1, the code shown above has a spectre vulnerability within the lines 16 through 17. To mitigate the vulnerability found in this *test* code, it was created a Makefile, as shown in B.3 that compiles all the necessary steps for the Blade tool to mitigate this vulnerability by adding fence instructions. After running this Makefile with the command `make build`, all the files are created

[2] https://github.com/PLSysSec/blade

including the files `ref.so` and `lfence_with_v1_1.so`. The first file creates the binary reference code and the second code creates the binary code with the necessary fence instructions. By using the command `objdump -S lfence_with_v1_1.so` it creates the assembly code shown in the link.

The Figure 13 shows the victim function for the reference code and for the mitigated code. The reference code (ref.so) presents the vulnerable pattern in the *victim* function. This pattern was detected by the oo7 tool and it was discussed in detail in the section 4.1. The Blade tool, as seen in the `lfence_with_v1_1.so` file, also detected this vulnerable pattern within this *test* code. To mitigate this vulnerability the tool added two lfences, in order to stop speculative execution between those two instructions. The addition of these two lfences is an essential and highly efficient approach to mitigate the spectre vulnerabilities present in this code by using fence instructions.



```
00000000000004b0 <victim_fun>:
 4b0:   55                      push   %rbp
 4b1:   48 89 e5                mov    %rsp,%rbp
 4b4:   b8 00 04 00 00          mov    $0x400,%eax
 4b9:   89 c0                   mov    %eax,%eax
 4bb:   8b 04 07                mov    (%rdi,%rax,1),%eax
 4be:   39 f0                   cmp    %esi,%eax
 4c0:   76 2b                   jbe    4ed <victim_fun+0x3d>
 4c2:   b8 10 04 00 00          mov    $0x410,%eax
 4c7:   89 c0                   mov    %eax,%eax
 4c9:   0f b6 0c 07             movzbl (%rdi,%rax,1),%ecx
 4cd:   81 c6 20 04 02 00       add    $0x20420,%esi
 4d3:   89 f2                   mov    %esi,%edx
 4d5:   0f b6 14 17             movzbl (%rdi,%rdx,1),%edx
 4d9:   c1 e2 09                shl    $0x9,%edx
 4dc:   81 c2 20 04 00 00       add    $0x420,%edx
 4e2:   89 d2                   mov    %edx,%edx
 4e4:   0f b6 14 17             movzbl (%rdi,%rdx,1),%edx
 4e8:   21 d1                   and    %edx,%ecx
 4ea:   88 0c 07                mov    %cl,(%rdi,%rax,1)
 4ed:   5d                      pop    %rbp
 4ee:   c3                      retq
```

```
00000000000004b0 <victim_fun>:
 4b0:   55                      push   %rbp
 4b1:   48 89 e5                mov    %rsp,%rbp
 4b4:   b8 00 04 00 00          mov    $0x400,%eax
 4b9:   89 c0                   mov    %eax,%eax
 4bb:   8b 04 07                mov    (%rdi,%rax,1),%eax
 4be:   39 f0                   cmp    %esi,%eax
 4c0:   0f ae e8                lfence
 4c3:   76 2e                   jbe    4f3 <victim_fun+0x43>
 4c5:   b8 10 04 00 00          mov    $0x410,%eax
 4ca:   89 c0                   mov    %eax,%eax
 4cc:   0f b6 0c 07             movzbl (%rdi,%rax,1),%ecx
 4d0:   81 c6 20 04 02 00       add    $0x20420,%esi
 4d6:   89 f2                   mov    %esi,%edx
 4d8:   0f b6 14 17             movzbl (%rdi,%rdx,1),%edx
 4dc:   c1 e2 09                shl    $0x9,%edx
 4df:   81 c2 20 04 00 00       add    $0x420,%edx
 4e5:   89 d2                   mov    %edx,%edx
 4e7:   0f ae e8                lfence
 4ea:   0f b6 14 17             movzbl (%rdi,%rdx,1),%edx
 4ee:   21 d1                   and    %edx,%ecx
 4f0:   88 0c 07                mov    %cl,(%rdi,%rax,1)
 4f3:   5d                      pop    %rbp
 4f4:   c3                      retq
```

ref.so **lfence_with_v1_1.so**

Figure 13: Mitigation for spectre vulnerabilities using the lfence approach by the Blade tool

Blake2b library

To run the Blade tool using the Blake2b library, it was created a Makefile similar to the one presented in B.3, but instead of using the *test* code it was used the code presented in A.2. There was made some changes to this file in comparison to the one presented in A.1 since the Blade tool is not able to read external inputs from a file, as shown in the `main` function of this code. This function was changed in order for the tool to take as input variables from the command line which come from an un-trusted source. It was also removed from this new file, the function `blake2b_long` since the output of the Blade tool became easier to read and understand without this function which does not affect the outcome of the program.

As mentioned in 4.1, the vulnerabilities in the reference code of the Blake2b library, can be found in the `blake2b_update` and `blake2b_final` function. To mitigate these vulnerabilities by adding

fence instructions, the Blade tool needs to add these instructions throughout the code so it is completely protected against spectre attacks. To abbreviate the discussion of the mitigation result, the study will focus on the fence instructions added to the `blake2b_update` and `blake2b_final` function. In the link it is shown the complete assembly code with all the fence instructions added to mitigate spectre vulnerabilities.

Figure 14 shows portion of the `blake2b_update` function that was mitigated using fence instructions. The yellow section in the file `ref.so` corresponds to the following code snippet from the reference code.

```
1 int blake2b_update(blake2b_ctx *ctx, const void *in, size_t inlen) {
2     ...
3         if (ctx->c == 128) {
4             ctx->t[0] += ctx->c;
5             if (ctx->t[0] < ctx->c)
6             ...
7         }
8     ...
9 }
```

The Blade tool considered that these two conditions use vulnerable variables which could be manipulate to be invalid allowing for attackers to leak private information under speculative execution. Hence, in accordance with the details provided in Figure 14, Blade incorporated fence instructions preceding the initial condition and following the second. This strategic inclusion effectively prevents the CPU from speculatively executing this particular code segment by requiring the prior instructions to complete their execution before entry into the branch becomes viable. As a result, potential attackers are unable to exploit out-of-bounds variables to access both conditional branches.

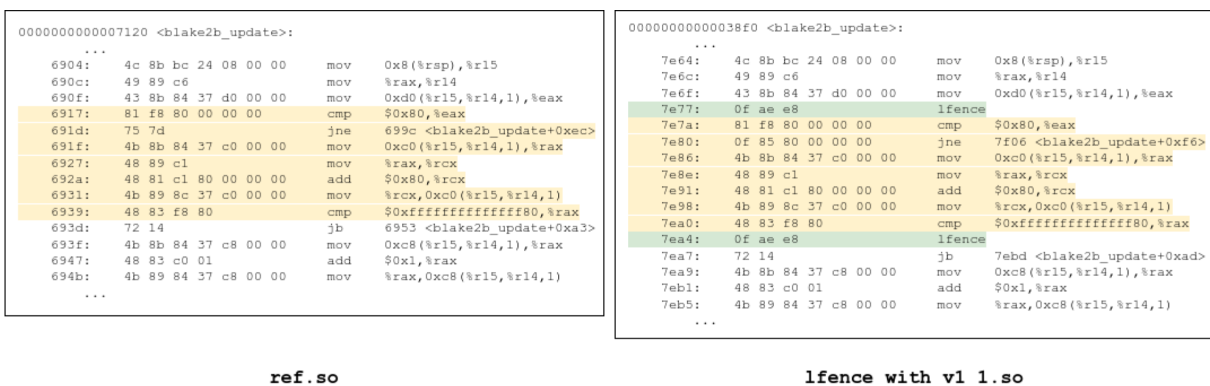


Figure 14: Mitigation for spectre vulnerabilities found in `blake2b_update` function

Figure 16 shows portion of the `blake2b_final` function that was mitigated using fence instructions. Both blue, yellow and red sections from the `ref.so` file correspond to the code snippets shown in Figure 15.

```
int blake2b_final(blake2b_ctx *ctx, void *out){
    size_t i;

    ctx->t[0] += ctx->c;

    if (ctx->t[0] < ctx->c)
        ...
    ...
}
```

```
int blake2b_final(blake2b_ctx *ctx, void *out){
    ...
    while (ctx->c < 128)
        ...

    blake2b_compress(ctx, 1);

    for (i = 0; i < ctx->outlen; i++)
        ...
    ...
}
```

```
int blake2b_final(blake2b_ctx *ctx, void *out){
    ...
    for (i = 0; i < ctx->outlen; i++){
        ...
    }
    ...
}
```

Figure 15: Vulnerable code snippets from the `blake2b_final` function

In the blue section, it can be seen that Blade adds two fence instructions to stop speculative execution before the branch condition. This way it is possible that all variables inside this branch condition are valid and thus, not allowing attackers to leak confidential data. The yellow section corresponds to the end of the `while` loop and the begin of the `for` loop. The addition of fence instructions in this spots allows for the call to the `blake2b_compress` function and for the storage of all the variables used in this called to never be execute under speculative execution. Finally, the red section corresponds to the end of the `for` loop which allows the program to exit the `for` loop correctly by executing the final jump after the condition completely executes.

Besides the fences added to the `blake2b_update` and the `blake2b_final` function, the Blade tool also adds other fences in other functions and locations. In total the Blade tool adds a total of 304 fence instructions throughout the code, in locations different than the ones mentioned as vulnerable by the oo7 tool. However, not all of these fence instructions were added to the main functions of this code (`blake2b_init`, `blake2b_update`, `blake2b_final` and `blake2b`). The Blade tool in order to mitigate code written in C creates new functions which represents other functions that are called within the implementation of the main functions. For example, the `guest_func_2` as shown in **??** is called in `blake2b_init`, `blake2b_update`, `blake2b_final` and `blake2b` and represents all of these functions aside from the `blake2b_compress`. Therefore, this new function created by Blade represents all the other main functions which forces the tool to also mitigate the `guest_func_2` function which would be unnecessary if the the code would jump to their original functions instead of this created one.

```
00000000000010b0 <blake2b_final>:
      ...
    54b6:    49 89 ff                  mov    %rdi,%r15
    54b9:    49 8b 8c 07 c0 00 00      mov    0xc0(%r15,%rax,1),%rcx
    54c1:    41 8b 94 07 d0 00 00      mov    0xd0(%r15,%rax,1),%edx
    54c9:    89 d3                     mov    %edx,%ebx
    54cb:    48 89 ce                  mov    %rcx,%rsi
    54ce:    48 01 de                  add    %rbx,%rsi
    54d1:    49 89 b4 07 c0 00 00      mov    %rsi,0xc0(%r15,%rax,1)
    54d9:    48 39 ce                  cmp    %rcx,%rsi
    54dc:    73 21                     jae    54ff <blake2b_final+0x7f>
    54de:    44 8b bc 24 0c 00 00      mov    0xc(%rsp),%r15d
      ...
    5543:    4d 89 f7                  mov    %r14,%r15
    5546:    41 8b 8c 07 d0 00 00      mov    0xd0(%r15,%rax,1),%ecx
    554e:    81 f9 80 00 00 00         cmp    $0x80,%ecx
    5554:    72 b0                     jb     5506 <blake2b_final+0x86>
    5556:    b8 01 00 00 00            mov    $0x1,%eax
    555b:    4c 8b bc 24 00 00 00      mov    0x0(%rsp),%r15
    5563:    44 8b b4 24 0c 00 00      mov    0xc(%rsp),%r14d
    556b:    4c 89 ff                  mov    %r15,%rdi
    556e:    44 89 f6                  mov    %r14d,%esi
    5571:    40 89 c2                  rex mov %eax,%edx
    5574:    e8 e7 eb ff ff            callq  4160 <guest_func_2>
    5579:    44 8b bc 24 0c 00 00      mov    0xc(%rsp),%r15d
    5581:    44 89 f8                  mov    %r15d,%eax
    5584:    4c 8b bc 24 00 00 00      mov    0x0(%rsp),%r15
    558c:    41 8b 8c 07 d4 00 00      mov    0xd4(%r15,%rax,1),%ecx
    5594:    85 c9                     test   %ecx,%ecx
    5596:    74 7d                     je     5615 <blake2b_final+0x195>
    5598:    44 8b bc 24 10 00 00      mov    0x10(%rsp),%r15d
      ...
    5603:    83 c2 01                  add    $0x1,%edx
    5606:    4d 89 e7                  mov    %r12,%r15
    5609:    41 8b 9c 07 d4 00 00      mov    0xd4(%r15,%rax,1),%ebx
    5611:    39 da                     cmp    %ebx,%edx
    5613:    72 90                     jb     55a5 <blake2b_final+0x125>
    5615:    44 8b bc 24 10 00 00      mov    0x10(%rsp),%r15d
    561d:    44 89 f8                  mov    %r15d,%eax
      ...
```

ref.so

```
0000000000000680 <blake2b_final>:
      ...
    7c76:    49 89 ff                  mov    %rdi,%r15
    7c79:    49 8b 8c 07 c0 00 00      mov    0xc0(%r15,%rax,1),%rcx
    7c81:    41 8b 94 07 d0 00 00      mov    0xd0(%r15,%rax,1),%edx
    7c89:    0f ae e8                  lfence
    7c8c:    89 d3                     mov    %edx,%ebx
    7c8e:    48 89 ce                  mov    %rcx,%rsi
    7c91:    48 01 de                  add    %rbx,%rsi
    7c94:    49 89 b4 07 c0 00 00      mov    %rsi,0xc0(%r15,%rax,1)
    7c9c:    48 39 ce                  cmp    %rcx,%rsi
    7c9f:    0f ae e8                  lfence
    7ca2:    73 21                     jae    7cc5 <blake2b_final+0x85>
    7ca4:    44 8b bc 24 0c 00 00      mov    0xc(%rsp),%r15d
    7cac:    41 81 c7 c8 00 00 00      add    $0xc8,%r15d
      ...
    7d09:    4d 89 f7                  mov    %r14,%r15
    7d0c:    41 8b 8c 07 d0 00 00      mov    0xd0(%r15,%rax,1),%ecx
    7d14:    0f ae e8                  lfence
    7d17:    81 f9 80 00 00 00         cmp    $0x80,%ecx
    7d1d:    72 ad                     jb     7ccc <blake2b_final+0x8c>
    7d1f:    b8 01 00 00 00            mov    $0x1,%eax
    7d24:    4c 8b bc 24 00 00 00      mov    0x0(%rsp),%r15
    7d2c:    44 8b b4 24 0c 00 00      mov    0xc(%rsp),%r14d
    7d34:    4c 89 ff                  mov    %r15,%rdi
    7d37:    44 89 f6                  mov    %r14d,%esi
    7d3a:    40 89 c2                  rex mov %eax,%edx
    7d3d:    e8 8e b9 ff ff            callq  36d0 <guest_func_2>
    7d42:    44 8b bc 24 0c 00 00      mov    0xc(%rsp),%r15d
    7d4a:    44 89 f8                  mov    %r15d,%eax
    7d4d:    4c 8b bc 24 00 00 00      mov    0x0(%rsp),%r15
    7d55:    41 8b 8c 07 d4 00 00      mov    0xd4(%r15,%rax,1),%ecx
    7d5d:    0f ae e8                  lfence
    7d60:    85 c9                     test   %ecx,%ecx
    7d62:    0f 84 80 00 00 00         je     7de8 <blake2b_final+0x1a8>
    7d68:    44 8b bc 24 10 00 00      mov    0x10(%rsp),%r15d
      ...
    7dd3:    83 c2 01                  add    $0x1,%edx
    7dd6:    4d 89 e7                  mov    %r12,%r15
    7dd9:    41 8b 9c 07 d4 00 00      mov    0xd4(%r15,%rax,1),%ebx
    7de1:    39 da                     cmp    %ebx,%edx
    7de3:    0f ae e8                  lfence
    7de6:    72 8d                     jb     7d75 <blake2b_final+0x135>
    7de8:    44 8b bc 24 10 00 00      mov    0x10(%rsp),%r15d
    7df0:    44 89 f8                  mov    %r15d,%eax
      ...
```

lfence_with_v1_1.so

Figure 16: Mitigation for spectre vulnerabilities found in `blake2b_final` function

In addition, the Blade tool also mitigates outside functions that are called within the main functions, like `strlen`, `realloc`, `memcpy`, among others. In summary, the Blade tool adds 21 fence instructions between all the main functions, 12 in the `blake2b` function, 5 in the `blake2b_final` function and 2 in the `blake2b_init` and `blake2b_update` function.

4.2.2 Jasmin Code

As it was presented in the section 4.1.2, the flag `-checkSCT` shows all the speculative constant-time violations in a program, which implies that a program has spectre vulnerabilities. For the Jasmin program presented in A.3, the flag alerted for a vulnerability in line 441 as shown in Figure 12.

To mitigate this vulnerability will be necessary to use the type system primitives that were presented in section 2.3.2. As presented in this section there are three primitives (`#init_msf`, `#set_msf` and `#protect`) that combine mitigate spectre vulnerabilities in Jasmin code. However, in order to mitigate the vulnerability found in the Blake2b Jasmin code it is necessary to only use the primitive `#init_msf`, as seen below.

```
1  export fn blake2b(reg u64 in inlen, reg u64 out outlen , reg u64 key
```

```
    keylen) {
2
3     reg u64 outlenCpy;
4   stack u64[8] h;
5   stack u64[2] t;
6   #msf reg u64 ms;
7
8   outlenCpy = outlen;
9
10  _ = #init_msf();
11  h,t = init(outlenCpy, keylen, key, inlen);
12
13  h = update(in, inlen, h, t);
14
15  final(out, outlenCpy, h);
16 }
```

As explained previously, all the variables that depend on input from outside entities are normally considered transient by the type system, therefore variables like `inlen`, `in`, `out`, `outlen`, `key` and `keylen` that are taken from user input are considered private values under speculative mode and need to be carefully protected since their use in branch conditions could lead to spectre vulnerabilities. In order for those variables to be considered public under speculative mode, it is necessary to implement the primitive `#init_msf`. This primitive as stated above, creates a `lfence` in the assembly code, which settles every variable to public in the type system. Therefore, when the execution arrives at the fence instruction it waits for previous instructions to be complete, making all transient variables public after the fence instruction. This allows the variables to be used in branch conditions making the branches protected against spectre attacks. This simple primitive solves all the spectre vulnerabilities that might occur in the program without the need to use the rest of the primitives in the type system. The assembly code below shows a snippet of the blake2b function in assembly after the addition of the primitive `#init_msf`. As seen in the cod, this primitive translates to a fence instruction that protects the code against spectre attacks.

```
1 blake2b :
2     movq %rsp , %rax
3     leaq -352(% rsp), %rsp
```

```
4       andq $-8, %rsp
5       movq %rax , 336(% rsp)
6       movq %rbp , 344(% rsp)
7       movq %rcx , %rax
8       lfence
9       movq $0 , %rcx
10      movq $7640891576956012808 , %rcx
11      movq %rcx , 16(% rsp)
12      ...
```

Lastly, by using the flag referenced in section 4.1.2, the system can verify whether this protective statement effectively secures the program against Spectre vulnerabilities. By executing the command `jasminc -checkSCT Blake2b.jazz`, the subsequent output is displayed.

```
constraints:
```

Figure 17: Output from `-checkSCT` flag after the mitigation

Figure 17 shows that there are no violations of the SCT property, meaning that the program is protected against spectre attacks.

Chapter 5

Evaluation and Comparison of Results

This chapter is dedicated to the evaluation and comparison the results obtained by using the Jasmin Framework and the Blade tool to mitigate the spectre vulnerabilities found in the reference code and in the Jasmin version of the Blake2b library. The purpose of this evaluation is to establish which tool or framework is better in terms of efficiency, security, complexity, and knowledge level. This chapter starts by establishing the criteria to be employed in the comparison of the Jasmin Framework and the Blade tool. This will be followed by an analysis of each tool in relation to the other. Ultimately, this study will analyse comparatively the Jasmin framework and the Blade tool taking into account the criteria discussed in the following section.

5.1 Comparison criteria

The subsequent sections delineate distinct criteria intended for comparing the mitigation outcomes and strategies employed within the Jasmin Framework and the Blade tool.Each section will detail the purpose of the respective criterion, explain the methodology for its measurement, and conclude with a comparative assessment of both tools.

5.1.1 Availability and Installation Process

This criteria evaluates the installation process of the Jasmin framework and the Blade Tool and determines the availability of these tools for the public. For this, the study will focus on the amount of time that took to install and compile both tools including the amount of compile errors that were found in order to test and use both tools.

The Jasmin framework is publicly available on their Github page[1]. Installing this framework merely requires cloning the Git repository and setting up the Nix shell. Once this is done, the framework and its type

[1] https://github.com/jasmin-lang/jasmin

system becomes accessible for anyone to write cryptography code or for mitigate spectre vulnerabilities in a Jasmin program.

The Blade tool as mentioned previously is available on their Github page. To install the tool it is necessary to clone the git repository from this tool and from lucet, WASI-SDK, WABT, Binaryan, and HACL tool. In order to compile this tool, first it is fundamental that all of these aforementioned tools also be compiled according to their Github. The developers created a Makefile that compiles and runs all the benchmarks stated in Vassena et al. [2020] by using the compiled programs from the other tools. However, when trying to compile using this Makefile, it occurs an error that seemed to be connected with the WASI-SDK documentation, since the WASI-SDK could not find a library that was mentioned in one of the benchmarks . After a long time trying to solve the problem, it was added the following line `--sysroot=$(WASI_SDK)/wasi-sdk-12.0/share/wasi-sysroot` which allows for the wasi-sdk-12.0 to search for libraries outside the ones in the WASI-SDK folder. After solving this problem, it was possible to run the tool with different benchmarks.

The installation processes for both tools differed significantly. Setting up the Jasmin framework was smooth, with no compilation errors encountered. However, the Blade tool encountered compilation errors that required a substantial amount of time to resolve. Upon examining their GitHub pages in more detail, it became apparent that the Jasmin framework receives more frequent updates compared to the Blade tool. The latest commit for the Blade tool was made three years ago, whereas the Jasmin framework had a commit just a few days ago. This discrepancy becomes particularly relevant when aiming to maintain versions of these tools that are compatible with the versions of other tools that are used in both the Jasmin framework and the Blade tool.

5.1.2 Level of Knowledge

This indicator was created to assess the amount of technical knowledge it is necessary for a programmer to have in order to use the Jasmin framework and the Blade Tool. This criteria will evaluate how easy it is to use both tools, by comparing their implementation strategy (automatic or manual) and by reasoning about the necessary programming skills.

To mitigate spectre vulnerabilities in a cryptography function with the Jasmin framework it is necessary to add the type system primitives to the Jasmin version of the function. For that, it is necessary for the user to write the cryptography function in the Jasmin language and then it needs to add the primitives to particular location within the code. To accomplish this, the user must possess an intricate understanding of the Jasmin language, along with a comprehensive grasp of the nuanced application of the type system.

This depth of understanding is essential for using the primitives in the most optimal manner.

In the Blade tool, the mitigation process is done automatically by the tool. The user needs to give to the tool the cryptography code written in C or WebAssembly, which is easily found online. The tool, then automatically mitigates all the spectre vulnerabilities found without any interaction from the user.

For a user aiming to mitigate Spectre vulnerabilities within a cryptography code using the Jasmin framework, the initial step involves dedicating time to comprehend the intricacies of the Jasmin framework, including its type system. In contrast, when dealing with the Blade tool, the user can simply use a cryptography code written in C or WebAssembly, for Blade to then automatically apply the necessary protecting statements to the code.

5.1.3 Mitigation customisation

The criteria for Mitigation Customisation assesses the extent to which a user can tailor the mitigation strategy within both the Jasmin framework and the Blade Tool. This involves evaluating the degree of programming effort required for a user to effectively mitigate a program.

As previously mentioned, the Jasmin framework requires users to manually incorporate mitigation primitives from the type system into the code. Consequently, there exists a substantial degree of customisation available to users. This is due to the freedom to introduce these primitives at locations designated by the user. This level of control empowers users to tailor the mitigation approach according to their preferences, thereby enabling the mitigation to be as customised as the user desires.

Since the Blade tool automatically inserts protective statements without requiring any user input, it lacks the ability to adapt these insertions. As a result, users are unable to personalise the mitigation technique provided by the Blade tool.

5.1.4 Mitigation Strategy

This criteria measure the amount of protecting statements added to the mitigated program by the Jasmin framework and the Blade Tool. Considering that the number of protecting statements added to the code, could have a toll on the performance and efficiency, the less protecting statements allows for a possible more efficient code.

The Jasmin framework according to the assembly snippet found in 4.2.2 has 1 fence instruction, which was added because of the `#init_msf()` primitive as discussed previously, while the blade tool inserted 21 fence instructions into the main functions of the assembly code found in **??** as explained in 4.2.1. Therefore, the Jasmin framework provides a more efficient method for inserting protecting statements

compared to the Blade tool.

5.1.5 Efficiency and Performance

This metric quantifies the time required for the program, when mitigated using either the Jasmin framework or the Blade Tool, to accomplish its task, as opposed to the unmitigated version. To measure this performance it will be used the same environment as the one that the Blade tool was installed, in the glorfindel machine mentioned in section 4.1. For the inputs to be used in both the Jasmin framework and the Blade Tool, the input message will be "cryptography Component - Blake2b hash function" and the key will be "Blake2b Project".

To calculate the time that it takes to create an hash function with the mitigated Jasmin code, it will be use the Linux command `time` which determines how long a specific command will take to run and gives as output three measurements. The first is the real-life time it takes for the process to run from start to finish (real). The second one is the amount of CPU time spent in user mode during the process (user). The final one measures the total CPU time spent in kernel mode during the process (sys). For this study it will be used the real measurement since it includes all time spent waiting for I/O and other processes, making this the most important measurement of the three.

The Table 5 shows the performance measurements for the the reference Jasmin code and for the Jasmin mitigated code. As seen in the table the reference Jasmin code takes in average 0,0016 seconds to complete the task, while the protected Jasmin code takes 0,0023 seconds. The addition of the fence instructions cost 0,0007 seconds in performance, which is quite insignificant and establishes that the protected version of the Jasmin implementation does not take a toll on performance, making this version feasible to be used.

	Measure 1	Measure 2	Measure 3	
Reference	0,001 s	0,001 s	0,003 s	0,0016 s
Protected	0,001 s	0,003 s	0,003 s	0,0023 s

Table 5: Performance measurements for Jasmin Blake2b version

For the Blade tool to measure the performance of the protected code, it is necessary to create a Rust program that measures the time that it takes to run the mitigated code with the defined inputs. The main two files for this program can be found in B.4.

The Table 6 shows the performance measurements for the the reference C code and for the protected code by using the Blade tool. As seen in the table the reference code takes in average 0,0031 seconds to

complete the task, while the protected code takes 0,0055 seconds. The addition of the fence instructions cost 0,0024 seconds in performance, which in comparison to the Jasmin version is more significant. However, the results establish that the protected version does not take a massive toll on performance, making this version also feasible to be used.

	Measure 1	Measure 2	Measure 3	
Reference	0,0031 s	0,0031 s	0,0031 s	0,0031 s
Protected	0,0055 s	0,0055 s	0,0055 s	0,0055 s

Table 6: Performance measurements for Blade Blake2b version

5.1.6 Security

This final criteria will evaluate if the Jasmin framework and the Blade Tool adds enough protecting statements in order to protect the code against spectre attacks. It also reasons about the consistency of the mitigated program, that is, if the insertion of the protecting statements do not change the output of the program and will analyse other relevant properties from both tools. To this this evaluation the study will take into account some mathematical proofs developed by Vassena et al. [2020] and by Shivakumar et al. [2022].

According to Shivakumar et al. [2022] the type system from the Jasmin framework is considered to be sound and expressive. The article mathematically proves that the type system is sound since it only accepts speculative constant-time programs which implies that control-flow and memory accesses be independent of secret data during speculative execution, after the insertion of the type system primitives. This article also proves how expressive the type system is. It asserts the type system's primitives could be added to any Jasmin program, making any Jasmin program typable by the type system.

The Vassena et al. [2020] proves that Blade is consistent and sound. Blade is consistent since the programs that were protected by the Blade processor produce the same results and output as the program without the protect statements. The article also proves that Blade's type system is sound since the addition of the protect statements enforce constant time under speculative executions, similar to the Jasmin type system.

In line with the mathematical proofs shown in both articles, the Jasmin framework and Blade make the program secure against spectre attacks by having mitigating algorithms that automatically or manually insert enough protecting statements.

5.2 Comparative analysis

The Table 7 summarises all the comparison criteria between the Jasmin and Blade presented in the last section.

	Jasmin	Blade
Available	Yes	Yes
Installation	Easy	Complex
Skill Level	High	Low
Mitigation Strategy	Manual	Automatic
Protecting Statements	1	21 (in main functions)
Performance	0,0023 s	0,0055 s
Security	Yes	Yes

Table 7: Summary of the criteria results

When it comes to installation, Jasmin framework is way easier to install and compile than Blade and it also adds less number of fence instructions than Blade which takes a toll on performance. When comparing the performance results, it is evident that Jasmin reference code executes faster than the C version of the code, since the Jasmin language it was build to implement high-speed cryptography code. In addition, the protected code using the Jasmin type system tends to have a better performance than the code protected by Blade, not only because the Jasmin code executes faster than the C code but also because the Jasmin type system enables a more judicious placement of less protecting statements than the Blade tool.

The Blade tool also presents its own advantages, since it allows for users with a low level of technical skills to use the tool without needing to understand to the fullest how this tool works. Seeing that the Blade tool implements an automatic strategy of mitigating spectre vulnerabilities it does not allow for the user to customise the addition of protecting statements but it allows an easier mitigation strategy than the one presented by the Jasmin framework.

The table shows that both the Jasmin framework and the Blade tool are feasible to be used in a real-life setting when protecting a cryptography library, since both are available to the public and manage to protect the library by using the necessary number of fence instructions in order to make the library secure.

Chapter 6

Conclusions and future work

This final chapter presents the conclusion for this study and summaries some details about the algorithms from the tools and frameworks used. This chapter also introduces some work that could be developed and implemented in the future.

6.1 Conclusion

Software security techniques rely on the fact that processors will faithfully execute program instructions, including safety checks. Nevertheless, spectre attacks leverage the usage of speculative execution in processors to violate this assumption.

As it is presented in this document spectre vulnerabilities arise from a longstanding focus in the technology industry on maximising performance and as a result, many hardware components and operating systems compound layers of complex optimisations that introduce security risks. Therefore, it is mandatory to apply countermeasures to mitigate these vulnerabilities so that no secret data is leaked to outside entities, compromising programs and entire systems. While there are many tools and techniques that protect systems against spectre attacks, it is still difficult to find an optimal solution that does not hinder performance.

One solution that tries to develop high-assurance and high-speed cryptography code without decreasing performance is the framework Jasmin and its type system. This type system uses three main primitives that compile into assembly-level instructions in order to create cryptography algorithms that are protected against spectre attacks. To use these primitives, the user needs to first identify transient variables within the code, that is, variables whose value speculative depends on secrets. These primitives work together in order to track whether the program is being executed in speculative or normal mode. If a program is being executed in speculative mode, then the value of a transient variable is discarded for it not to be leaked and thereby, protecting the program against spectre attacks.

Another solution called Blade, automatically eliminates spectre vulnerabilities in a cryptography code written in a WebAssembly or C. It starts identifying all vulnerable expressions and possible ways of leaking secret data. Then, it creates a def-use graph, whose edges capture the data dependencies between the expressions and variables of a program, in order to track how secret data could flow within the exaction of a program. To stop the flow of such secret data, the Blade tool either inserts fences or Speculative Load Hardening instructions at cut points in the graph that minimise the usage of these protect primitives. Consequently, this algorithm protects the program efficiently against spectre attacks.

To compare both solutions, that was used the hash function library, Blake2b. Using the tool oo7, which focus on identifying code patterns on program binaries that are vulnerable to spectre attacks, it was demonstrated that the library was vulnerable to spectre attacks. Then, in the Jasmin version of this library was implemented the Jasmin type system in order to mitigate the code and the C version was used by the Blade tool to automatically protect the code against spectre attacks.

Both tools could be easily used in a real-life setting for different proposes. If the user has an high technical skill level then the Jasmin framework might be the best suit, since it allows for customisation in the addition of the protecting statements and has the best results performance wise. For users with a low technical level the Blade tool also protects the program with a small performance overhead.

6.2 Prospect for future work

This study was focused on comparing the Jasmin framework and its type system with the Blade tool by using the Blake2b library. To accomplish this, was used the Blake2b library written in Jasmin and the C version of the same library. This section focus on presenting possible future approaches to this study that could lead to valuable results which would allow for this study to become more complete. The following sections introduce different approaches to this study and ways that could improve future work.

6.2.1 Analyse the addition of SLH instructions

As stated previously, the Blade tool could either insert fence and speculative load hardening (SLH) instructions. The study was focus on the insertion of fence instructions since the usage of SLH instructions could require more code insertions into more locations which might end up decreasing the program's performance in comparison with the addition of fence instructions. However, it would be beneficial to this study to better understand the impact of adding SLH instructions to the Blake2b library and comparing the results to addition of fence instructions to study the change of performance between these two protecting

statements.

6.2.2 Analyse the WebAssembly version of the library

The Blade tools allows the user to protect C and WebAssembly code against spectre attacks. Since the begin of this study that the focus was to compare the C and Jasmin version of the Blake2b library, however WebAssembly code allows for more statically allocated variables and arrays in the memory which results in many constant-address loads, opposite to the the C version where variables are constantly changing its address in memory. This difference could lead to less insertion of protecting statements which might result in better performance for the WebAssembly protected code. Therefore, comparing the C and WebAssembly version of the Blake2b library could improve the conclusions of this study.

6.2.3 Analyse other tools and libraries

This study only compares the Jasmin Framework and the Blade tool by using the Blake2b library, nonetheless this study would gain a more complete view of the different mitigation techniques if it compared the Jasmin tool against other techniques, like the ones presented in 2.2. The comparison results could be evaluated in detail for the study to gain a better understand of the real-life use of these tools.

Finally, it would also be very beneficial to the study if these tools and the Jasmin framework were compared against different libraries. Even though this library determined that the Jasmin framework has a better performance then the Blade tool, it could be possible that with a different library the results could reverse. By determine the reason why this might happened could improve the study and its results.

Bibliography

Jade Alglave et al. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, pages 1–44, 2016.

José Bacelar Almeida et al. Jasmin: High-assurance and high-speed cryptography. *CCS'2017, Dallas, TX, USA*, pages 1807–1823, 2017.

Jean-Philippe Aumasson et al. Blake2: simpler, smaller, fast as md5. pages 1–20, 2013. URL https://www.blake2.net/.

David Brumley, Ivan Jager, and Thanassis Avgerinos. Bap: A binary analysis platform. *International Conference on Computer Aided Verification*, pages 463–469, 2011.

Claudio Canella et al. A systematic evaluation of transient execution attacks and defenses. *28th USENIX Security Symposium*, pages 249–266, 2019.

Sunjay Cauligi et al. Constant-time foundations for new spectre era. *PLDI'20. June 15-20, 2020, London UK*, pages 913–926, 2020.

Sunjay Cauligi et al. Sok: Practical foundations for software spectre defenses. *Security Privacy*, pages 1–15, 2022.

Hernán Ponce de León et al. Cats vs spectre: An axiomatic approach to modeling speculative execution attaks. *2022 IEEE Symposium on Security and Privacy*, pages 235–248, 2022.

Shengjian Guo et al. Specusym: Speculative symbolic execution for cache timing leak detection. *ICSE'20, May 23-29, 2020, Seoul, Republic of Korea*, pages 1–14, 2020.

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

Paul Kocher et al. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

Tiago Oliveira. High-speed high-assurance cryptography. *Faculty of Sciences of the University of Porto*, page PhD Thesis, 2022.

M. Rajaram and M. Arul Then Mathi. Implementation of cryptography hash function blake 64 bit using sha-3 algorithm. *Special Issue of International Jounal of Computer Applications*, pages 18–21, 2012.

Zhuojia Shen et al. Restricting control flow during speculative execution with venkman. *arXiv:1903.1065v1*, pages 1–15, 2019.

Basavesh Ammanaghatta Shivakumar et al. Typing high-speed cryptography against spectre v1. *Cryptology ePrint Archive, Paper 2022/1270*, pages 1–20, 2022.

Marco Vassena et al. Automatically eliminating speculative leaks from cryptography code with blade. *Proceedings of ACM on Programming Languages*, pages 49–49:61, 2020.

Guanhua Wang et al. oo7: Low-overhead defence against spectre attacks via program analysis. *IEEE Transactions on Software Engineering, Vol 47, No 11*, pages 2504–2519, 2021.

Part II
Appendices

Appendix A
Listings

This chapter introduces the code files for the Blake2b library written in Jasmin, C and for the Blade tool.

A.1 Blake2b - Reference Code

This section presents the reference code written in C of the Blake2b library.

```
1  #include <stdint.h>
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  // state context
8  typedef struct {
9      uint8_t b[128]; // input buffer
10     uint64_t h[8];  // chained state
11     uint64_t t[2];  // total number of bytes
12     size_t c;       // pointer for b[]
13     size_t outlen;  // digest size
14 } blake2b_ctx;
15
16 /* Necessary to blake2b_long */
17 static void store32(void *dst, uint32_t w) {
18     #if defined(NATIVE_LITTLE_ENDIAN)
19         memcpy(dst, &w, sizeof w);
20     #else
21         uint8_t *p = (uint8_t *)dst;
22         *p++ = (uint8_t)w;
23         w >>= 8;
24         *p++ = (uint8_t)w;
25         w >>= 8;
26         *p++ = (uint8_t)w;
27         w >>= 8;
28         *p++ = (uint8_t)w;
29     #endif
30 }
31
```

```
32 #ifndef ROTR64
33 #define ROTR64(x, y)  ( ( (x) >> (y) ) ^ ( (x) << (64 - (y)) ))
34 #endif
35
36 #define B2B_GET64(p)                                   \
37 (((uint64_t) ((uint8_t *) (p))[0]) ^         \
38 (((uint64_t) ((uint8_t *) (p))[1]) << 8) ^   \
39 (((uint64_t) ((uint8_t *) (p))[2]) << 16) ^ \
40 (((uint64_t) ((uint8_t *) (p))[3]) << 24) ^ \
41 (((uint64_t) ((uint8_t *) (p))[4]) << 32) ^ \
42 (((uint64_t) ((uint8_t *) (p))[5]) << 40) ^ \
43 (((uint64_t) ((uint8_t *) (p))[6]) << 48) ^ \
44 (((uint64_t) ((uint8_t *) (p))[7]) << 56))
45
46
47 #define B2B_G(a, b, c, d, x, y) {   \
48 v[a] = v[a] + v[b] + x;         \
49 v[d] = ROTR64(v[d] ^ v[a], 32); \
50 v[c] = v[c] + v[d];             \
51 v[b] = ROTR64(v[b] ^ v[c], 24); \
52 v[a] = v[a] + v[b] + y;         \
53 v[d] = ROTR64(v[d] ^ v[a], 16); \
54 v[c] = v[c] + v[d];             \
55 v[b] = ROTR64(v[b] ^ v[c], 63); }
56
57
58 static const uint64_t blake2b_iv[8] = {
59     0x6A09E667F3BCC908, 0xBB67AE8584CAA73B,
60     0x3C6EF372FE94F82B, 0xA54FF53A5F1D36F1,
61     0x510E527FADE682D1, 0x9B05688C2B3E6C1F,
62     0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179
63 };
64
65
66 static void blake2b_compress(blake2b_ctx *ctx, int last)   {
67     //printf("Start Compress. \n");
68     const uint8_t sigma[12][16] = {
69         { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
70         { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 },
71         { 11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4 },
72         { 7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8 },
73         { 9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13 },
74         { 2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9 },
75         { 12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11 },
76         { 13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10 },
77         { 6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5 },
78         { 10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0 },
79         { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
80         { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 }
81     };
82
```

```
83      int i;
84      uint64_t v[16], m[16];
85
86      for (i = 0; i < 8; i++) {           // init work variables
87          v[i] = ctx->h[i];
88          v[i + 8] = blake2b_iv[i];
89      }
90
91      v[12] ^= ctx->t[0];                 // low 64 bits of offset
92      v[13] ^= ctx->t[1];                 // high 64 bits
93      if (last)                           // last block flag set ?
94          v[14] = ~v[14];
95
96      for (i = 0; i < 16; i++)            // get little-endian words
97          m[i] = B2B_GET64(&ctx->b[8 * i]);
98
99
100     for (i = 0; i < 12; i++) { // twelve rounds
101
102         B2B_G( 0, 4,  8, 12, m[sigma[i][0]], m[sigma[i][ 1]]);
103
104         B2B_G( 1, 5,  9, 13, m[sigma[i][2]], m[sigma[i][ 3]]);
105
106         B2B_G( 2, 6, 10, 14, m[sigma[i][4]], m[sigma[i][ 5]]);
107
108         B2B_G( 3, 7, 11, 15, m[sigma[i][6]], m[sigma[i][ 7]]);
109
110         B2B_G( 0, 5, 10, 15, m[sigma[i][8]], m[sigma[i][ 9]]);
111
112         B2B_G( 1, 6, 11, 12, m[sigma[i][10]], m[sigma[i][11]]);
113
114         B2B_G( 2, 7,  8, 13, m[sigma[i][12]], m[sigma[i][13]]);
115
116         B2B_G( 3, 4,  9, 14, m[sigma[i][14]], m[sigma[i][15]]);
117     }
118
119
120     for( i = 0; i < 8; ++i )
121         ctx->h[i] ^= v[i] ^ v[i + 8];
122
123 }
124
125
126 int blake2b_init(blake2b_ctx *ctx, size_t outlen, const void *key,
        size_t keylen) {
127     size_t i;
128     if (outlen == 0 || outlen > 64 || keylen > 64)
129         return -1;                      // illegal parameters
130
131     for (i = 0; i < 8; i++)             // state, "param block"
132         ctx->h[i] = blake2b_iv[i];
```

```
133
134     ctx->h[0] ^= 0x01010000 ^ (keylen << 8) ^ outlen;
135
136     ctx->t[0] = 0;                         // input count low word
137     ctx->t[1] = 0;                         // input count high word
138
139     ctx->c = 0;                            // pointer within buffer
140     ctx->outlen = outlen;
141
142     for (i = keylen; i < 128; i++)         // zero input block
143         ctx->b[i] = 0;
144
145     if (keylen > 0) {
146         blake2b_update(ctx, key, keylen);
147         ctx->c = 128;                      // at the end
148     }
149     return 0;
150 }
151
152 int blake2b_update(blake2b_ctx *ctx, const void *in, size_t inlen) {
153     size_t i;
154     for (i = 0; i < inlen; i++) {
155         if (ctx->c == 128) {               // buffer full ?
156             ctx->t[0] += ctx->c;           // add counters
157             if (ctx->t[0] < ctx->c)        // carry overflow ?
158                 ctx->t[1]++;               // high word
159             blake2b_compress(ctx, 0);      // compress (not last)
160             ctx->c = 0;                    // counter to zero
161         }
162         ctx->b[ctx->c++] = ((const uint8_t *) in)[i];    /* ----
    Spectre vulnerability ---- */
163     }
164     //printf("[update] c: %d\n",ctx->c);
165     return 0;
166 }
167
168 int blake2b_final(blake2b_ctx *ctx, void *out)   {
169     size_t i;
170
171     ctx->t[0] += ctx->c;                  // mark last block offset
172
173     if (ctx->t[0] < ctx->c)                // carry overflow
174         ctx->t[1]++;                       // high word
175
176     while (ctx->c < 128)                   // fill up with zeros
177         ctx->b[ctx->c++] = 0;              /* ---- Spectre vulnerability
    ---- */
178
179     blake2b_compress(ctx, 1);              // final block flag = 1
180
181     // little endian convert and store
```

```
182     for (i = 0; i < ctx->outlen; i++) {
183         ((uint8_t *) out)[i] = (ctx->h[i >> 3] >> (8 * (i & 7))) & 0
       xFF;
184     }
185     return 0;
186 }
187
188 int blake2b(void *out, size_t outlen, const void *key, size_t keylen,
        const void *in, size_t inlen) {
189     blake2b_ctx ctx;
190
191     if (blake2b_init(&ctx, outlen, key, keylen))
192         return -1;
193
194     blake2b_update(&ctx, in, inlen);
195
196     blake2b_final(&ctx, out);
197     return 0;
198 }
199
200
201 int blake2b_long(void *pout, size_t outlen, const void *in, size_t
       inlen) {
202     uint8_t *out = (uint8_t *)pout;
203     blake2b_ctx blake_state;
204     uint8_t outlen_bytes[sizeof(uint32_t)] = {0};
205     int ret = -1;
206
207     if (outlen > UINT32_MAX) {
208         goto fail;
209     }
210
211     // Ensure little-endian byte order!
212     store32(outlen_bytes, (uint32_t)outlen);
213
214 #define TRY(statement)
                 \
215     do {
                 \
216         ret = statement;
                 \
217         if (ret < 0) {
                 \
218             goto fail;
                 \
219         }
                 \
220     } while ((void)0, 0)
221
222     if (outlen <= 64) {
223         TRY(blake2b_init(&blake_state, outlen, NULL,0));
```

```
224         TRY(blake2b_update(&blake_state, outlen_bytes, sizeof(
        outlen_bytes)));
225         TRY(blake2b_update(&blake_state, in, inlen));
226         TRY(blake2b_final(&blake_state, out));
227     } else {
228         uint32_t toproduce;
229         uint8_t out_buffer[64];
230         uint8_t in_buffer[64];
231         TRY(blake2b_init(&blake_state, 64, NULL, 0));
232         TRY(blake2b_update(&blake_state, outlen_bytes, sizeof(
        outlen_bytes)));
233         TRY(blake2b_update(&blake_state, in, inlen));
234         TRY(blake2b_final(&blake_state, out_buffer));
235         memcpy(out, out_buffer, 64 / 2);
236         out += 64 / 2;
237         toproduce = (uint32_t)outlen - 64 / 2;
238 
239         while (toproduce > 64) {
240             memcpy(in_buffer, out_buffer, 64);
241             TRY(blake2b(out_buffer, 64, NULL, 0, in_buffer, 64));
242             memcpy(out, out_buffer, 64 / 2);
243             out += 64 / 2;
244             toproduce -= 64 / 2;
245         }
246 
247         memcpy(in_buffer, out_buffer, 64);
248         TRY(blake2b(out_buffer, toproduce, NULL, 0, in_buffer, 64));
249         memcpy(out, out_buffer, toproduce);
250     }
251 
252 fail:
253     //clear_internal_memory(&blake_state, sizeof(blake_state));
254     return ret;
255 
256 #undef TRY
257 }
258 
259 
260 int main(int argc, char *argv[]){
261 
262     FILE *file = fopen("temp.txt", "r");
263     char in[1000];
264     char key[512];
265 
266     if (file == NULL) {
267         printf("No file!\n\n");
268         return 0;
269     }
270 
271     uint8_t* out = (uint8_t*) calloc(0x20, sizeof(uint8_t));
272 
```

```
273     fgets(in, 1000, file);
274     fgets(key, 512, file);
275
276     //char* in = "Certificação da Componente Criptográfica: Blake2b
        Hash Function";
277     //char* key = "PassCert";
278
279     blake2b(out, 0x20, key, strlen(key), in, strlen(in));
280
281     printf("\n\nC Implementation - Blake2b Hash Function(\"%s\") with
        [ \"%s\" ] as key:\n",in,key);
282
283     for(int i = 0; i < 0x20; i++){
284         printf("%02x", out[i]);
285     }
286
287     printf("\n\n\n");
288 }
```

A.2 Blake2b - Reference Code - Blade

This section presents the reference code written in C of the Blake2b library for the Blade tool.

```
 1 #include <stdint.h>
 2 #include <stddef.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <string.h>
 6
 7 // state context
 8 typedef struct {
 9     uint8_t b[128]; // input buffer
10     uint64_t h[8];  // chained state
11     uint64_t t[2];  // total number of bytes
12     size_t c;       // pointer for b[]
13     size_t outlen;  // digest size
14 } blake2b_ctx;
15
16 #ifndef ROTR64
17 #define ROTR64(x, y)  ( ( (x) >> (y) ) ^ ( (x) << (64 - (y)) ))
18 #endif
19
20 #define B2B_GET64(p)                            \
21 (((uint64_t) ((uint8_t *) (p))[0]) ^          \
22 (((uint64_t) ((uint8_t *) (p))[1]) << 8) ^   \
23 (((uint64_t) ((uint8_t *) (p))[2]) << 16) ^ \
24 (((uint64_t) ((uint8_t *) (p))[3]) << 24) ^ \
25 (((uint64_t) ((uint8_t *) (p))[4]) << 32) ^ \
26 (((uint64_t) ((uint8_t *) (p))[5]) << 40) ^ \
27 (((uint64_t) ((uint8_t *) (p))[6]) << 48) ^ \
```

```
28 (((uint64_t) ((uint8_t *) (p))[7]) << 56))
29
30
31 #define B2B_G(a, b, c, d, x, y) {   \
32 v[a] = v[a] + v[b] + x;         \
33 v[d] = ROTR64(v[d] ^ v[a], 32); \
34 v[c] = v[c] + v[d];             \
35 v[b] = ROTR64(v[b] ^ v[c], 24); \
36 v[a] = v[a] + v[b] + y;         \
37 v[d] = ROTR64(v[d] ^ v[a], 16); \
38 v[c] = v[c] + v[d];             \
39 v[b] = ROTR64(v[b] ^ v[c], 63); }
40
41
42 static const uint64_t blake2b_iv[8] = {
43     0x6A09E667F3BCC908, 0xBB67AE8584CAA73B,
44     0x3C6EF372FE94F82B, 0xA54FF53A5F1D36F1,
45     0x510E527FADE682D1, 0x9B05688C2B3E6C1F,
46     0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179
47 };
48
49
50 static void blake2b_compress(blake2b_ctx *ctx, int last)   {
51     //printf("Start Compress. \n");
52     const uint8_t sigma[12][16] = {
53         { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
54         { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 },
55         { 11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4 },
56         { 7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8 },
57         { 9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13 },
58         { 2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9 },
59         { 12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11 },
60         { 13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10 },
61         { 6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5 },
62         { 10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0 },
63         { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
64         { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 }
65     };
66
67     int i;
68     uint64_t v[16], m[16];
69
70     for (i = 0; i < 8; i++) {           // init work variables
71         v[i] = ctx->h[i];
72         v[i + 8] = blake2b_iv[i];
73     }
74
75     v[12] ^= ctx->t[0];                 // low 64 bits of offset
76     v[13] ^= ctx->t[1];                 // high 64 bits
77     if (last)                           // last block flag set ?
78         v[14] = ~v[14];
```

```
79
80      for (i = 0; i < 16; i++)            // get little-endian words
81          m[i] = B2B_GET64(&ctx->b[8 * i]);
82
83
84      for (i = 0; i < 12; i++) { // twelve rounds
85
86          B2B_G( 0, 4,  8, 12, m[sigma[i][0]], m[sigma[i][ 1]]);
87
88          B2B_G( 1, 5,  9, 13, m[sigma[i][2]], m[sigma[i][ 3]]);
89
90          B2B_G( 2, 6, 10, 14, m[sigma[i][4]], m[sigma[i][ 5]]);
91
92          B2B_G( 3, 7, 11, 15, m[sigma[i][6]], m[sigma[i][ 7]]);
93
94          B2B_G( 0, 5, 10, 15, m[sigma[i][8]], m[sigma[i][ 9]]);
95
96          B2B_G( 1, 6, 11, 12, m[sigma[i][10]], m[sigma[i][11]]);
97
98          B2B_G( 2, 7,  8, 13, m[sigma[i][12]], m[sigma[i][13]]);
99
100         B2B_G( 3, 4,  9, 14, m[sigma[i][14]], m[sigma[i][15]]);
101     }
102
103
104     for( i = 0; i < 8; ++i )
105         ctx->h[i] ^= v[i] ^ v[i + 8];
106
107 }
108
109
110 int blake2b_init(blake2b_ctx *ctx, size_t outlen, const void *key,
        size_t keylen) {
111     size_t i;
112     if (outlen == 0 || outlen > 64 || keylen > 64)
113         return -1;                      // illegal parameters
114
115     for (i = 0; i < 8; i++)             // state, "param block"
116         ctx->h[i] = blake2b_iv[i];
117
118     ctx->h[0] ^= 0x01010000 ^ (keylen << 8) ^ outlen;
119
120     ctx->t[0] = 0;                      // input count low word
121     ctx->t[1] = 0;                      // input count high word
122
123     ctx->c = 0;                         // pointer within buffer
124     ctx->outlen = outlen;
125
126     for (i = keylen; i < 128; i++)      // zero input block
127         ctx->b[i] = 0;
128
```

```
129     if (keylen > 0) {
130         blake2b_update(ctx, key, keylen);
131         ctx->c = 128;                       // at the end
132     }
133     return 0;
134 }
135
136 int blake2b_update(blake2b_ctx *ctx, const void *in, size_t inlen) {
137     size_t i;
138     for (i = 0; i < inlen; i++) {
139         if (ctx->c == 128) {                // buffer full ?
140             ctx->t[0] += ctx->c;            // add counters
141             if (ctx->t[0] < ctx->c)         // carry overflow ?
142                 ctx->t[1]++;                // high word
143             blake2b_compress(ctx, 0);       // compress (not last)
144             ctx->c = 0;                     // counter to zero
145         }
146         ctx->b[ctx->c++] = ((const uint8_t *) in)[i];    /* ----
    Spectre vulnerability ---- */
147     }
148     //printf("[update] c: %d\n",ctx->c);
149     return 0;
150 }
151
152 int blake2b_final(blake2b_ctx *ctx, void *out)   {
153     size_t i;
154
155     ctx->t[0] += ctx->c;                   // mark last block offset
156
157     if (ctx->t[0] < ctx->c)                 // carry overflow
158         ctx->t[1]++;                        // high word
159
160     while (ctx->c < 128)                    // fill up with zeros
161         ctx->b[ctx->c++] = 0;               /* ---- Spectre vulnerability
     ---- */
162
163     blake2b_compress(ctx, 1);               // final block flag = 1
164
165     // little endian convert and store
166     for (i = 0; i < ctx->outlen; i++) {
167         ((uint8_t *) out)[i] = (ctx->h[i >> 3] >> (8 * (i & 7))) & 0
    xFF;
168     }
169     return 0;
170 }
171
172 int blake2b(void *out, size_t outlen, const void *key, size_t keylen,
     const void *in, size_t inlen) {
173     blake2b_ctx ctx;
174
175     if (blake2b_init(&ctx, outlen, key, keylen))
```

```
176            return -1;
177
178        blake2b_update(&ctx, in, inlen);
179
180        blake2b_final(&ctx, out);
181        return 0;
182    }
183
184    int main(int argc, char *argv[]){
185
186        char in[1000];
187        char key[512];
188
189        uint8_t* out = (uint8_t*) calloc(0x20, sizeof(uint8_t));
190
191        //char* in = "Certificação da Componente Criptográfica: Blake2b
           Hash Function";
192        //char* key = "PassCert";
193
194        for (int i = 0; i < 1000; i++) {
195            in[i] = argv[1][i];
196        }
197
198        for (int i = 0; i < 512; i++) {
199            key[i] = argv[2][i];
200        }
201
202        blake2b(out, 0x20, key, strlen(key), in, strlen(in));
203    }
```

A.3 Blake2b - Jasmin Code

This section presents the reference code written in Jasmin of the blake2b library.

```
1  /*-----Updates Buffer-----*/
2  inline fn addFullBlock(reg u64 input) -> stack u64[16] {
3    reg u64 i temp;
4    stack u64[16] buffer;
5
6    i=0;
7
8    while( i < 16) {
9      temp = [input + i*8];
10     buffer[(int) i]= temp;
11     i += 1;
12   }
13   return buffer;  //Full tested and Correct
14 }
15
16 inline fn addLastBlock(reg u64 input inlen) -> stack u64[16] {
```

```
17    reg u64 i i8 counter temp mask;
18    reg u8 temp_u8;
19    stack u64[16] buffer;
20
21    i=0;
22    i8 = 8;
23    counter = 0;
24
25    while(i8 <= inlen){    //has some u64 left
26      temp = [input + 8*i];
27      buffer[(int)i] = temp;
28      i += 1;
29      counter += 8; //counts every u64 left
30      inlen -= 8;
31    }
32
33    input += counter;
34    mask = 0;
35
36    while(inlen > 0) { //some u8 left
37      inlen -=1 ;
38      mask <<= 8;
39
40      temp_u8 = (u8)[input + inlen]; //get u8
41
42      temp = (64u)temp_u8;
43
44      mask ^= temp;
45    }
46
47    buffer[(int)i] = mask; //put the last bytes in buffer
48    i += 1;
49
50    while(i < 16){ // No more bytes left, fill the buffer with 0's;
51      buffer[(int)i] = 0x0;
52      i += 1;
53    }
54
55    return buffer;
56  }
57  /*-----Updates Buffer-----*/
58
59
60  /*-----UPDATE V-----*/
61  inline fn array_v(stack u64[8] h, stack u64[2] t, inline int last) ->
        stack u64[16] {
62    stack u64[16] v;
63
64    reg u64 temp t0 t1;
65
66    inline int i;
```

```
67
68   for i = 0 to 8 {  // fill first 8 positions [0 .. 7] of array v
      with array h
69     temp = h[i];
70     v[i] = temp;
71   }
72
73   temp = 0x6a09e667f3bcc908; v[8] = temp;
74   temp = 0xbb67ae8584caa73b; v[9] = temp;
75   temp = 0x3c6ef372fe94f82b; v[10] = temp;
76   temp = 0xa54ff53a5f1d36f1; v[11] = temp;
77   temp = 0x510e527fade682d1; v[12] = temp;
78   temp = 0x9b05688c2b3e6c1f; v[13] = temp;
79   temp = 0x1f83d9abfb41bd6b; v[14] = temp;
80   temp = 0x5be0cd19137e2179; v[15] = temp;
81
82   t0 = t[0] ; v[12] ^= t0;
83   t1 = t[1] ; v[13] ^= t1;
84
85
86   if(last == 1) {
87     v[14] ^= 0xFFFFFFFFFFFFFFFF;
88   }
89
90   return v;
91 }
92 /*-----UPDATE V-----*/
93
94
95 /*-----BLAKE FUNCTIONS-----*/
96
97 /* Mix Function  */
98 inline fn doubleG(stack u64[16] v, inline int a0 b0 c0 d0, reg u64 x0
       y0, inline int a1 b1 c1 d1, reg u64 x1 y1) -> stack u64[16] {
99   reg u64 v_a, v_b, v_c, v_d;
100
101   v_b = v[b0] ; v[a0] += v_b ; v[a0] += x0;
102   v_b = v[b1] ; v[a1] += v_b ; v[a1] += x1;
103
104   v_a = v[a0] ; v_d = v[d0] ; v_d ^= v_a ; _,_,v_d = #ROR_64(v_d, 32)
      ; v[d0] = v_d;
105   v_a = v[a1] ; v_d = v[d1] ; v_d ^= v_a ; _,_,v_d = #ROR_64(v_d, 32)
      ; v[d1] = v_d;
106
107
108   v_d = v[d0] ; v[c0] += v_d;
109   v_d = v[d1] ; v[c1] += v_d;
110
111   v_b = v[b0] ; v_c = v[c0] ; v_b ^= v_c ; _,_,v_b = #ROR_64(v_b, 24)
      ; v[b0] = v_b;
```

```
112   v_b = v[b1] ; v_c = v[c1] ; v_b ^= v_c ; _,_,v_b = #ROR_64(v_b, 24)
       ; v[b1] = v_b;
113
114   v_b = v[b0] ; v[a0] += v_b ; v[a0] += y0;
115   v_b = v[b1] ; v[a1] += v_b ; v[a1] += y1;
116
117   v_a = v[a0] ; v_d = v[d0] ; v_d ^= v_a ; _,_,v_d = #ROR_64(v_d, 16)
       ; v[d0] = v_d;
118   v_a = v[a1] ; v_d = v[d1] ; v_d ^= v_a ; _,_,v_d = #ROR_64(v_d, 16)
       ; v[d1] = v_d;
119
120   v_d = v[d0] ; v[c0] += v_d;
121   v_d = v[d1] ; v[c1] += v_d;
122
123   v_b = v[b0] ; v_c = v[c0] ; v_b ^= v_c ; _,_,v_b = #ROR_64(v_b, 63)
       ; v[b0] = v_b;
124   v_b = v[b1] ; v_c = v[c1] ; v_b ^= v_c ; _,_,v_b = #ROR_64(v_b, 63)
       ; v[b1] = v_b;
125
126   return v;
127 }
128
129 /* Compressions */
130 inline fn compressMix(stack u64[16] buffer , stack u64[16]v ) ->
       stack u64[16] {
131   // i = 0
132   v = doubleG(v, 0, 4,  8, 12, buffer[0], buffer[1],
133                  1, 5,  9, 13, buffer[2], buffer[3]);
134
135   v = doubleG(v, 2, 6, 10, 14, buffer[4], buffer[5],
136              3, 7, 11, 15, buffer[6], buffer[7]);
137
138   v = doubleG(v, 0, 5, 10, 15, buffer[8], buffer[9],
139                1, 6, 11, 12, buffer[10], buffer[11]);
140
141   v = doubleG(v, 2, 7,  8, 13, buffer[12], buffer[13],
142                  3, 4,  9, 14, buffer[14], buffer[15]);
143
144
145   // i = 1
146   v = doubleG(v, 0, 4,  8, 12, buffer[14], buffer[10],
147              1, 5,  9, 13, buffer[4], buffer[8]);
148
149   v = doubleG(v, 2, 6, 10, 14, buffer[9], buffer[15],
150              3, 7, 11, 15, buffer[13], buffer[6]);
151
152   v = doubleG(v, 0, 5, 10, 15, buffer[1], buffer[12],
153              1, 6, 11, 12, buffer[0], buffer[2]);
154
155   v = doubleG(v, 2, 7,  8, 13, buffer[11], buffer[7],
156                3, 4,  9, 14, buffer[5], buffer[3]);
```

```
157
158
159   // i = 2
160   v = doubleG(v, 0, 4,  8, 12, buffer[11], buffer[8],
161                  1, 5,  9, 13, buffer[12], buffer[0]);
162
163   v = doubleG(v, 2, 6, 10, 14, buffer[5], buffer[2],
164                  3, 7, 11, 15, buffer[15], buffer[13]);
165
166   v = doubleG(v, 0, 5, 10, 15, buffer[10], buffer[14],
167                  1, 6, 11, 12, buffer[3], buffer[6]);
168
169   v = doubleG(v, 2, 7,  8, 13, buffer[7], buffer[1],
170                  3, 4,  9, 14, buffer[9], buffer[4]);
171
172
173   // i = 3
174   v = doubleG(v, 0, 4,  8, 12, buffer[7], buffer[9],
175                  1, 5,  9, 13, buffer[3], buffer[1]);
176
177   v = doubleG(v, 2, 6, 10, 14, buffer[13], buffer[12],
178                  3, 7, 11, 15, buffer[11], buffer[14]);
179
180   v = doubleG(v, 0, 5, 10, 15, buffer[2], buffer[6],
181                  1, 6, 11, 12, buffer[5], buffer[10]);
182
183   v = doubleG(v, 2, 7,  8, 13, buffer[4], buffer[0],
184                  3, 4,  9, 14, buffer[15], buffer[8]);
185
186
187
188   // i = 4
189   v = doubleG(v, 0, 4,  8, 12, buffer[9], buffer[0],
190                  1, 5,  9, 13, buffer[5], buffer[7]);
191
192   v = doubleG(v, 2, 6, 10, 14, buffer[2], buffer[4],
193                  3, 7, 11, 15, buffer[10], buffer[15]);
194
195   v = doubleG(v, 0, 5, 10, 15, buffer[14], buffer[1],
196                  1, 6, 11, 12, buffer[11], buffer[12]);
197
198   v = doubleG(v, 2, 7,  8, 13, buffer[6], buffer[8],
199                  3, 4,  9, 14, buffer[3], buffer[13]);
200
201
202   // i = 5
203   v = doubleG(v, 0, 4,  8, 12, buffer[2], buffer[12],
204                  1, 5,  9, 13, buffer[6], buffer[10]);
205
206   v = doubleG(v, 2, 6, 10, 14, buffer[0], buffer[11],
207                  3, 7, 11, 15, buffer[8], buffer[3]);
```

```
208
209     v = doubleG(v, 0, 5, 10, 15, buffer[4], buffer[13],
210                     1, 6, 11, 12, buffer[7], buffer[5]);
211
212     v = doubleG(v, 2, 7,  8, 13, buffer[15], buffer[14],
213                     3, 4,  9, 14, buffer[1], buffer[9]);
214
215
216
217     // i = 6
218     v = doubleG(v, 0, 4,  8, 12, buffer[12], buffer[5],
219                     1, 5,  9, 13, buffer[1], buffer[15]);
220
221     v = doubleG(v, 2, 6, 10, 14, buffer[14], buffer[13],
222                     3, 7, 11, 15, buffer[4], buffer[10]);
223
224     v = doubleG(v, 0, 5, 10, 15, buffer[0], buffer[7],
225                     1, 6, 11, 12, buffer[6], buffer[3]);
226
227     v = doubleG(v, 2, 7,  8, 13, buffer[9], buffer[2],
228                     3, 4,  9, 14, buffer[8], buffer[11]);
229
230
231
232     // i = 7
233     v = doubleG(v, 0, 4,  8, 12, buffer[13], buffer[11],
234                     1, 5,  9, 13, buffer[7], buffer[14]);
235
236     v = doubleG(v, 2, 6, 10, 14, buffer[12], buffer[1],
237                     3, 7, 11, 15, buffer[3], buffer[9]);
238
239     v = doubleG(v, 0, 5, 10, 15, buffer[5], buffer[0],
240                     1, 6, 11, 12, buffer[15], buffer[4]);
241
242     v = doubleG(v, 2, 7,  8, 13, buffer[8], buffer[6],
243                     3, 4,  9, 14, buffer[2], buffer[10]);
244
245
246     // i = 8
247     v = doubleG(v, 0, 4,  8, 12, buffer[6], buffer[15],
248                     1, 5,  9, 13, buffer[14], buffer[9]);
249
250     v = doubleG(v, 2, 6, 10, 14, buffer[11], buffer[3],
251                     3, 7, 11, 15, buffer[0], buffer[8]);
252
253     v = doubleG(v, 0, 5, 10, 15, buffer[12], buffer[2],
254                     1, 6, 11, 12, buffer[13], buffer[7]);
255
256     v = doubleG(v, 2, 7,  8, 13, buffer[1], buffer[4],
257                     3, 4,  9, 14, buffer[10], buffer[5]);
258
```

```
259
260
261   //i = 9
262   v = doubleG(v, 0, 4,  8, 12, buffer[10], buffer[2],
263                1, 5,  9, 13, buffer[8], buffer[4]);
264
265   v = doubleG(v, 2, 6, 10, 14, buffer[7], buffer[6],
266                3, 7, 11, 15, buffer[1], buffer[5]);
267
268   v = doubleG(v, 0, 5, 10, 15, buffer[15], buffer[11],
269                1, 6, 11, 12, buffer[9], buffer[14]);
270
271   v = doubleG(v, 2, 7,  8, 13, buffer[3], buffer[12],
272                3, 4,  9, 14, buffer[13], buffer[0]);
273
274
275   // i = 10
276   v = doubleG(v, 0, 4,  8, 12, buffer[0], buffer[1],
277                1, 5,  9, 13, buffer[2], buffer[3]);
278
279   v = doubleG(v, 2, 6, 10, 14, buffer[4], buffer[5],
280            3, 7, 11, 15, buffer[6], buffer[7]);
281
282   v = doubleG(v, 0, 5, 10, 15, buffer[8], buffer[9],
283              1, 6, 11, 12, buffer[10], buffer[11]);
284
285   v = doubleG(v, 2, 7,  8, 13, buffer[12], buffer[13],
286                3, 4,  9, 14, buffer[14], buffer[15]);
287
288
289   // i = 11
290   v = doubleG(v, 0, 4,  8, 12, buffer[14], buffer[10],
291            1, 5,  9, 13, buffer[4], buffer[8]);
292
293   v = doubleG(v, 2, 6, 10, 14, buffer[9], buffer[15],
294            3, 7, 11, 15, buffer[13], buffer[6]);
295
296   v = doubleG(v, 0, 5, 10, 15, buffer[1], buffer[12],
297            1, 6, 11, 12, buffer[0], buffer[2]);
298
299   v = doubleG(v, 2, 7,  8, 13, buffer[11], buffer[7],
300              3, 4,  9, 14, buffer[5], buffer[3]);
301
302   return v;
303 }
304
305 inline fn compression(stack u64[16] buffer, stack u64[8] h, stack u64
        [2] t, inline int last) -> stack u64[8] {
306   stack u64[16] v;
307   reg u64[8] rh;
308   reg u64 vi vi_plus8;
```

```
309
310   inline int i;
311
312   v = array_v(h,t,last);
313
314   v = compressMix(buffer,v);
315
316   for i = 0 to 8 {
317     vi = v[i]; vi_plus8 = v[i+8];
318
319     h[i] ^= vi; h[i] ^= vi_plus8;
320   }
321
322   return h;
323 }
324
325 /*  Initialize arrays ** h ** and ** t **   */
326 inline fn init(reg u64 outlen keylen key inlen) -> stack u64[8],
      stack u64[2] {
327   stack u64[16] buffer;
328   stack u64[8] h ;
329   stack u64[2] t ;
330   reg u64 h0_xor temp;
331
332   temp = 0x6a09e667f3bcc908; h[0] = temp;
333   temp = 0xbb67ae8584caa73b; h[1] = temp;
334   temp = 0x3c6ef372fe94f82b; h[2] = temp;
335   temp = 0xa54ff53a5f1d36f1; h[3] = temp;
336   temp = 0x510e527fade682d1; h[4] = temp;
337   temp = 0x9b05688c2b3e6c1f; h[5] = temp;
338   temp = 0x1f83d9abfb41bd6b; h[6] = temp;
339   temp = 0x5be0cd19137e2179; h[7] = temp;
340
341   h0_xor = keylen;
342   h0_xor <<= 8;
343   h0_xor ^= 0x01010000;
344   h0_xor ^= outlen;
345
346   h[0] ^= h0_xor;
347
348   if (keylen > 0 ){
349     t[0] = 128;
350     t[1] = 0;
351
352     if(keylen > 0) { //some remain bytes
353       buffer = addLastBlock(key, keylen);
354
355       if(inlen == 0){
356         h  = compression(buffer,h,t,1);
357       }
358       else{
```

```
359         h = compression(buffer,h,t,0);
360       }
361
362     }
363   }
364   else {
365     t[0] = 0; t[1] = 0;
366   }
367
368   return h, t;
369 }
370
371 inline fn update( reg u64 input inlen,  stack u64[8] h, stack u64[2]
     t) -> stack u64[8] {
372
373   stack u64[16] buffer;
374   reg u64 sum_t;
375   regx u64 inlen_s;
376
377   while (inlen >= 128) {
378
379     buffer = addFullBlock(input);
380
381     inlen -= 128;
382     input += 128;
383
384     t[0] += 128; // Add to t[0] the number of bytes read : 16 * 8
     bytes [ 8 bytes == 64 bits ] = 128;
385
386     if(inlen == 0) { //last block
387       h = compression(buffer,h,t,1);
388     }
389     else{ //not the last block
390       h = compression(buffer,h,t,0);
391     }
392   }
393
394   if(inlen > 0) { //some remain bytes
395     sum_t = inlen;
396     buffer = addLastBlock(input, inlen);
397     t[0] += sum_t; // Add to t[0] the remaining bytes;
398
399     h  = compression (buffer,h,t,1);
400   }
401
402   return h;
403 }
404
405 inline fn final(reg u64 out outlen, stack u64[8] h) {
406   reg u64 x y h_x i;
407
```

```
408   i = 0;
409
410   while(i < outlen) {
411     x = i;
412
413     x >>= 3;
414
415     y = i;
416     y &= 7;
417     y *= 8;
418
419     h_x = h[(int) x]; // <-
420
421     h_x >>= y;
422
423     h_x &= 0xFF;
424
425     [out + i] = h_x;
426
427     i+=1;
428   }
429 }
430
431 /*-----BLAKE FUNCTIONS-----*/
432 export fn blake2b(reg u64 in inlen, reg u64 out outlen , reg u64 key
       keylen) {
433
434   reg u64 outlenCpy;
435   stack u64[8] h;
436
437   stack u64[2] t;
438
439   outlenCpy = outlen;
440
441   h,t = init(outlenCpy, keylen, key, inlen);
442
443   h = update(in, inlen, h, t);
444
445   final(out, outlenCpy, h);
446 }
```

Appendix B
Tooling

This chapter presents all the files necessary to install or compile tools with different restrictions. It also shows a program that measures the performance of the resulting files from the Blade.

B.1 oo7 - Dockerfile

This section presents the dockerfile to install the oo7 tool in a docker container.

```
1  FROM ubuntu:18.04
2
3  RUN apt-get update && \
4        apt-get -y install sudo
5
6  RUN useradd -m docker && echo "docker:docker" | chpasswd && adduser
       docker sudo
7
8  USER docker
9  CMD /bin/bash
10
11 WORKDIR /home/docker
12
13 USER root
14 RUN apt-get update
15 RUN apt-get -y install build-essential
16 RUN apt-get -y install software-properties-common
17 RUN apt-get -y install zip
18 RUN apt-get -y install wget
19 RUN apt-get -y install git
20 RUN apt-get -y install cmake
21 RUN apt-get -y install clang
22 RUN apt-get -y install python
23 RUN apt-get -y install vim
24 RUN apt-get -y install cargo
25 RUN wget -qO /usr/local/bin/ninja.gz https://github.com/ninja-build/
       ninja/releases/latest/download/ninja-linux.zip
26 RUN gunzip /usr/local/bin/ninja.gz
27 RUN chmod a+x /usr/local/bin/ninja
28
29 USER docker
```

```
30 RUN git clone https://github.com/PLSysSec/blade.git
31 RUN git clone https://github.com/PLSysSec/lucet-blade.git -b blade
32 RUN git clone --recursive https://github.com/WebAssembly/wasi-sdk.git
33 RUN git clone --recursive https://github.com/WebAssembly/wabt.git -b
       1.0.15
34 RUN git clone https://github.com/WebAssembly/binaryen.git -b
       version_90
35 RUN git clone https://github.com/hacl-star/hacl-star.git
36
37 WORKDIR /home/docker/lucet-blade
38 RUN git submodule update --init --recursive
39
40 WORKDIR /home/docker/wasi-sdk
41 RUN wget https://github.com/WebAssembly/wasi-sdk/releases/download/
       wasi-sdk-12/wasi-sdk-12.0-linux.tar.gz
42 RUN tar xvf wasi-sdk-12.0-linux.tar.gz
43
44 WORKDIR /home/docker/wabt
45 RUN git submodule update --init
46 RUN make
47
48 WORKDIR /home/docker/binaryen
49 RUN git submodule init
50 RUN git submodule update
51 RUN cmake . && make
52
53 WORKDIR /home/docker/hacl-star
54 RUN git checkout de6a314ab
55
56 WORKDIR /home/docker
```

B.2 Blade - Dockerfile

This section presents the dockerfile to install the Blade tool in a docker container.

```
1 FROM ubuntu:18.04
2
3 RUN apt-get update && \
4       apt-get -y install sudo
5
6 RUN useradd -m docker && echo "docker:docker" | chpasswd && adduser
      docker sudo
7
8 USER docker
9 CMD /bin/bash
10
11 WORKDIR /home/docker
12
13 USER root
14 RUN apt-get update
```

```
15 RUN apt-get -y install build-essential
16 RUN apt-get -y install software-properties-common
17 RUN apt-get -y install zip
18 RUN apt-get -y install wget
19 RUN apt-get -y install git
20 RUN apt-get -y install cmake
21 RUN apt-get -y install clang
22 RUN apt-get -y install python
23 RUN apt-get -y install vim
24 RUN apt-get -y install cargo
25 RUN wget -qO /usr/local/bin/ninja.gz https://github.com/ninja-build/
       ninja/releases/latest/download/ninja-linux.zip
26 RUN gunzip /usr/local/bin/ninja.gz
27 RUN chmod a+x /usr/local/bin/ninja
28
29 USER docker
30 RUN git clone https://github.com/PLSysSec/blade.git
31 RUN git clone https://github.com/PLSysSec/lucet-blade.git -b blade
32 RUN git clone --recursive https://github.com/WebAssembly/wasi-sdk.git
33 RUN git clone --recursive https://github.com/WebAssembly/wabt.git -b
       1.0.15
34 RUN git clone https://github.com/WebAssembly/binaryen.git -b
       version_90
35 RUN git clone https://github.com/hacl-star/hacl-star.git
36
37 WORKDIR /home/docker/lucet-blade
38 RUN git submodule update --init --recursive
39
40 WORKDIR /home/docker/wasi-sdk
41 RUN wget https://github.com/WebAssembly/wasi-sdk/releases/download/
       wasi-sdk-12/wasi-sdk-12.0-linux.tar.gz
42 RUN tar xvf wasi-sdk-12.0-linux.tar.gz
43
44 WORKDIR /home/docker/wabt
45 RUN git submodule update --init
46 RUN make
47
48 WORKDIR /home/docker/binaryen
49 RUN git submodule init
50 RUN git submodule update
51 RUN cmake . && make
52
53 WORKDIR /home/docker/hacl-star
54 RUN git checkout de6a314ab
55
56 WORKDIR /home/docker
```

B.3 Blade - Makefile to add fence instructions

This section presents the Makefile to add all the lfences necessary to mitigate spectre vulnerabilities in a test C code by using the Blade tool.

```
1  LUCET_BLADE=$(HOME)/lucet-blade
2  WASI_SDK=$(HOME)/wasi-sdk
3  WABT=$(HOME)/wabt
4  BINARYEN=$(HOME)/binaryen
5  HACL_STAR=$(HOME)/hacl-star
6
7  LUCETC=$(LUCET_BLADE)/target/debug/lucetc
8  LUCETC_FLAGS=--emit=asm --guard-size "4GiB" --min-reserved-size "4GiB
      " --max-reserved-size "4GiB"
9  WASI_CLANG=$(WASI_SDK)/wasi-sdk-12.0/bin/clang
10 WASI_CLANG_FLAGS=-O3 --sysroot=$(WASI_SDK)/wasi-sdk-12.0/share/wasi-
      sysroot
11 WASI_LINK_FLAGS=-nostartfiles -Wl,--no-entry -Wl,--export-all
12 WAT2WASM=$(WABT)/out/clang/Debug/wat2wasm
13 WASM2WAT=$(WABT)/out/clang/Debug/wasm2wat
14 WASM_OPT=$(BINARYEN)/bin/wasm-opt
15 HACL_FLAGS=-I$(HACL_STAR)/dist/kremlin/include -I$(HACL_STAR)/dist/
      kremlin/kremlib/dist/minimal
16
17 .DEFAULT_GOAL=build
18
19 FORCE:
20 $(LUCETC): FORCE
21   cd $(LUCET_BLADE) && cargo build
22
23 wasm_src/%.wasm.unopt: c_code/%.c
24   $(WASI_CLANG) $(WASI_CLANG_FLAGS) $(HACL_FLAGS) $< -o $@ $(
      WASI_LINK_FLAGS)
25
26 wasm_src/%.wasm: wasm_src/%.wasm.unopt
27   $(WASM_OPT) -mvp --disable-mutable-globals -O4 $< -o $@
28
29 wasm_src/%.wasm: wasm_src/%.wat
30   $(WAT2WASM) $< -o $@
31
32 wasm_wat/%.wat: wasm_src/%.wasm
33   mkdir -p wasm_wat
34   $(WASM2WAT) $< -o $@
35
36 wasm_obj/%/ref.so: wasm_src/%.wasm $(LUCETC)
37   mkdir -p wasm_obj/$*
38   $(LUCETC) $(LUCETC_FLAGS) --blade-type=none $< -o $@
39
40 wasm_obj/%/lfence_with_v1_1.so: wasm_src/%.wasm $(LUCETC)
41   mkdir -p wasm_obj/$*
42   $(LUCETC) $(LUCETC_FLAGS) --blade-type=lfence --blade-v1-1 $< -o $@
```

```
43
44 wasm_obj/%/lfence_per_block_with_v1_1.so: wasm_src/%.wasm $(LUCETC)
45   mkdir -p wasm_obj/$*
46   $(LUCETC) $(LUCETC_FLAGS) --blade-type=lfence_per_block --blade-v1
      -1 $< -o $@
47
48 wasm_obj/%/slh_with_v1_1.so: wasm_src/%.wasm $(LUCETC)
49   mkdir -p wasm_obj/$*
50   $(LUCETC) $(LUCETC_FLAGS) --blade-type=slh --blade-v1-1 $< -o $@
51
52 all_spectre: spectre_unopt spectre_wasm spectre_so
53
54 spectre_unopt: \
55   wasm_src/test.wasm.unopt
56
57 spectre_wasm: \
58   wasm_src/test.wasm
59
60 spectre_so: \
61   wasm_obj/test/ref.so \
62   wasm_obj/test/lfence_with_v1_1.so \
63   wasm_obj/test/lfence_per_block_with_v1_1.so \
64   wasm_obj/test/slh_with_v1_1.so \
65
66 target/debug/spectre: all_spectre
67   cargo build
68
69 build: target/debug/spectre
```

B.4 Blade - Measure Performance - Reference Code

This section presents the main files of a Rust program that measures the performance of the Blake2b code with the added fence instructions and without.

B.4.1 Main

```
1 use blade_benchmarks::{blake2b, blade_setting::BladeType, BladeModule
     };
2
3 fn main() {
4     lucet_runtime::lucet_internal_ensure_linked();
5
6     let mut module = blake2b::Blake2bModule::new(BladeType::None,
     false);
7     let message = String::from("Cryptographic Component - Blake2b
     hash function").into_bytes();
8     let key = String::from("Blake2b Project").into_bytes();
9
```

```
10     let hash = module.blake2b(&key, &message);
11
12     println!("Blake2b hash of \"{}\" with key {} is {}", std::str::
    from_utf8(&message).unwrap(), std::str::from_utf8(&key).unwrap(),
    hex::encode(&hash));
13 }
```

B.4.2 Blake2b

```
1 use crate::blade_setting::BladeType;
2 use crate::module::{get_lucet_module, BladeModule};
3
4 use lucet_runtime::InstanceHandle;
5 use std::fmt;
6
7 const OUT_BYTES: usize = 32;
8
9 pub struct Blake2bModule {
10     so: InstanceHandle,
11 }
12
13 impl BladeModule for Blake2bModule {
14     fn new(blade_type: BladeType, blade_v1_1: bool) -> Self {
15         Self {
16             so: get_lucet_module("wasm_obj/blake", blade_type,
    blade_v1_1),
17         }
18     }
19 }
20
21 impl Blake2bModule {
22     /// Returns the encryption of `msg`. Result will have the same
    length as `msg`.
23     pub fn blake2b(&mut self, key: &[u8], msg: &[u8]) -> Vec<u8> {
24         // allocation
25         let mut heap_base = unsafe {
26             self.so.globals()[0].i_32 as u32 // seems like global 0
    is the heap base?
27         };
28         let key_ptr = heap_base;
29         heap_base += key.len() as u32;
30         let msg_ptr = heap_base;
31         heap_base += msg.len() as u32;
32         let out_ptr = heap_base;
33
34         // set up inputs
35         let heap = self.so.heap_mut();
36         let key_heap_idx = key_ptr as usize;
37         for i in 0 .. key.len() {
38             heap[key_heap_idx + i] = key[i];
39         }
40         let msg_heap_idx = msg_ptr as usize;
```

```
41          for i in 0 .. msg.len() {
42              heap[msg_heap_idx + i] = msg[i];
43          }
44
45          // call wasm
46          let _ = self.so.run("blake2b", &[
47              out_ptr.into(),
48              OUT_BYTES.into(),
49              key_ptr.into(),
50              key.len().into(),
51              msg_ptr.into(),
52              msg.len().into(),
53          ]).unwrap();
54
55          let mut output = vec![];
56          let heap = self.so.heap();
57          let out_heap_idx = out_ptr as usize;
58          for i in 0 .. OUT_BYTES {
59              output.push(heap[out_heap_idx + i]);
60          }
61          output
62      }
63  }
```