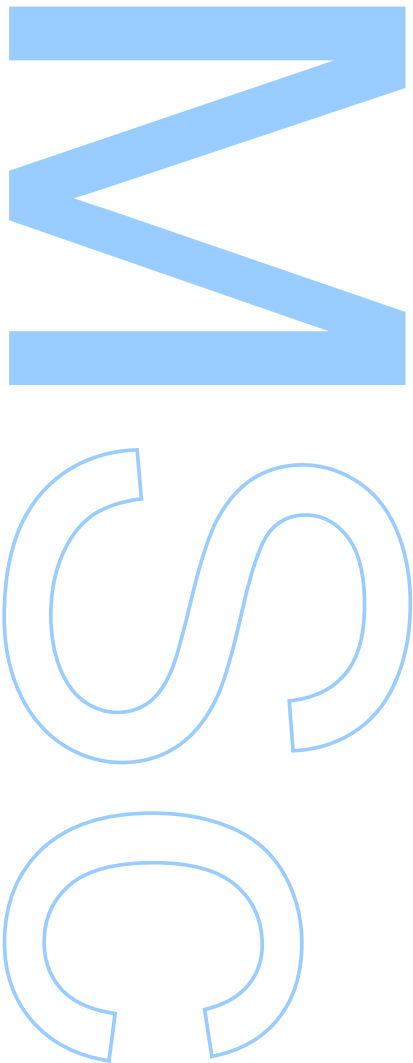
Speculative Execution Resilient Cryptography

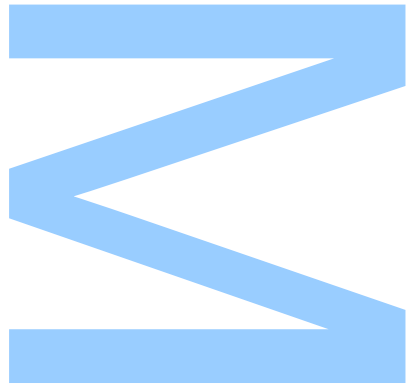
Rui Fernandes
Mestrado em Segurance Informática
Departamento de Ciência de Computadores
2023

Orientador
Bernardo Portela, Professor Auxiliar
Faculdade de Ciências da Universidade do Porto

Coorientador
José Bacelar Almeida
Departamento de Informática da Universidade do Minho




U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO



Declaração de Honra

Eu, Rui Pedro Gomes Fernandes, inscrito no Mestrado em Segurança Informática da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta dissertação, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Rui Fernandes

Porto, 29 de junho de 2023

Acknowledgments

I would like to thank my supervisors, Bernardo Portela, José Bacelar Almeida and Tiago Oliveira for their invaluable guidance and feedback throughout this year.

I would also like to express my gratitude to my friends and family for their constant motivation and encouragement.

Abstract

Modern processors make use of speculative execution, an optimization technique in which the CPU to begin executing instructions before their dependencies are resolved, thus allowing the execution of instruction ahead of time. However, the disclosure of the Spectre and Meltdown attacks in 2018 demonstrated that speculative execution can be exploited to access unauthorized data, leaking it to the cache.

Existing countermeasures are based on the insertion of fence instructions that prevent speculative execution. However, using these too often leads to a considerable performance penalty. An alternative approach called *Speculative Load Hardening*, addresses this issue by "poisoning" speculatively loaded values during a misspeculation, ensuring that they are not visible to an attacker.

In this context, this dissertation focuses on protecting `libjbn`, a Jasmin big number library, against Spectre v1 attacks using an information-flow type system. This type system allows Jasmin source code source code to be formally validated as resilient to a specific variant of speculative execution attacks. We then extend `libjbn` with a generic implementation of arithmetic operations over elliptic curves motivated by the fact that these are the foundation for the implementation of Elliptic Curve Cryptography protocols.

To evaluate the impact of these protections, we conduct a comprehensive benchmarking of the modified code. The results demonstrate that these protections introduce a relatively low overhead, ensuring protection against Spectre v1 attacks while achieving an acceptable level of performance.

Keywords: Speculative Execution, Jasmin, Elliptic Curve Cryptography

Resumo

Os processadores modernos fazem uso da execução especulativa, uma técnica de otimização na qual o CPU começa a executar as instruções antes de as respetivas dependências serem resolvidas, permitindo, assim, a execução destas instruções antecipadamente. No entanto, a divulgação dos ataques Spectre e Meltdown em 2018 demonstrou que a execução especulativa pode ser aproveitada para aceder a dados não autorizados, alterando, assim, o estado da cache.

As contramedidas existentes baseiam-se sobretudo na inserção de instruções que bloqueiam a execução especulativa. No entanto, o uso excessivo deste tipo de instruções resulta numa perda de *performance* considerável. Uma abordagem alternativa chamada *Speculative Load Hardening* consiste "mascarar" valores lidos especulativamente de memória, garantindo, assim, que estes não são visíveis para um atacante.

Neste contexto, esta dissertação tem como objetivo proteger a `libjbn`, uma biblioteca de números grandes escrita em Jasmin, contra ataques Spectre v1, usando, para isso, um *information-flow type system* que permite validar código Jasmin como resiliente a ataques de execução especulativa. Além disso, extendemos a `libjbn` com a implementação genérica de operações aritméticas sobre curvas elípticas motivados pelo facto de que estas são a base para a implementação de protocolos de Criptografia de Curvas Elípticas.

Para avaliar o impacto das proteções, realizamos uma comparação entre do código original e código protegido. Os resultados obtido demonstram que esta proteções apresentam um *overhead* mínimo, garantindo a proteção contra a variante 1 do Spectre, ao mesmo tempo que garantimos um nível de desempenho aceitável.

Palavras Chave: Execução Especulativa, Jasmin, Criptografia de Curvas Elípticas

Contents

List of Tables

List of Figures

Listings

List of Algorithms

Acronyms

ABI	Application Binary Interface
AES	Advanced Encryption Standard
AVX	Advanced Vector Extensions
AWS	Amazon Web Services
BL	Baseline
BTB	Branch Target Buffer
CL	Cache Line
CPU	Central Processing Unit
CT	Constant-Time
DPA	Differential Power Analysis
DSL	Domain-Specific Language
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
KEM	Key Encapsulation Mechanism
KPTI	Kernel Page-Table Isolation
MDS	Microarchitectural Data Sampling
ML	Machine Learning
NIKE	Non-Interactive Key Exchange
NIST	National Institute of Standards and Technology
PC	Program Counter
PHT	Pattern History Table
PQC	Post-Quantum Cryptography
ROB	Reorder Buffer
RSB	Return Stack Buffer

SCT	Speculative Constant-Time
selSLH	Selective Speculative Load Hardening
SIMD	Single Instruction Multiple Data
SLH	Speculative Load Hardening
SSBD	Speculative Store Bypass Disable
STL	Store-To-Load
STT	Speculative Taint Tracking
TLS	Transport Layer Security
TV	Time Variable
USLH	Ultimate Speculative Load Hardening
ZK	Zero-Knowledge

Chapter 1

Introduction

1.1 Motivation

Modern processors use speculative execution to improve performance by predicting and executing instructions ahead of time. Essentially, speculative execution is a performance optimization used by the CPU to predict and execute instructions ahead of time based on certain assumptions. As an example, in the case of conditional statements, instead of idling and waiting for the condition to be evaluated, the CPU makes an educated guess about which path is likely to be taken and begins executing instructions along that path. This improves performance by keeping the processor pipeline filled with instructions and avoiding waiting for the condition to be resolved. However, if the prediction turns out to be wrong, the CPU discards these instructions and resumes execution at the correct path.

While speculative execution may seem safe at first, in the sense that the result of a misprediction are never committed, speculative execution attacks such as Spectre [58] and Meltdown [64] have shown that this is not the case. Essentially, these attacks exploit speculative execution to execute instructions that would not be executed during sequential execution, potentially accessing secret information and leaking it to the cache or other observable side-channels.

One notable example of such an attack is Spook.js [4], a speculative execution attack affecting all Chromium-based browser that allows a potential attacker to read sensitive information such as passwords. Other speculative execution attacks have shown that it is possible to read kernel memory from the user-space [64], as well as extracting cryptographic keys from Intel SGX enclaves [88]. Quoting Hill et al. [49]:

> *Spectre variants have shown the need to move beyond just delivering performance [...] to more deeply consider security.*

Because this vulnerability is so prevalent, it is crucial to ensure that cryptographic code is

efficient and provably resistant to such attacks, which is the focus of this dissertation.

By inserting `LFENCE` instructions at specific points in the code, we can prevent the CPU from accessing sensitive data during speculative execution, ensuring that no secret information is leaked. However, the excessive use of these instructions results in a considerable performance penalty. For this reason, we consider an alternative and more efficient approach called *Speculative Load Hardening (SLH)* that hardens speculative loads, thus preventing secret data from being leaked.

1.2 Objectives & Contribution

In this dissertation, we focus on protecting cryptographic code against a type of speculative execution attack, namely Spectre v1. To do so, we consider an information-flow type system that allows Jasmin source code to be formally validated as resilient to the Spectre v1 speculative execution attack. In particular, we protect `libjbn` [34], a library for multi-precision arithmetic, against this attack.

`libjbn` is a library that provides a range of functions for both integer and finite field arithmetic that other Jasmin implementations can use. These functions allow developers to perform arithmetic operations on large numbers, such as addition, subtraction, multiplication, division and exponentiation. By providing these functions as a standalone library, other Jasmin implementations can easily incorporate them into their codebase rather than having to re-implement the same functionality. As a result, `libjbn` serves as a valuable resource when implementing cryptographic algorithms that rely on arithmetic with big numbers.

Examples of such libraries include Fiat Cryptography [38], which offers elliptic curve implementations, that were included in BoringSSL. In addition, HACL* is a verified cryptographic library used by Mozilla Firefox. In the context of verified cryptographic protocols, examples include miTLS [20], a formally verified implementation of the TLS protocol, and LibSignal [76], an implementation of the Signal protocol that relies on HACL*.

Furthermore, the significance of protecting this library increases substantially in the sense that it is already being used in cryptographic implementations – the Schnorr protocol [40] is implemented using integer arithmetic functions from `libjbn`, and Swoosh, a post-quantum lattice-based Non-Interactive Key Exchange (NIKE) scheme [43], uses the arithmetic functions over finite fields provided by `libjbn` to implement polynomial arithmetic in $\mathcal{R}_q$ in Rust.

Finally, we extend `libjbn` by incorporating arithmetic operations on prime-order elliptic curves, achieving a performance overhead of only 3% for addition, 2% for mixed addition, 4% for point doubling, and around 1% for scalar multiplication, when compared to a Jasmin implementation that is vulnerable to Spectre v1 attacks. The source code is available at https://github.com/ruipedro16/libjbn-sslh

1.3 Structure of the Dissertation

The rest of this dissertation is structured in the following chapters:

Chapter 2, Background: This chapter establishes the necessary background on speculative execution attacks, and discusses preliminary concepts including computer-aided cryptography and Elliptic Curve Cryptography (ECC).

Chapter 3, Jasmin: This chapter discusses the Jasmin programming language, which is a verification-friendly, low-level Domain-Specific Language (DSL) that allows to write efficient cryptographic implementations. This chapter also covers the Jasmin verification toolchain, which includes a series of tools that ensure the compiled assembly is memory safe and resilient to side-channel attacks.

Chapter 4, Implementation and Experimental Results: This chapter discusses how we can use an information-flow type system to protect a Jasmin library for multi-precision arithmetic. We then extend `libjbn` with arithmetic operations over prime-field elliptic curves motivated by the fact that it forms the foundation for the implementation of ECC protocols.

Chapter 5, Conclusion & Future Work: This chapter concludes the dissertation by summarizing the main findings of this work and presenting concise overview of the key conclusions. Finally, we discuss the limitations of the study and provide suggestions for areas of future research.

Chapter 2

Background

In this chapter, we introduce fundamental concepts that are necessary for understanding the remaining sections of the dissertation. §2.1 introduces side-channel attacks, and §2.2 summarizes the state-of-the-art on speculative execution attacks, evaluating their impact, and addressing some possible countermeasures, both hardware and software-based. §2.3 explores computer-aided cryptography, where formal methods are used to provide proof of the security and correctness of cryptographic implementations. Finally, §2.4 covers the key concepts of Elliptic Curve Cryptography (ECC), focusing on their application in the implementation of cryptographic primitives and protocols.

2.1 Side-Channel Attacks

Even though the algorithms underlying cryptosystems may be secure against cryptanalysis, their implementations may still be vulnerable to other kinds of attacks. Side-channel attacks are a class of attacks that exploit information leaked through the implementation itself rather than targeting the underlying cryptographic algorithms or protocols. These include, for example, timing attacks, that exploit variations in execution time, power analysis attacks, that exploit power consumption during program execution, and fault injection attacks, that manipulate hardware to compromise the integrity of the system. In addition, this type of attacks can be carried out remotely, making them particularly concerning. In this section, we will discuss two types of side-channel attacks: cache side-channel attacks as well as timing side-channel attacks.

Cache Side-Channel Attacks

Cache side-channel are a class of side-channel attacks that exploit the fact that accessing data in the cache is faster than accessing data from memory. The basic idea of this kind of attack is to bring the cache to a known state – for example, we can load from known memory addresses into the cache and we can flush it with the `clflush` instruction –, letting the victim's

process run and then measuring timing differences between cache and memory accesses in order to infer sensitive information. This is based on the fact that the behaviour of the cache may leak information about the operations performed by a process.

A Pribe+Probe attacks consists of two phases. First, the attacker fills the cache with attacker-controlled data. When the victim's process executes, some of this data will be evicted from the cache. Finally, the attacker measures the time it takes to access this data. If the primed data is evicted by the victim's process, this will result in a cache miss, providing the attacker with information about the victim's process' memory access patterns, allowing them to infer potentially sensitive information.

In Flush+Reload, first, the attacker repetitively flushes shared data with the victim, which can be achieved with the `clflush` instruction in `x86`. After the victim's process has executed, the attacker observes the time it takes to access this data – if there is a cache hit, it means that the victim's process accessed this data. Other variants of this attack include Evict+Reload, Flush+Flush.

Overall, cache side-channel attacks have long been used to target cryptographic implementations, successfully targeting AES [87]. In addition, in [96], the authors demonstrate that, using the Flush+Reload side-channel attack, it is possible to recover the secret key from OpenSSL ECDSA requests by just observing a relatively small number of executions.

Timing Side-Channel Attacks & Constant-Time Programming

Another type of side-channel attacks are timing attacks. Timing attacks take advantage of differences in execution time of a program in order to infer information about the data being processed. While other side-channel attacks, such as Differential Power Analysis (DPA), require physical access to the target machine, timing attacks, on the other hand, can be carried out remotely. Timing attacks on cryptosystems were first introduced by Kocher in 1996 [59].

One common approach to prevent timing attacks is Constant-Time (CT) programming, which states that the control flow and memory accesses should be independent of secret data. Likewise, secret data should not be used in operations whose execution time depends on the value of the operands – e.g. the `div` instruction in `x86`.

However, CT alone does not provide resistance against speculative execution attacks. Instead, we consider the notion of Speculative Constant-Time (SCT) that states that, for every possible choice of speculation, the program should not leak anything beyond what is leaked during sequential execution. A formal definition of SCT is provided in §3.3. In fact, as stated in [28]:

> *Even programs whose architectural (non-speculative) execution is carefully designed to not have any side-channel leaks could be vulnerable to transient execution attacks.*

Example: Memory Comparison

When sensitive information, such as cryptographic keys or authentication tokens, is stored in memory, it is important to ensure that the comparison operation does not leak any information about the values being compared. In other words, a standard comparison algorithm that returns as soon as the first different byte is encoured should not be used. In fact, standard comparison functions such as `memcmp` should not be used. From the `memcmp` man page:

> *Do not use memcmp() to compare security critical data, such as cryptographic secrets, because the required CPU time depends on the number of equal bytes. Instead, a function that performs comparisons in constant time is required. Some operating systems provide such a function (e.g., NetBSD's consttime_memequal()), but no such function is specified in POSIX. On Linux, it may be necessary to implement such a function oneself.*

One example of a CT comparison implementation is found in Go's `crypto/subtle` package, illustrated bellow:

```
// ConstantTimeByteEq returns 1 if x == y and 0 otherwise.
func ConstantTimeByteEq(x, y uint8) int {
    return int((uint32(x^y) - 1) >> 31)
}

// ConstantTimeCompare returns 1 if the two slices, x and y, have
// equal contents and 0 otherwise. The time taken is a function of
// the length of the slices and is independent of the contents. If
// the lengths of x and y do not match it returns 0 immediately.
func ConstantTimeCompare(x, y []byte) int {
    if len(x) != len(y) {
        return 0
    }

    var v byte

    for i := 0; i < len(x); i++ {
        v |= x[i] ^ y[i]
    }

    return ConstantTimeByteEq(v, 0)
}
```

Listing 2.1: Constant-time comparison in Go [2]

Here, the `ConstantTimeByteEq` function compares two unsigned 8-bit integers using bitwise operations, ensuring that the function is CT. The function `ConstantTimeCompare` compares two byte slices of the same size to determine if their contents are equal. To do so, an accumulator variable `v` is initialized to zero. The function then iterates through all of the bytes of each slice and performs a bitwise XOR between the corresponding bytes. If they are equal, the XOR of the two values is zero. This value is then ORed with `v` variable. If the contents of the byte slices being compared are equal, then `v` will remain zero. To check if this is the case, we call the function `ConstantTimeByteEq`, ensuring that the comparison is also done in CT.

In Listing 2.2, we present a standard comparison algorithm from Go's standard library that is not CT. The function enters a for loop that iterates through the bytes of the byte slices and, in each iteration, it compares the corresponding byte from both slices. If the current bytes being compared are not equal, the function returns early; otherwise, if the loop completes without any early returns, it means that the the contents of the byte slices are equal.

Here, it is important to note that, because this function returns as soon as the first different byte is encountered, it introduces variations in execution time. In other words, the total number of iterations of the for loop varies depending on the contents of the byte slices being compared. In this case, the number of iterations increases with the number of bytes that are equal at the beginning of the slices. These variations can be exploited by an attacker to potentially deduce information about the compared data through side-channel attacks. To avoid leaking secret information through side-channels, it is crucial that the control-flow and memory accesses do not depend on secret data.

```
func Compare(a, b []byte) int {
    l := len(a)
    if len(b) < l {
        l = len(b)
    }
    if l == 0 || &a[0] == &b[0] {
        goto samebytes
    }
    for i := 0; i < l; i++ {
        c1, c2 := a[i], b[i]
        if c1 < c2 {
            return -1
        }
        if c1 > c2 {
            return +1
        }
    }
samebytes:
    if len(a) < len(b) {
        return -1
    }
    if len(a) > len(b) {
        return +1
    }
    return 0
}
```

Listing 2.2: Non Constant-time comparison in Go [2]

Leakage Models

Leakage Models are theoretical models that allow the reasoning of information that is leaked through side-channels during program execution. These models aim to capture the various ways through which information may be leaked, which include timing variations during instruction execution, and memory accesses. Table 2.1 summarizes common leakage models.

Leakage Model	Description
Program Counter	Conditionals leak their guards
Baseline	PC + Memory Reads/Writes leak addresses
Cache line	PC + Memory accesses leak cache lines
Time-Variable	BL + TV arithmetic operators leak
TV + CL	TV arithmetic operators leak + CL

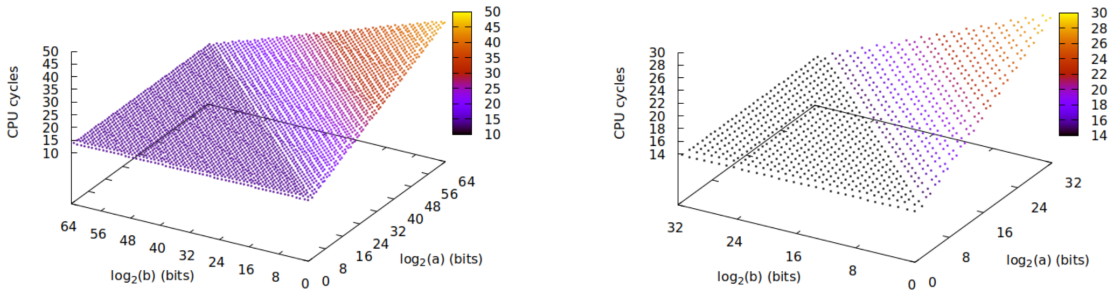
Table 2.1: Common leakage models [12]

- **Program Counter (PC) Leakage Model:** In this leakage model, we consider that conditional statements leak the value of the condition.
- **Baseline (BL) Leakage Model:** In the Baseline leakage model, we also assume that conditional statements leak their guards. In addition, we also consider that memory accesses – either reads or writes – leak the memory address that was accessed.
- **Cache Line (CL) Leakage Model:** In the Cache Line leakage model, we assume that conditional statements leak the value of the condition and, in addition, memory accesses leak the cache line if the address that was accessed. However, it is important to note that implementing defenses within the context of this specific model can be expensive. As such, it may make sense to adopt a less precise leakage model.
- **Time Variable (TV) Leakage Model:** While the BL leakage model is a useful starting point for constant-time programming, it is important to be aware that it does not capture certain details of program execution – time-variable instructions leak information about its inputs. In fact, in [54], the authors demonstrated that such timing variations can be exploited in order to recover cryptographic keys.

  As an example, the execution time of the `div` instruction on `x86` architectures depends on its operands. Figure 2.1 illustrates these timings variations for both 32 and 64-bit operands. Because of these timing variations, an attacker who is able to observe the execution time may be able to infer some information about the operands being processed. As a result, this instruction is not suitable for cryptographic implementations.

  Nevertheless, the BL leakage model does not take into account arithmetic instructions with variable latency. To address this, the Time Variable (TV) leakage model builds upon the

BL leakage model by considering that time-variable instruction leak a function of their inputs.



(a) 64-bit operands

(b) 32-bit operands

Figure 2.1: Timing behaviour of the `div` instruction on the AMD EPYC 7F52 `x86` CPU [12]

- **TV + CL Leakage Model:** The TV + CL combines both the TV and the CL leakageage models. In other words, we consider that conditional statements leak their guards, memory accesses leak the cache line of the addressed that was accessed, and instructions whose execution time depends on the operands leak a function of their inputs.

For the remainder of this dissertation, we will consider the BL leakage model.

2.2 Speculative Execution Attacks

Traditionally, CPUs execute instructions sequentially, following the order in which they appear in the program. However, modern computer architectures can achieve a higher throughput by leveraging instruction execution pipelining, which allows multiple instructions to be executed at the same time by using multiple execution units. A simple execution pipeline is illustrated in Figure 2.2. Each stage of the pipeline focuses on a specific task, such as instruction fetch, decode, execute, and write back.

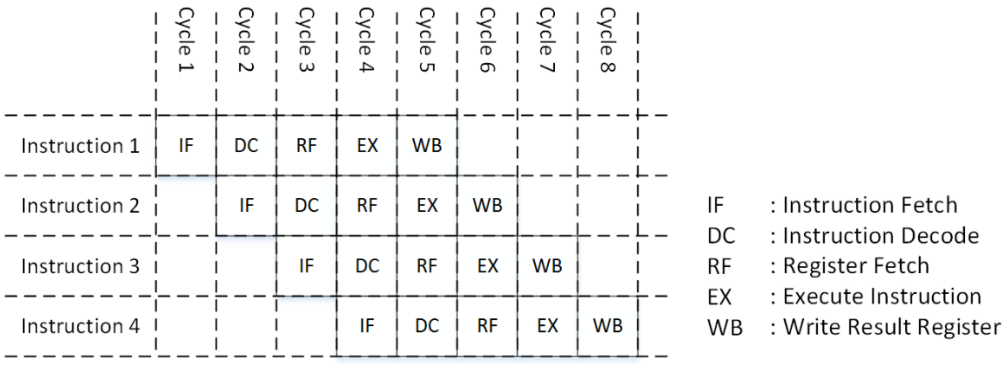


Figure 2.2: Pipelined instruction execution [66]

In addition, in out-of-order execution, the CPU may executes instructions in a different order than they were originally written in order to maximize instruction-level parallelism and resource utilization. This only occurs if there is no data dependencies between the instructions. Data dependencies between instructions occur when an instruction relies on the result of a previous instruction. In this case, the CPU must wait for this data to become available. Despite being executed out-of-order, the results of these instructions need to be committed to the architectural state. This means that instructions are completed in the order in which they were originally issued, even if they were executed out-of-order.

In addition to out-of-order execution, CPUs may also execute instructions *speculatively*. In this scenario, when encountering a branching instruction, the CPU tries to "guess" which path should be executed. Then, execution continues at the predicted branch ahead of time, i.e. even before the branch condition is resolved. These predictions are based on multiple prediction structures, which include the Branch Target Buffer (BTB) – used to predict the destination address of indirect jumps –, the Pattern History Table (PHT) – used to predict the result of conditional branches –, the Return Stack Buffer (RSB) – used to predict the destination address of return instructions, etc. Instead of waiting for the branch condition to be resolved, the CPU uses these prediction structures to continue execution speculatively, which, if the prediction is correct, reduces the latency caused by waiting for the branch instruction to be evaluated and the correct path to be determined.

It is important to note, however, that these predictions may turn out to be wrong. In that case, the executed instructions along the incorrect path are discarded from the execution pipeline. Furthermore, the CPU has to rollback the program state to the stage before the mispredicted path was taken. This requires updating values in register and in memory, which can be facilitated by the Reorder Buffer (ROB). Finally, once the misprediction is resolved, the CPU continues execution along the correct path. It is also worth noting that, due to mispredictions, the CPU incurs a performance penalty as a consequence of the wasted cycles spent executing the mispredicted path. Speculative execution attacks are a type of attack that take advantage of the CPU's ability to execute instructions out-of-order. While speculative execution can be an effective technique for improving performance, it can also introduce microarchitectural side-channels, through which information about the CPU's interal state can be leaked.

These attacks leverage this feature of modern CPUs to access data that otherwise should not be accessed, which is leaked through a side-channel channel – usually the cache. However, there are other possibilities – e.g. SMoTherSpectre [21] leverages port-contention, and NetSpectre [80] uses AVX-based side-channel.

Stages of a Speculative Execution Attack

Usually, a speculative execution attack consists of the following four phases.

1. **Setup Phase:** In this first phase, the goal of the attacker is to prepare the *exfiltration*

channel – the channel through which they will later be able to retireve the secret information. This can be, for example, the cache. In this scenario, it envolves priming the cache – i.e. filling it with attacker-controlled values. The key idea is to manipulate the cache to into a known state, then execute the victim program, and observe and infer the changes that occured in the cache after the execution.

In addition, the attacker mistrains the branch predictor, a component of the CPU that predicts whether a branch will be taken or not. This way, when a conditional branch is encountered, the prediction will be incorrect, leading the execution to continue along a mispredicted path.

In [58], Kocher et al. show that the branch predictor can be reliably trained to mispredict branches. This can be achieved by repeatedly executing victim code with carefully chosen inputs. This way, an attacker can mistrain the branch predictor to either always take or never take a particular branch. In the case of the variant 1 of Spectre, illustrated in Listing 2.3, this can be done by repeatedly calling the function with in-bound indices.

2. **Invocation of Victim Code:** In this phase of the attack, the victim code is executed with an input that triggers misspeculation. Here, it is worth mentioning that speculative execution attacks may be carried out remotely.

3. **Misspeculation:** In this phase of the attack, when encountering a branch instruction, the CPU tries to predict its outcome. However, because the branch predictor was previously mistrained, this prediction will be wrong and execution will continue along the mispredicted path. As a consequence of executing these instructions, both the architectural state – registers and memory – and the microarchitectural state – cache – will be changed. Eventually, when the CPU resolves the branch condition, the results of the speculatively executed instructions are discarded and the CPU resumes the execution along the correct path as if the branch was never taken. However, because the microarchitectural state is not restored, an attacker may be able to exploit these changes – e.g. through cache side-channel attacks – to potentially read secret data. This is explained in more detail in the "**Spectre v1 Bounds Check Bypass**" example.

4. **Exfiltration:** At this point, the attacker tries to retrieve secret information from the exfiltration channel. This can be done, for example, trough cache side-channel attacks. In [**?**], the authors use Flush+Reload or Evict+Reload.

2.2.1 Attack Taxonomy

In [24], the authors provide a comprehensive survey of speculative execution attacks and a taxonomy of speculative execution attacks based on the cause of the transient execution. This taxonomy, illustrated in Figure 2.3, considers two main classes of attacks based on the cause of speculative execution:

- **Spectre-type:** In this type of attack, an attacker takes advantage of CPU mispredictions to access data that would not otherwise be accessed. These attacks can be further categorized based on the different prediction structures they exploit – Spectre v1 exploits the PHT, Spectre v2 exploits the BTB, `ret2spec` and Spectre-RSB exploit the RSB, and Spectre v4 exploits Store-To-Load (STL) forwarding. These attacks are described in more detail below.
- **Meltdown-type:** During program execution, when an exception such as a page fault occurs, subsequent instructions should not be executed. However, due to speculative execution, these instructions may be executed even before the exception is handled. Eventually, these instructions will be discarded when the exception is handled. However, the microarchitectural state may be changed as a result of the execution, potentially leaking sensitive information which can be exploited via side-channel attacks.

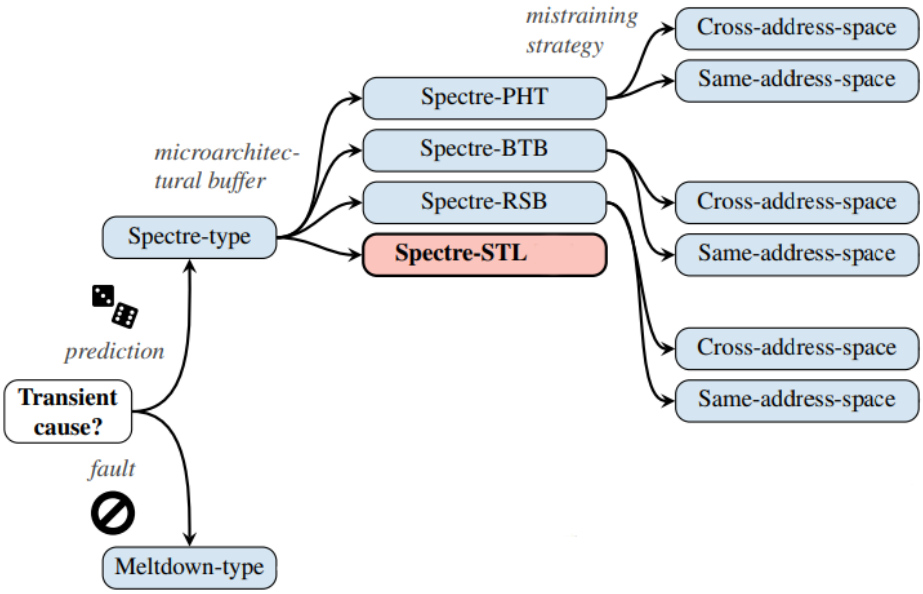


Figure 2.3: Taxonomy of speculative execution attacks. Adapted from [24]

We can also consider another class of attacks – **Microarchitectural Data Sampling (MDS)**. Unlike Spectre and Meltdown-like attacks, these type of attacks leaks data from internal CPU buffers instead of cache – this includes data that was never stored in the cache. Examples of this type of attack include RIDL [89], ZombieLoad [79], and Fallout [69].

For the remainder of this dissertation, we will focus only on Spectre-type attacks, more precisely, on the Spectre v1 variant. This stems from the fact that this variant is particularly challenging to mitigate completely. As will be discussed bellow, Spectre v2 can be mitigated through the use of `retpoline`, Kernel Page-Table Isolation (KPTI) mitigates Meltdown, and disabling Speculative Store Bypass Disable (SSBD) mitigates Spectre v4. Spectre v1, however, is more challenging.

Spectre v1 – Bounds Check Bypass

Spectre v1 [58] is a speculative execution attack that exploits conditional branch misprediction, allowing an attacker to read potentially secret data. Let us consider the Spectre gadget illustrated in Listing 2.3. In the setup phase, the attacker mistrains the branch predictor by repeatedly invoking this snippet of code with values of `x` that satisfy the condition `x < array1_size` After that, the CPU predicts that, in the next call, `x` will also satisfy this condition – this prediction is based on the PHT. However, if an attacker supplies a value of `x` outside of the bounds of the array, the CPU will (wrongly) predict that the conditional expression will evaluate to true and start executing the code inside the if statement. Note that, the statement `y = array2[array1[x] * 4096];` brings into cache data whose memory address depends on the value of `array1[x]`. Because the prediction turns out to be incorrect, the instructions that were speculatively executed are discarded. While at the architectural level it is as if these instructions were never executed, the state of the cache is not reverted. As such, an attacker can potentially retrieve sensitive information through a cache side-channel attack.

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

Listing 2.3: Conditional branch misprediction [58]

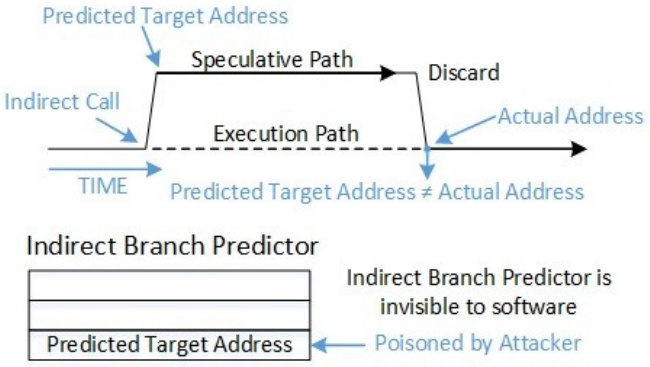
Spectre v2 – Branch Target Injection

Spectre v2, also reffered to as Spectre-BTB or "Branch Target Injection", is a speculative execution attack that exploits indirect branch misprediction to redirect the execution of the program to addresses that would not occur normally.

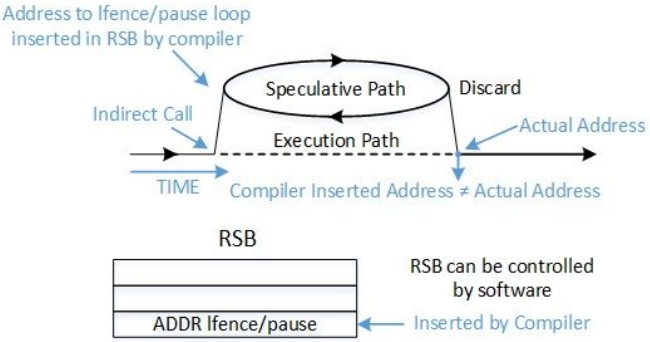
In this attack, an attacker first mistrains the branch predictor. Then, when encountering an indirect jump instruction, the CPU mispredicts the target address relying on the BTB. Because of the misprediction, execution continues speculatively at an attacker-chosen address, containing a Spectre gadget. During the speculative execution, the CPU may access memory addresses containing potentially secret information, which will be loaded to the cache. Eventually, the CPU discards the instructions that were speculatively executed, and execution continues at the correct address. However, the microarchitectural state is not reverted, and the attacker may be able to retrieve this secret through a cache side-channel attack.

One possible software mitigation to this attack is to replace all of the the indirect jumps with retpolines [44]. The motivation behind this approach is that by replacing indirect jumps with `ret` instructions, we avoid predictions based on the BTB – instead, we rely on the RSB, which can be controlled in the software. Following this approach, we push the address of the `pause; LFENCE` instruction onto the RSB. If the CPU speculates the target address of the indirect jump – using the RSB –, then we enter an infinite loop, which we eventually exit when

the CPU becomes aware of the misprediction. In this case, the execution continues along the correct path. This is illustrated in Figure 2.4.



(a) Before retpoline



(b) After retpoline

Figure 2.4: Retpoline [52]

Consider the indirect jump to the address stored in the `%rax` register illustrated in Listing 2.4. Here, after the `call load_label;` the address of the `pause; LFENCE` is pushed onto the stack, and onto the RSB. Next, when `load_label` starts executing, the `mov` overwrites the return address stored in the stack with the target of the indirect jump. At this point, the RSB and the return address on the stack differ. This means that, if the CPU is speculating – i.e. using the address from the RSB –, it will enter an infinite loop, which eventually is exited when the CPU realizes that the return address on the stack differs from the one in the RSB. At this point, the indirect jump is taken.

```
jmp *%rax
```

Listing 2.4: Indirect jump [52]

```
call load_label;
capture_spec:
    pause; LFENCE
    jmp capture_spec;
load_label:
    mov %rax, (%rsp);
    ret;
```

Listing 2.5: Compiled assembly for an indirect jump using retpoline [52]

Spectre v4 – Speculative Store Bypass

Spectre v4, also referred to as Spectre-STL exploits mispredictions stemming from the memory disambiguator to speculatively execute memory loads before all previous stores to that memory address have completed. In this scenario, the memory disambiguator mispredicts that a memory load does not depend on previous stores and, as a consequence, the load instruction reads values to the L1 cache, allowing attackers to read stale values through a microarchitectural covert channel. Eventually, when the addresses of the store instructions are resolved, these instructions will be re-executed.

To mitigate the Spectre v4 vulnerability, we can use Speculative Store Bypass Disable [53].

Return Address Misprediction

Spectre-RSB [60] and `ret2spec` [65] are Spectre variants that expoit the Return Stack Buffer (RSB), a microarchitectural buffer that stores the return address of functions. During speculative execution, when encountering a `ret` instruction, the CPU fetches the return address from the RSB. Essentiallty, by polluting the RSB, an attacker is able to divert execution and speculatively execute code that should not otherwise be executed. Eventually, when the return address is fetched from memory, these instructions will be discarded, but the microarchitectural state may be changed.

Meltdown – Rogue Data Cache Load

In [64], Lipp et al. demonstrated that, with this attack, it was possible to read kernel memory from a user-space process.

Essentially, kernel memory is isolated and privileged, meaning that processes running in user-space do not have direct access to it. However, Meltdown exploits speculative execution to bypass these privilege checks. When such process attempts to access kernel memory, the privileges checks must be performed, and eventually an exception is triggered. The exception is then handled by the operating system, preventing the process from reading from that memory. However, the illegal read may have already been speculatively executed, causing data from the

kernel-space to be leaked to the cache. Even though the exception is eventually handled, an attacker may be able to retrieve this data through the cache side-channel attack.

We can mitigate Metldown with Kernel Page-Table Isolation. Essentially, by creating two different sets of page tables, one for the kernel-space and one for the user-space, we prevent processes running on the user-space from accessing the kernel-space.

Other Attack Variants

In addition to the previously mentioned variants, encompasses a wide range of attacks. As an example, regarding trusted execution environments, Foreshadow [88] and SgxPectre [29] are speculative execution attacks that target Intel SGX enclaves. NetSpectre [80], is a remote speculative execution attack. Finally, ExSpectre [91] is an attack that exploits speculative execution to execute malware.

2.2.2 Attack Detection & Countermeasures

Several approaches have been proposed to detect speculative execution attacks. Regarding Machine Learning (ML) techniques, in [62], the approach consists in collecting traces of microarchitectural events – e.g. cache misses and branch mispredictions – using existing CPU performance counters and then using ML classifiers to analyze this data. In [75], the authors propose an approach based on Explainable ML, providing an explanation for the classification of the ML model. Both of these approaches are designed to detect both Spectre and Meltdown.

On the other hand, in [92], the authors apply taint analysis to track the flow of sensitive data within a program, allowing for the the detection of different variants of Spectre. SpecFuzz [73] is an automated tool to detect Spectre v1 attacks based on fuzzing. Finally, both Spectector [46], PitchFork [26], and BinsecRel [36] are static analysis tools based on symbolic execution that automatically detect speculative leaks. While Spectector and PitchFork are only capable of detecting Spectre v1 attacks, BinsecRel is able to detect both the Spectre v1 and Spectre v4 variants.

In [58], the authors suggest how Spectre attacks can be mitigated. The most straightforward solution is to disable speculative execution altogether. However, due to a significant performance penalty, this is not a viable solution. Another potential solution is preventing speculatively executed code from accessing to secret data. Finally, we can also limit data extraction from covert channels. For example, high-resolution timers facilitate cache-based side-channel attacks. As such, degrading the accuracy of such timers makes it more difficult for attackers to exploit timing differences.

In the remainder of this section, we discuss countermeasures to address Spectre attacks. This discussion covers both software and hardware-based mitigations. [94] provides a more in-depth analysis of Spectre mitigations, covering both hardware and software-based. Similarly [27]

provides an extense overview of software-based defenses for Spectre speculative execution attacks. As expected, these countermeasures introduce varying levels of performance degradation. In fact, not only do these countermeasures impact the performance, they also incur in a energy overhead of up to 72% [48].

Hardware Countermeasures

One possible approach to mitigate speculative execution attacks is to prevent data from being leaked to the covert channel. In this context, in [97], the authors propose a hardware protection mechanism called Speculative Taint Tracking (STT) that protects speculatively accessed data, by delaying the execution of instructions that read secret data until they become non-speculative, achieving an overhead of about 8.5%. Similarly, Conditional Speculation [63] speculatively executes instructions that are safe, but blocks those that may change the microarchitectural state until they become non-speculative. With this approach, the performance overhead of Spectre mitigations is about 13%. InvisiSpec [95] makes speculation invisible to the cache by speculatively loading data into a speculative buffer, without modifying the cache. When such load instructions are known to be safe, then this data is loaded into the cache. This approach incurs in a overhead of about 21%.

On another note, Context-Sensitive Fencing [85] dynamically inject fence instructions into the instruction stream, which prevents leaking data to the cache. With this approach, the authors we able to reduce the overhead of Spectre mitigations to about 8%.

Software Countermeasures

The most trivial solution is to insert fence instructions immediately after any conditional branch that depends on secret data, with the goal of preventing the CPU from executing continuing execution until the branch condition is resolved.

In this context, there are different fence instructions. `LFENCE` prevents the CPU from speculatively loading values from memory. On the other hand, `SFENCE` forces the CPU to wait untiull all previous store instructions are completed before continuing execution. Finally, `MFENCE` can be though of as a combination of `LFENCE` and `SFENCE` in the sense that the CPU is forced to wait until all stores and loads are completed.

Consider the example in Listing 2.6. The condition `index >= 0 && index < array_size` is used to check whether the index is in-bounds. However, the CPU may assume that the branch condition will evaluate to `true` and proceeds to execute the body of the if statement speculatively.

```
uint64_t get_secret(const uint64_t *array, size_t array_size,
                    size_t index) {
    if (index >= 0 && index < array_size) {
        return array[index];
    }

    return 0;
}
```

Listing 2.6: Spectre-v1 vulnerable array read

By doing so, we prevent the CPU from speculatively execute until the branch condition is resolved. As a result, we have the guarantee that no out-of-bounds memory accesses occurs. Eventually, the speculatively loaded value is discarded because the branch condition evaluates to `false`. Nevertheless, the microarchitectural state is changed and sensitive information may be leaked via side-channels.

```
uint64_t get_secret_(const uint64_t *array, size_t array_size,
                     size_t index) {
    if (index >= 0 && index < array_size) {
        __asm__ volatile ("lfence");
        return array[index];
    }

    return 0;
}
```

Listing 2.7: Array read protected using the `LFENCE` instruction

Here, it is worth noting that, while inserting fence instructions on all conditional jumps is a rather straightforward mitigation for speculative execution attacks, the performance impact is too severe. For example, inserting `LFENCE` instructions on all conditional jumps of the main loop of an implementation of SHA-256 resulted in a performance impact of around 60% [57].

Speculative Load Hardening

Speculative Load Hardening (SLH) [25], a countermeasure that addresses Spectre v1, works by "poisoning" the speculatively loaded values value. The idea is to mantain a predicate indicating whether the execution is misspeculating or not. If it is, this value is then used to "poison" both values and addresses of load instructions. As a consequence, we prevent speculatively executed code from accessing secret information. Here, it is important to note that this approach does not prevent any side-channel, we only prevent an attacker from being able to observe secret data.

However, SLH is still too restrictive in the sense that it considers that all memory accesses must be hardened. Selective Speculative Load Hardening (selSLH) [81] improves SLH by considering that not all memory accesses need to be hardened – we only need to protect speculative loads to public variables.

Other variants of SLH can be considered. In [98], the authors propose Ultimate Speculative

Load Hardening (USLH), a variant of SLH that also masks inputs of variable-latency instructions – e.g. the `div` instruction on `x86` architectures.

Similarly, Blade [90] provides an automatic approach to provably eliminate speculative leaks from code. The underlying idea is that, in order to eliminate speculative leaks, we do not need to stop speculation altogether. Instead, we only need to stop speculation from expressions that speculatively access sensitive information – *sources* – to expressions that leaks these secrets to the cache – *sinks*. As such, to protect programs, it is necessary to cut the information flow from sources to sinks. To achieve this, the authors propose a `protect` primitive similar to SLH that stops such speculative data-flows only for a given variable.

2.3 Computer Aided Cryptography

Correctly implementing cryptographic software is a difficult task: ideally, cryptographic implementations should achieve the following properties, referred to as *Big Four* in [18]:

- **Memory Safety:** Memory-related errors are common and can be a significant issue in software development [84]. A recent survey [23] has shown that about 37% of vulnerabilities found on cryptographic libraries are memory-related errors, while only 27% are cryptographic issues. Memory safety ensures that programs do not access memory that they are not supposed to, such as memory that has already been freed or memory that belongs to other processes. For example, recently, a buffer overflow vulnerability was found in multiple implementations of SHA-3 that allowed an attacker to find second preimages and preimages [71]. Essentially, it was possible to XOR attacker-controlled values into memory, thus rendering typical defenses against buffer overflows like stack canaries ineffective.
- **Functional correctness with respect to a standard specification:** Functional correctness of programs is usually checked via unit testing or fuzzing. In cryptography, however, these approaches are not suitable in the sense that a higher level of assurance is required. Instead, cryptographic algorithms are often specified in a formal language, and the correctness with respect to a specification can be proved using formal methods and program verification.
- **Provable Security:** Provable Security refers to the fact that cryptographic schemes must provide a rigorous proof that they achieve certain properties. This usually consists in formalizing the security goals, defining a precise adversarial model that encompasses the capabilities of the attacker, and proving that the scheme satisfies the goal. In this context, *proofs by reduction* are often considered. This type of proof provides a way of relating the success of an adversary against a certain protocol to their success of attacking a problem that is believed to be hard – e.g. the ECDLP.
- **Resistance against timing side-channel attacks:** To avoid leaking sensitive information through side-channels, cryptographic implementations should be resistant to side-channel

attacks. In fact, as stated in the NIST evaluation criteria for Post-Quantum Cryptography (PQC) standardization effort [72]:

> *Schemes that can be made resistant to side-channel attack at minimal cost are more desirable than those whose performance is severely hampered by any attempt to resist side-channel attacks. We further note that optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not.*

In addition, cryptographic implementations must be *efficient*. For example, in secure communication protocols like Transport Layer Security (TLS) or IPSec, data packets are encrypted and signed to ensure confidentiality, integrity, and authentication. As such, cryptographic implementations must be efficient as to prevent any significant performance degradation. Furthermore, it is important to note that cryptographic software runs on a multitude of devices, ranging from embedded systems, which are often resource-constrained, to high-performance servers.

2.3.1 Type Systems

Type systems are an essential component of programming languages, providing the mechanisms to ensure type safety. Type safety guarantees that operations are performed only on data of the appropriate type, which prevents errors that can arise from mismatched data types during program execution. As such, by catching potential errors that may violate type safety at compile time, type systems help prevent of runtime errors.

On the other hand, information flow type systems focus on the flow of information within a program execution. In information flow type systems, labels or annotations are associated with data values to indicate their security classifications. These labels enable the type system to track the flow of information and ensure that data with higher sensitivity levels is not inadvertently mixed with less sensitive data. For example, if a program's branching condition depends on secret data, an information flow type system may reject the program to prevent potential leaks of sensitive information through conditional branches.

Overall, by identifying type-related errors at compile time, type systems help us prevent errors that arise from handling data of incompatible types, which can potentially cause runtime errors.

2.3.2 Proof Assistants Machine-Checked Proofs

Traditionally, the security proofs of cryptographic schemes have been proven by hand. However, and considering the increasing complexity of cryptographic schemes, writing cryptographic proofs is often complex and error-prone [56]. As a response to these challenges, computer-aided cryptography [14] is a line of research that leverage formal methods and program verification

techniques to ensure that software is correct and safe, which is achieved by proving properties such as functional correctness, memory-safety and resistance against side-channel attacks.

In this context, proof assistants play an important role. There are several proof assistants available such as Coq, Agda, and Lean. Essentially, proof assistants guarantee that in a proof, each step is a logical consequence of previous ones. In other words, we have the guarantee that logical rules and axioms are properly applied, therefore improving the reliability of proofs.

Some proof assistants are specifically oriented towards security proofs of cryptographic primitives and protocols, such as EasyCrypt [16] and ProVerif [22]. In fact, EasyCrypt has been used in a variety of machine-checked proofs. Some examples include the the proof of security of Amazon Web Services (AWS) key management service [6], for proving ballot privacy of the Belenius voting protocol [32], in the formalization of the security properties of Zero-Knowledge (ZK) protocols [39] and in the context of Post-Quantum Cryptography [15, 50].

2.3.3 Verified Compilation

During compilation, compilers may perform aggressive optimizations in an attempt to improve the performance. While such compiler optimizations may increase the performance of programs, not all of them preserve the security properties.

The following example, presented in [51], illustrates how compilers may introduce branching instructions while optimizing on code that is otherwise CT. Listing 2.8 illustrates a simple function that takes two integers as arguments, `b` and `x`, and returns the result of the double negation of `b` multiplied by `x`. Essentially, if the function is called with a non-zero value for `b`, it will return the value of `x`, and will return zero otherwise. Because of the double negation, the compiler may interpret the integer `b` as boolean value, thus converting any non-zero value to `true` and zero to `false`.

```
int f(int b, int x) {
    return (!!b) * x;
}
```

Listing 2.8: Integer multiplication

```
int f(int b, int x) {
    if (b) {
        return x;
    } else {
        return 0;
    }
}
```

Listing 2.9: Integer multiplication with optimizations

Compiling the function illustrated in Listing 2.8 with the `clang` compiler and using the `-m32` and `-O1` flags, we get the following assembly:

```
f:
   movl  4(%esp), %eax
   testl %eax, %eax
   je .LBB0_2
   movl  8(%esp), %eax
.LBB0_2:
   retl
```

Listing 2.10: Assembly demonstrating that compiler breaks CT

In [83], the authors highlight several instances in which compiler optimizations may break properties of cryptographic code. However, for critical software such as cryptographic implementations, these effects cannot be ignored. Verified compilation is a line of research that focuses on developing compilers that are accompanied by formal proofs of correctness. Informally, this means that the compiler preserves the behaviour and security properties of the source program during compilation. In other words, we have the guarantee that the compiled program has the same behaviour as the source program, and the same security properties.

For example, CompCert [61] is a fomally verified compiler, both programmed and proven in the Coq proof assistant, with a level of performance similar to that of `gcc` with no optimization flags [1], supporting the `x86`, ARM and RISC-V architectures. In addition, in [9], the authors extend the CompCert compiler by adding support support vectorized SIMD instructions in the `x86` architecture.

In addition to preserving the behaviour of programs, compilers may also be proven to preserve other properties of the source program. For example, in the context of CT, CompCert-CT [17], a modified version of CompCert, is proven to preserve the CT of the source code.

2.4 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is based on the algebraic structure of elliptic curves over finite fields. The use of elliptic curves in Cryptography was proposed both by the mathematicians Koblitz [55] and Miller [68]. In this section, we establish the necessary background on ECC. This section is based on [47, 93].

ECC offers advantages over other cryptosystems, such as RSA, mainly due to the fact that shorter key sizes are required to achieve similar security levels, as illustrated in Table 2.2. Here, a security level of n bits means that the best known algorithm for breaking the cryptosystem takes at least 2^n steps.

Security Level (in bits)	**80**	**112**	**128**	**192**	**256**
RSA (key length in bits)	1024	2048	3072	8192	15360
ECC (key length in bits)	160	225	256	384	512

Table 2.2: Key lengths of ECC and RSA comparison for different security levels [47].

Definition

An elliptic curve defined a finite field K is a non-singular curve which can be written in the *Weierstrass form*

$$E\colon y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

with $a_1, \ldots, a_6 \in K$. If $char(K) \neq 2, 3$, then this equation can be simplified to the *short Weierstrass form*

$$E\colon y^2 = x^3 + ax + b$$

by applying a change of variables

For cryptographic applications, elliptic curves are usually defined either over prime fields $\mathbb{F}_p$, where p is a large prime, or over binary fields $\mathbb{F}_{2^m}$. Here, we also require that the discriminant $\Delta = 4a^3 + 27b^2 \neq 0$. Such curves are said to be *singular* and should not be used for cryptographic purposes.

Point Representation

The representation of points over curves on elliptic curves is a crucial aspect to take into consideration when implementing cryptographic algorithms. Different representations offer distinct advantages. Two common approaches to represent points are affine coordinates, which offer a straightforward and intuitive way of representing points, and projective coordinates, that allows for more efficient computations.

Using affine coordinates, we represent a point over an elliptic curve by the corresponding x and y coordinates. The set of all *affine points*, denoted by $\mathbb{A}(K)$, is given by

$$\mathbb{A}(K) = \{(x, y) \mid x, y \in K\}$$

On the other hand, using affine coordinates, the formulas for point addition require a series of multiplications, additions and a field inversion. However, field inversions are expensive operations. Projective coordinates have the advantage of requiring less field inversions. Thus, points on elliptic

curves are usually represented using projective coordinates. To represent a point $P \in E(\mathbb{F}_p)$ whose affine representation is (x, y) using coordinates, we introduce a new coordinate $Z \neq 0$ such that $x = X \cdot Z^{-1}$ and $y = Y \cdot Z^{-1}$, and represent the point as $(X : Y : Z)$. In the finite field $\mathbb{F}_p$, there are $p - 1$ values for Z that satisfy these restriction, which means that, using projective coordinates, we can represent the same point in $p - 1$ different ways. For example, the point $(2, 4)$ can be represented as $(2 : 4 : 1)$ as well as $(8 : 16 : 4)$.

Here, we are interested in the set of all projective points with $Z \neq 0$, denoted by $\mathbb{P}(K)^*$, which is given by

$$\mathbb{P}(K)^* = \{(X : Y : Z) \mid X, Y, Z \in K, Z \neq 0\}$$

The set of projective points with $Z = 0$, denoted by $\mathbb{P}(K)^0$, and given by

$$\mathbb{P}(K)^0 = \{(X : Y : Z) \mid X, Y, Z \in K, Z = 0\}$$

is called *line at infinity*. These points do not correspond to any affine point.

There is a 1-to-1 correspondence between $\mathbb{P}(K)^*$ and $\mathbb{A}(K)$. If $Z \neq 0$, the projective point $(X : Y : Z)$ corresponds to the affine point $(X/Z, Y/Z)$. In addition, the point at infinity $\mathcal{O}$ is given by the projective point $(0 : 1 : 0)$, and the negative of the projective point $(X : Y : Z)$ is the point $(X : -Y : Z)$.

The projective form of the Weierstrass equation for elliptic curves E over the field K can be obtained with the following change of variables:

$$x \mapsto \frac{X}{Z} \qquad y \mapsto \frac{Y}{Z}$$

In this case, the Weierstrass equation becomes

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

and the short Weierstrass equation becomes

$$Y^2Z = X^3 + aXZ^2 + bZ^3$$

In addition to affine and projective coordinate representations, there are alternative coordinate systems that can be considered. Two such examples are Jacobian coordinates and Chudnovsky coordinates. Jacobian coordinates map a point in Jacobian form, represented as $(X : Y : Z)$, to its corresponding affine point, given by $(X/Z^2, Y/Z^3)$. Using Chudnovsky coordinates, on the other hand, a point is represented as $(X : Y : Z : Z^2 : Z^3)$. In this context, it is important to

note that the Chudnovsky coordinate system is the inherently redundant, which allows for the efficient computation of scalar multiplications.

Points on the curve

An elliptic curve E over a finite field $\mathbb{F}_p$ is defined by the set of points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ that satisfy the equation of the curve. In addition, we consider an additional point denoted as $\mathcal{O}$, called the *point at infinity*.

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

The number of points in the elliptic curve, $\#E(\mathbb{F}_p)$, can be computed efficiently with Schoof's algorithm.

Hasse's theorem defines the upper and lowers bound of the number of points of an elliptic curve E defined over a finite field $\mathbb{F}_p$:

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbb{F}_p) \leq p + 1 + 2\sqrt{p},$$

which can also be stated as

$$|\#E(\mathbb{F}_p) - (p + 1)| \leq 2\sqrt{p}$$

This theorem states that the number of points on the curve is close to the size of the field p, with an error term bounded by $2\sqrt{p}$. This property is important in the security analysis of elliptic curves as it provides an upper bound on the order of the group of points on the curve.

Group Law

The points on an elliptic curve E defined over the finite field $\mathbb{F}_p$ form an abelian group under point addition, meaning that they satisfy the following properties:

Associativity:

$$(P_1 + P_2) \oplus P_3 = P_1 \oplus (P_2 \oplus P_3),\ \forall\ P_1, P_2, P_3 \in E(\mathbb{F}_p)$$

Existence of Identity: Here, the point at infinity, $\mathcal{O}$, serves as the identity element.

$$P \oplus \mathcal{O} = \mathcal{O} \oplus P = P,\ \forall\ P \in E(\mathbb{F}_p)$$

Existence of Inverse:

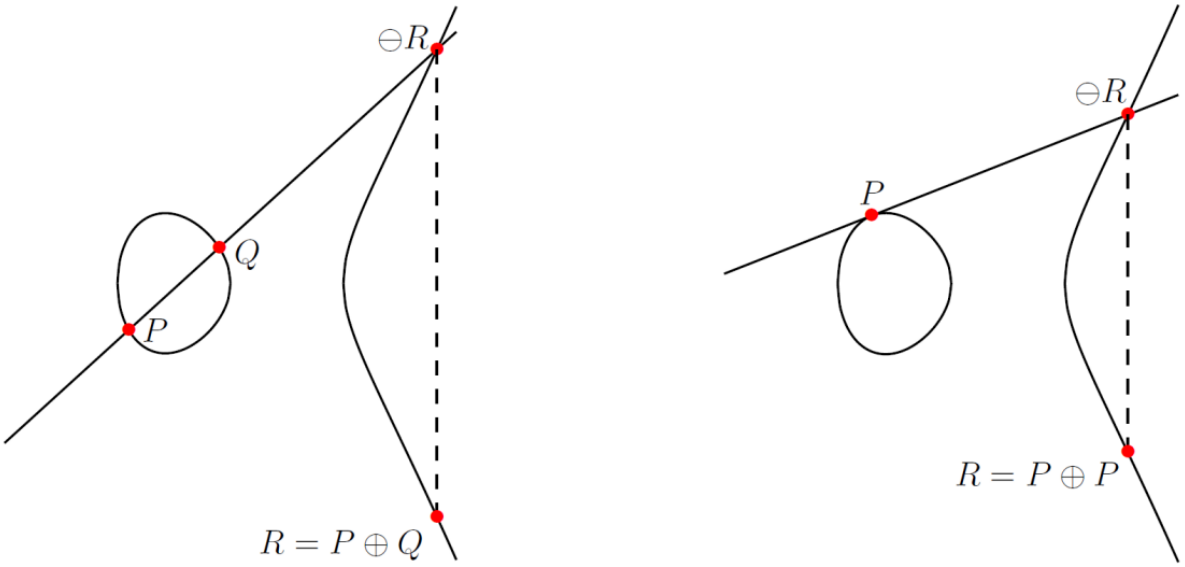
$$P \oplus (-P) = (-P) \oplus P = \mathcal{O},\ \forall\ P \in E(\mathbb{F}_p)$$

Commutativity:

$$P_1 \oplus P_2 = P_2 \oplus P_1, \ \forall \ P_1, P_2 \in E(\mathbb{F}_p)$$

Geometric Interpretation of the Group Law

The group law on elliptic curves provides a way to combine two points on the curve to get a third point, also on the curve. Due to its visual interpretation, the group law is often referred to as the *chord-and-tangent rule*. As illustrated in Figure 2.5, any three points lying on the same line add to $\mathcal{O}$. Given two points $P, Q \in E(\mathbb{F}_p)$, we compute the $R = P \oplus Q$ by drawing a line through these two points, that intersects the curve at another point, $-R$, which is then reflected across the x-axis to get R. Similarly, given $P \in E(\mathbb{F}_p)$, we get $-P$ by reflecting P across the x-axis. Finally, to double a point $P \in E(\mathbb{F}_p)$, we draw a tangent line to the elliptic curve at the point P, which intersect the curve at another point, $-2 \cdot P$, which we then reflect across the x-axis to get $2 \cdot P$.



(a) Point addition (b) Point doubling

Figure 2.5: Geometric interpretation of the group law [33]

Algebraic Addition

Considering two points $P_1 = (x_1, y_1), P_2 = (x_2, y_2) \in E(\mathbb{F}_p)$, the algebraic addition formulas distinguish between the following cases:

- If $P_1 = \mathcal{O}$, then $P_1 \oplus P_2 = P_2$.
- If $P_2 = \mathcal{O}, P_1 \oplus P_2 = P_1$.
- If $P_2 = -P_1$, i.e. if $P_2 = (x_1, -y_1)$, then $P_1 \oplus P_2 = \mathcal{O}$.

- If $P_1 = P_2$, then $P_1 \oplus P_2 = 2 \cdot P_1 = (x_3, y_3)$, where:

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1, \qquad \text{where } \lambda = \frac{3x_1^2 + a}{2y_1}$$

- Otherwise, $P_1 \oplus P_2 = P_3 = (x_3, y_3)$, where:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1, \qquad \text{where } \lambda = \frac{y_1 - y_2}{x_1 - x_2}$$

Here, it is worth noting that these addition formulas are not *complete*. In other words, the formula used for computing the addition depends on the points being added. For example, when adding two points P_1 and P_2 such that $P_1 = P_2$, if we consider the last formula, we incur in a division by zero – we compute λ as $\lambda = (y_1 - y_2)/(x_1 - x_2)$, but $x_1 - x_2 = 0$.

In Chapter 4, we discuss *complete addition formulas* using projective coordinates.

Elliptic Curve Discrete Logarithm Problem (ECDLP)

Given an elliptic curve E over a finite field $\mathbb{F}_p$, a point $P \in E(\mathbb{F}_p)$, of order[1] n and a point $Q \in \langle P \rangle$, the ECDLP consists in finding the integer $l \in \mathbb{Z}$ such that

$$Q = l \cdot P$$

Here, $\langle P \rangle$ denotes the cyclic group generated by the generator P, and the integer l is called the *discrete logarithm of Q to the base P*, and is denoted by $l = \log_P Q$.

The security of ECC cryptographic schemes relies on the assumption that solving the ECDLP is computationally hard. However, it is important to note that it is easy to solve it in certain classes of elliptic curves, as discussed in [41]. As such, such curves should not be considered for cryptographic purposes. For all other cases, the best known algorithm is either the Pohlig-Hellman algorithm or Pollard's Rho algorithm.

The naive approach for solving ECDLP is an *exhaustive search*, where one computes the elements of the cyclic group $\langle P \rangle$ until the element $Q = l \cdot P$ is found. In this case, given a point $P \in E(\mathbb{F}_p)$ and a scalar $k \in \mathbb{Z}$, we can compute the point $Q = k \cdot P$ by repeatedly adding P with itself:

$$Q = k \cdot P = \underbrace{P \oplus P \oplus \cdots \oplus P}_{k \text{ terms in the sum}}$$

[1] The order of a point P is the smallest positive integer n such that $n \cdot P = \mathcal{O}$. In other words, $\underbrace{P \oplus P \oplus \cdots \oplus P}_{n \text{ terms in the sum}} = \mathcal{O}$.

While the repeated addition is a straightforward method to compute the scalar multiplication, there are more efficient methods. For example, the *double-and-add* method computes $k \cdot P$ in $O(\log k)$ steps.

Algorithm 1 illustrates how this algorithm can be implemented. This algorithm takes as input a t-bit scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$ and a point $P \in E(\mathbb{F}_p)$ and returns the point $Q = k \cdot P$. Essentially, this algorithm is the additive version of the square-and-multiply method for exponentiation. We start with the point $\mathcal{O}$ and iteratively scan all of the bits of the scalar, doubling the current value, and adding P to it if the bit being processed is 1.

Algorithm 1 Left-to-right binary method for point multiplication [47]

Input: t-bit scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_p)$
Output: $Q = k \cdot P$

```
Q ← O
for i from t − 1 down to 0 do
    Q ← 2Q                          ▷ Doubling of Q
    if k_i = 1 then
        Q ← Q ⊕ P                   ▷ Addition of P and Q
    end if
end for
return Q
```

Algorithm 1 processes the bits of the scalar k from the left to the right. However, the same result can be achieved by processing the bits from the right to the left, as illustrated in Algorithm 2.

Algorithm 2 Right-to-left binary method for point multiplication [47]

Input: t-bit scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_p)$
Output: $Q = k \cdot P$

```
Q ← O
for i from 0 to t − 1 do
    if k_i = 1 then
        Q ← Q ⊕ P
    end if
    P ← 2P
end for
return Q
```

It should be noted, however, that this method is vulnerable to side-channel attacks, potentially leaking information on the scalar k. Figure 2.6 illustrates the power trace of the execution of the

double-and-add algorithm on a RISC-V CPU. Because the power consumption of point addition and point doubling can be clearly distinguished from each other, an attacker is able to measure the power consumption and use such information to retrieve the binary representation of the scalar k.

This is particularly dangerous given the fact that, in the Elliptic Curve Digital Signature Algorithm (ECDSA) signature scheme, during the key pair generation process, the public key Q is computed as $Q = d \cdot G$, where $d \in \mathbb{Z}$ is the secret key and G is the base point of the elliptic curve. This means that applying the double-and-add method would leak confidential information about the secret key.

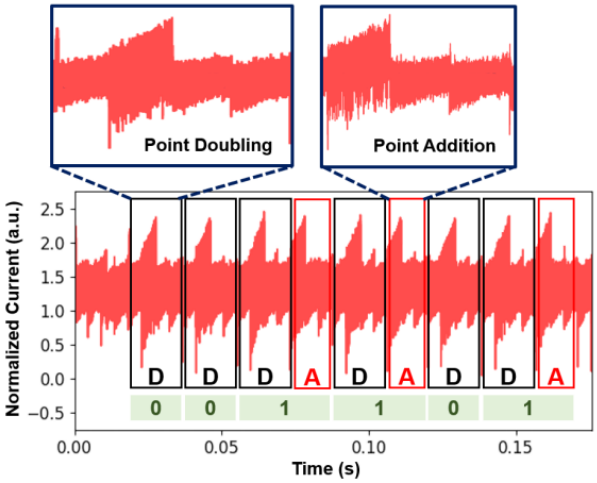


Figure 2.6: Power trace of the double-and-add algorithm on a RISC-V CPU. Adapted from [13]

On the other hand, the Montgomery Ladder [70], illustrated in Algorithm 3, computes the same result but is resistant to side-channel attacks. The algorithm works by iteratively adding and doubling points on the curve, in such a way that allows the efficient computation while still preventing side-channel attacks. Here, it is important to note that, using this algorithm, we make the same number of point additions and doublings regardless of the value of the scalar k. It is because of this that the Montgomery Ladder is resistant to side-channel attacks.

Algorithm 3 Montgomery Ladder for point multiplication

Input: t-bit scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_p)$
Output: $Q = k \cdot P$

$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
for i from 0 to $t-1$ **do**
    if $\mathrm{k}_i = 0$ **then**
        $R_1 \leftarrow R_0 \oplus R_1$
        $R_0 \leftarrow 2 \cdot R_0$
    else
        $R_0 \leftarrow R_0 \oplus R_1$
        $R_1 \leftarrow 2 \cdot R_1$
    end if
end for
return R_0

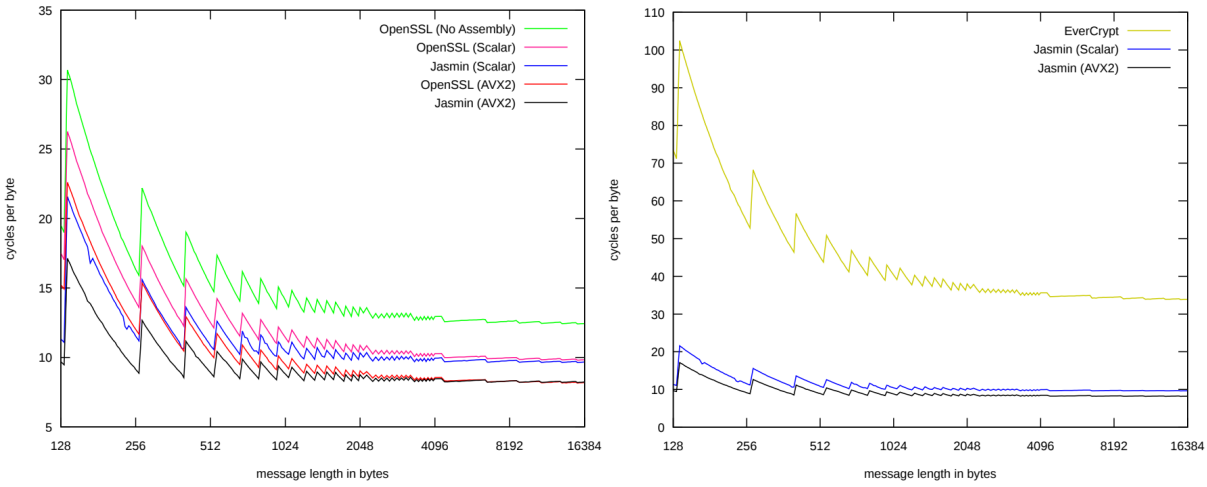
We should also note that, at each iteration of Algorithm 3, the variable R_0 is updated in the same way that it would be updated by using the double-and-add algorithms for point multiplication mentioned above. Furthermore, the variable R_1 is never used in the computation of the final result. Instead, its purpose is to ensure that both branches of the conditional contain an addition and a doubling instruction, thus allowing the efficient computation of the scalar multiplication while also avoiding any potential side-channel attacks.

Chapter 3

Jasmin

This chapter provides an overview of the most relevant features of the Jasmin framework. Jasmin [5, 8] is a verfication-friendly, low-level framework suitable for writing high-assurance and high-speed cryptographic implementations. It combines both high-level abstractions such as variables, arrays, and control flow structures with low-level constructions such as flag manipulation.

As illustrated in Figure 3.1, the Jasmin framework performs at a level comparable to other verified and non-verified implementations. In particular, it is worth mentioning that it achieves a level of performance similar to that of OpenSSL while providing formal guarantees of correctness, memory safety and Constant-Time (CT).



(a) Comparison to non-verified code

(b) Comparison to verified code

Figure 3.1: Performance evaluation of the SHA-3 function with respect to other verified and non-verified implementations [7]

Jasmin has been used in the implementation of multiple cryptographic primitives. Examples include `libjade` [35], a comprehensive library of cryptographic primitives – including the round 3 finalists of the NIST Post-Quantum Cryptography Competition: Dilithum [37] and Falcon [42] for digital signatures, and the Kyber Key Encapsulation Mechanism (KEM) [11] –, the implementation of a MPC-in-the-Head protocol [10], the Schnorr protocol [40], Swoosh [43], a post-quantum lattice-based Non-Interactive Key Exchange (NIKE), and in the implementation of the random password generator component of the PassCert password manager [45].

The Jasmin workflow, illustrated in Figure 3.2, includes several steps. The extension of Jasmin source code is either `.jazz` or `.jinc`. After writing the Jasmin source code, it is compiled into assembly code, which can then be integrated into a larger codebase. The Jasmin compiler is formally verified in the Coq [86] proof assistant. This is discussed in more detail in

Here, it is worth noting that EasyCrypt [16] plays a crucial role in the Jasmin workflow. It allows us to write cryptographic specifications of cryptographic primitives. Additionally, the Jasmin compiler supports the extraction of EasyCrypt models that are semantically equivalent to the Jasmin source program. This ensures that the properties and security proofs established for the extracted model also apply to the executed program. In this case, proving the functional correctness of the Jasmin source program with the EasyCrypt specification consists in proving that the specification and the extracted model are equivalent, and proving that a program is CT consists in proving that the its leakage is independent of secret data. This is illustrated in more detail in §3.2.

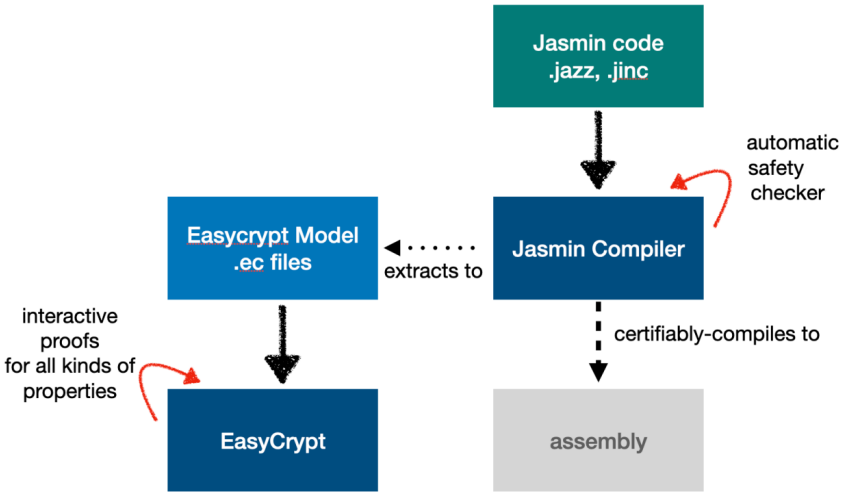


Figure 3.2: Jasmin workflow [78]

3.1 The Jasmin Language

Jasmin can be thought of as lying in between C and `qhasm` [19], a portable assembly language aimed at the efficient implementation of cryptographic algorithms. With assembly-like syntax, and leveraging the formal guarantees enforced by the Jasmin compiler, we can have a finer level of control over the compiled assembly, meaning that achieve optimized implementations of cryptographic functions without compromising security. For a more in-depth description of the Jasmin language, refer to [74].

Syntax

The syntax of Jasmin programs is defined as follows:

$e \in \text{Expr}$	::=	x	register
	|	$op(e, \ldots, e)$	operator
$i \in \text{Instr}$	::=	$x := e$	assignment
	|	$x := a[e]$	load from array a offset e
	|	$a[e] := x$	store to array a offset e
	|	if e then c else c	conditional
	|	while e do c	while loop
$c \in \text{Com}$	::=	$[\,]$	empty, do nothing
	|	$i; c$	sequencing

Table 3.1: Syntax of Jasmin programs. Adapted from [18]

When declaring a variable, we must choose where it will be stored, either in register, indicated by the `reg` keyword, in the stack, indicated by the `stack` keyword, or resolved at compile time, indicated by the `inline` keyword. It is worth noting that, in addition to regular registers, we can access MMX by prepending the variable declaration with the `#mmx` annotation. MMX registers are eight additional 64-bit registers that are generally used for performing SIMD instructions. However, in Jasmin these registers are only used to avoid spilling to the stack. Instead, vectorized instructions are implemented with the AVX/AVX-2 instructions.

Arrays can either be stored in contiguous memory regions in the stack – if declared with the `stack` keyword – or in register – if declared with the `reg` keyword.

In addition, Jasmin supports pointers to (internal) stack arrays, which are declared with

the `ptr` keyword. These pointers can be either mutable – defined with the `mut` keyword – or constant – if declared with the `const` keyword. Besides, pointer variables can be stored in the stack or in register. `reg ptr` are usually used to store the address of an array in a register in order to pass stack arrays to functions, while `stack ptr` variables are used to spill `reg ptr` variables to the stack.

Table 3.2 provides a concise overview of the Jasmin variable allocation, indicating where each variable is allocated – either in register, in the stack or in `.data` section of the assembly file – depending on its type. Here, the **Scalar** column indicates the types that can be hold individual values, while the **Array** column indicates whether each data type can be declared as an array, which allows storing multiple elements of the same type in a contiguous block of memory. For example, we cannot declare an array of booleans and, similarly, pointers must point to stack arrays, i.e. they cannot point to other data types. As an example, trying to declare a `reg ptr` variable that points to a `stack u128` will result in a compilation error. However, we can declare pointers to arrays that only contain one element.

Storage	Type	Scalar	Arrays	Stored in
reg	u8	✓	✓	al bl cl dl sil dil blp r8b-r15b
	u16	✓	✓	ax bx cx dx si di bp r8w-r15w eax ebx ecx edx esi edi ebp r8d-r15d
	u64	✓	✓	rax rbx rcx rdx rsi rdi rbp r8-r15
	u128	✓	✓	xmm0-xmm15
	u256	✓	✓	ymm0-ymm15
	bool	✓	✗	Flags: CF, PF, ZF, SF, OF
	ptr u8-u256	✗	✓	rax rbx rcx rsi rdi rbp r8-r15
stack	u8-u256	✓	✓	stack frame; 1-32 bytes alignment; rsp
	ptr u8-u256	✗	✓	8 bytes in the stack frame
global	u8-u256	✓	✓	.data section; 1-32 bytes alignment
inline	int	✓	✗	statically known
	u8-u256	✓	✗	

Table 3.2: Overview of Jasmin variable allocation. Adapted from [74]

In Jasmin, `for` loops iterate over `inline int` and are automatically unrolled. Because the compiler performs loop unrolling during the compilation process, the number of iterations must be known at compile time. For example, consider the function in Listing 3.1 that intializes the variable `r` to zero, and repeatedly increments it in a loop that runs for 5 iterations.

```
export fn foo() -> reg u64 {
    reg u64 r;
    r = 0;
    inline int i;
    for i = 0 to 5 {
        r += 1;
    }
    return r;
}
```

Listing 3.1: Jasmin for loop

```
foo:
    movq  $0, %rax
    incq  %rax
    incq  %rax
    incq  %rax
    incq  %rax
    incq  %rax
    ret
```

Listing 3.2: Compiled for loop

The compiled assembly shown in Listing 3.2 illustrates how the loop is unrolled by the Jasmin compiler. Here, it is also worth noting that, because the loop is unrolled, the compiled assembly does not contain any branching instructions, thus avoiding potentially branching on secret data.

Unlike for loops, in a while loop, the number of iterations does not need to be known at compile time. Thus, instead of performing loop unrolling, the compiler will test the loop condition every time the body of the loop is executed. This means that the compiled assembly will contain branching instructions.

If we implement the previous example using a while loop, as illustrated in Listing 3.3, we have the following assembly code:

```
export fn bar() -> reg u64 {
    reg u64 r i;
    r = 0;
    i = 0;
    while (i < 5) {
        r += 1;
        i += 1;
    }
    return r;
}
```

Listing 3.3: Jasmin while loop

```
bar:
    movq  $0, %rax
    movq  $0, %rcx
    jmp   Lbar$1
Lbar$2:
    incq  %rax
    incq  %rcx
Lbar$1:
    cmpq  $5, %rcx
    jb    Lbar$2
    ret
```

Listing 3.4: Compiled while loop

Boolean values are stored in the flags of the `rflags` register. These flags are updated after arithmetic operations based on the result. Here, it is important to note that these flags cannot be directly manipulated and, unlike other `reg` variables, booleans cannot be spilled to the stack. Instead, these variables are usually used for control-flow – e.g. conditional moves – and carry propagation.

Consider the `__bn_eq` function illustrated in Listing 3.5 that checks two big numbers for equality in constant-time, returning 1 if they are equal, or 0 otherwise. This function works by iterating over the limbs of each big number, XORing them and storing the result in the `acc` variable. Finally, if the two big numbers are equal, then the value of `acc` is zero. To check if this is the case, the `#AND` operation performs a bitwise AND, which changes the value of the Zero Flag. If both numbers are equal, then the Zero Flag is set to zero. This boolean value is

then used to perform a conditional move to the `res` variable, which is then returned. Here, it is important to note that the `res = are_equal if zf;` line does not compile to a branching instruction. Instead, a `cmov` instruction is used.

```
inline fn __bn_eq(reg ptr u64[NLIMBS] a b) -> reg u64 {
    inline int i;
    reg u64 acc are_equal t res;
    reg bool zf;

    res = 0;
    are_equal = 1;
    acc = 0;

    for i = 0 to NLIMBS {
        t = a[i];
        t ^= b[i];
        acc |= t;
    }

    (_, _, _, _, zf, _) = #AND(acc, acc);
    res = are_equal if zf; // conditional move

    return res;
}
```

Listing 3.5: Big number comparison using flag manipulation

Functions

Jasmin supports three types of functions, which we explain in more detail bellow.

- **Inline Functions:** In this phase, calls to inlines functions are replaced with its actual code. This improves performance by avoiding the overhead of function calls. These type of functions are particularly useful for small functions that are called frequently. Nevertheless, it is important to take into consideration that inlining large functions increases the size of the generated assembly code, making it harder to integrate into larger codebases. In addition, arguments of inline function can be stored either in register or in the stack.
- **Local Functions:** Local functions require arguments to be stored in registers. As such, if a local function needs to operate over stack arrays, a pointer (stored in a register) that points to the stack array needs to be given to the function. Each local function is compiled with a certain calling interface – i.e. the registers where arguments and results are passed –, which is found by the compiler. In other words, each argument is assigned by the compiler to the selected register. For example, considering a `foo` function that takes two arguments, the following function successive calls `foo(a, b)` and `foo(b, a)` do not compile. It is the developer's responsibility to ensure that, when calling the function, each variable is in the correct register. Finally, local functions can also be annotated with the `#[returnaddress="stack"]` annotation[1]. This ensures that the return address is

[1]The default behavior has recently been updated to store the return address in the stack. However, the `#[returnaddress=reg]` allows us to store the return address in a register instead.

placed on the stack instead of in a register. By placing the return address on the stack, we can free up one register that would otherwise be used to store the return address of the function.

- **Export Functions:** Export functions are functions that can be called from other programs – e.g. C or Rust programs. As such, these functions must follow the Application Binary Interface (ABI) of the target platform. In this case, export functions follow the System V ABI, which is the facto standard for Unix operating systems, specifying, among other things, in which registers the functions receive their arguments, and in which register the result is returned. Nevertheless, the System V ABI is only partially supported in the sense that export functions cannot receive arguments in the stack; instead, they have to be stored in registers.

Compiler

The Jasmin compiler – `jasminc` – is written in OCaml and Coq. In addition, the Jasmin compiler is proven to be functionally correct in the Coq proof assistant. In particular, each compilation step preserves the semantics of the Jasmin source program.

Figure 3.3 provides an overview of the compilation steps. During the *Inlining* step, calls to `inline` functions are replaced with the respective function body, and reference to `inline` variables are replaced with their corresponding value. In *Unrolling*, `for` loops are fully unrolled, which requires that the bounds are statically known. Furthermore, in the *Stack Sharing* and *Register Array Expansion* steps, the compiler optimizes the memory layout for local functions and translates the values of `reg` arrays to individual register variables, respectively. At this point, in the *Lowering* step, Jasmin instructions are converted into architecture-dependent low-level instructions. In *Register & Stack allocation* step, the compiler finds a map between each `reg` variable and available registers. Here, if it is not possible to find such a map – i.e. there are more `reg` variables than available registers – compilation fails. Finally, the *Linearization* converts the program into a sequence of instructions that can then be mapped to assembly.

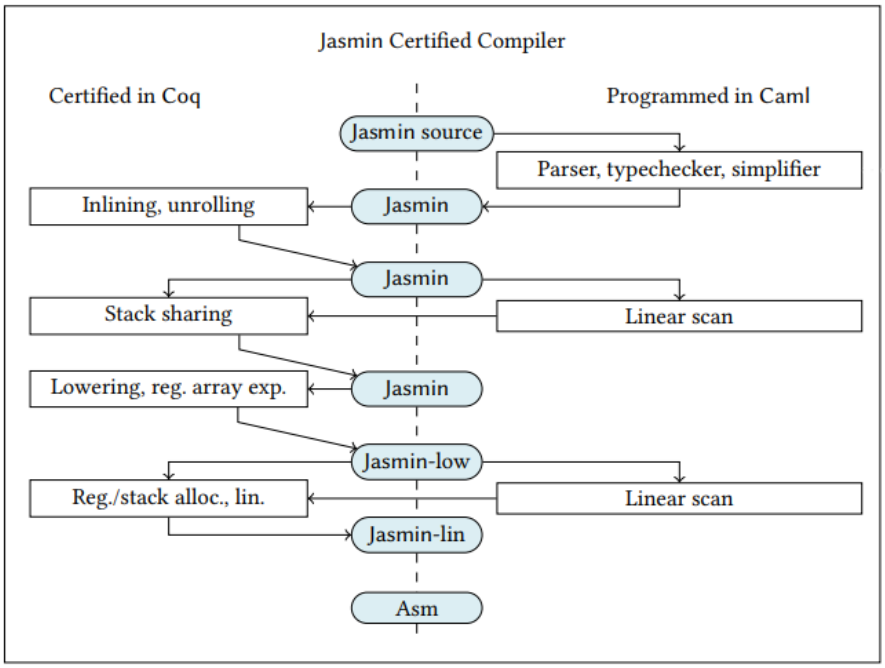


Figure 3.3: `jasminc` compilation steps [5]

Compiler Correctness

As defined in [5], the correctness theorem of the Jasmin compiler states that if a source program is successfully compiled, an execution of the source program corresponds to an execution of the compiled program, for the same initial state – thus preserving its *semantics*.

In this context, it is important to note that it is necessary to have enough stack space to allocate all variables. In other words, for all the exported functions, the source program or the compiled program exhibit the same behaviour. Ultimately, safe source programs are compiled to safe target programs, which execute without run-time errors, unless there is not enough stack space to execute the target program.

3.2 Verification Toolchain

The Jasmin framework offers a safety checker to check the memory-safety of source code. Essentially, it verifies that all array accesses within the program are performed within the bounds of the allocated memory, thereby preventing any potential buffer overflows, that memory accesses target allocated memory, and that arithmetic operations are applied to valid arguments – e.g. there are no divisions by zero. Furthermore, the safety checker attempts to prove that programs eventually terminates – e.g. there is no infinite loops.

Finally, the Jasmin framework also support Constant-Time verification, either through a type-checker, or in EasyCrypt. As an example, consider the `max` function illustrated in Listing 3.6

that computes the maximum value of the array passed as argument. The Jasmin compiler allows the extraction of an EasyCrypt model for constant-time verification. Here, the `leakages` variable in Listing 3.7 is a global variable that is updated based on each operation that results in leakage. In this case, we can see it contains the memory addresses – that result from the array accesses – and the value of the condition of the if statement.

```
fn max(reg ptr u64[2] a)
    -> reg u64 {
    reg u64 r;
    r = a[0];
    if (a[1] > r) { r = a[1]; }
    return r;
}
```

Listing 3.6: Jasmin source program

```
module M = {
  var leakages : leakages_t

  proc max (a:W64.t Array2.t) : W64.t = {
    var aux: W64.t;

    var r:W64.t;

    leakages <- LeakAddr([0]) :: leakages;
    aux <- a.[0];
    r <- aux;
    leakages <- LeakCond((r \ult a.[1]))
        :: LeakAddr([1]) :: leakages;
    if ((r \ult a.[1])) {
      leakages <- LeakAddr([1]) ::
          leakages;
      aux <- a.[1];
      r <- aux;
    } else {

    }
    return (r);
  }
}.
```

Listing 3.7: Leakage trace

At this point, to prove that code is CT, it suffices to check that secret data does not influence the leakage trace. In other words, for any two program executions starting from inputs such that the public data is the same but the secret data may be different, then the leakage trace must be equal.

Similarly, the Jasmin compiler also allows the extraction of an EasyCrypt model that is equivalent to the source program. Because of this equivalence, we have the guarantee that the properties we prove for the EasyCrypt model also hold for the program that is actually executed. Listing 3.8 illustrates an EasyCrypt model extracted by the Jasmin compiler from the `max` function.

```
module M = {
  proc max (a:W64.t Array2.t) : W64.t = {

    var r:W64.t;

    r <- a.[0];
    if ((r \ult a.[1])) {
      r <- a.[1];
    } else {

    }
    return (r);
  }
}.
```

Listing 3.8: Equivalent EasyCrypt model

3.3 Speculative Constant-Time Type System

Recent work [82] introduces a new approach to protect cryptographic implementations against Spectre v1 attacks with minimal performance overhead. Essentially, the authors propose a type system that allows Jasmin code to be formally validated as resilient to speculative execution attacks by enforcing Speculative Constant-Time (SCT) at the source level. In other words, the type system tracks the flow of sensitive data to ensure that memory accesses and branching conditions do not depend on secret data. In addition, this type system is proved to be *sound*, i.e. it only accepts programs that are indeed SCT, rejecting those that are not.

Using this type system, it is possible to protect cryptographic implementations against speculative execution attacks with a minimal overhead – e.g. the authors report less than 1% for the Kyber KEM.

Essentially, the idea is to keep track of a misspeculation flag that is used to track whether the CPU is misspeculating or not. In the case of misspeculation, we use this flag to prevent an attacker from reading secret values by hardening speculative loads.

Security Levels & Types

To represent the security levels of data, we consider two security levels L and H, where L denotes a low security level, and H a high security level.

In addition, a security type is a pair of security levels (τ_n, τ_s), where τ_n denotes the security level in a normal, i.e. non-speculative, execution and τ_s denotes the security level of all executions, which includes those in which the CPU is misspeculating.

With this notion, we can reason about the security types of data in the following way:

- (L, L) represents public data. In Jasmin source code, this is denoted with the `#public` annotatation.

- (H, H) represents secret data. In Jasmin source code, this is denoted with the `#secret` annotatation.
- (L, H) represents transient data, i.e. data that is public under sequential execution but that may depend on secret information during speculative execution. In Jasmin source code, this is denoted with the `#transient` annotatation.

Speculative Constant-Time

Speculative Constant-Time (SCT) is an property that ensures that programs are protected against speculative execution attacks. In simpler terms, this means that a program is SCT if its leakage does not depend on secrets, for every attacker's choice of branch decisions and unsafe memory accesses. Essentially, while a program is CT under sequential execution if it does not leak any sensitive data, a program is SCT if it does not leak any sensitive data for any choice of directives issued by an attacker.

Attacker Model

To model the capabilities of an attacker, we introduce the concepts of *directives*, that represent ways the attacker influences program execution, and *observations*, that represents information gained by an attacker as a consequence of the execution of each instruction.

Program execution is affected by *directives* provided by an adversary, potentially allowing them to steer the program's execution to misspeculated paths. Directives are defined by the following grammar:

$$d \in \mathsf{Dir} ::= \mathsf{step} \mid \mathsf{force} \mid \mathsf{load}\ a, i \mid \mathsf{store}\ a, i$$

The step directive is issued by an attacker to allow the program execution to continue normally. In this context, when encountering a branching instruction, execution resumes at the appropriate branch – i.e. execution does not misspeculate. The force directive is used by an attacker to force the CPU to enter a misspeculated branch, thus changing the value of the misspeculation flag. In fact, this directive is the only one that changes the value of this flag. Finally, regarding speculative memory accesses, the load a, i and store a, i directives are used to an attacker to load values from and write to memory addresses of their choice.

In addition, to model side-channel leakage, we consider the notion of *observation*. Every observation denotes a leak of information that is observable by an attacker. Observations are defined by the following grammar:

$$o \in \mathsf{Obs} ::= \bullet \mid \mathsf{read}\ a, v \mid \mathsf{write}\ a, v \mid \mathsf{branch}\ b$$

Here, • means that the execution of a given instruction does not not leak any observation. In addition, memory accesses leak the address, which is captured by the read a, v and write a, v observations. Finally, the branch b observation captures the notion that branching instructions leak their guards.

Type System Primitives

This type system provides a series of primitives that allows us to implement Selective Speculative Load Hardening (selSLH). These primitives, summarized in Table 3.3, are described in more detail in the following sections.

Jasmin	Semantics	Compiled to
`ms = #init_msf();`	`ms = 0`	`lfence; ms = 0;`
`ms = #set_msf(e, ms);`	`assert(e)`	`ms = -1 if !e;`
`x = #protect(x, ms);`	`assert(ms == 0)`	`x |= ms;`

Table 3.3: Type system primitives

Misspeculation Flag Initialization

The `ms = #init_msf()` statement, usually used at the beginning of a program to ensure that the programs initially executes in a non-speculative manner, sets the misspeculation flag `ms` to 0 and is compiled to `lfence; ms = 0;`. The misspeculation flag is used to track whether execution is misspeculating or not. By convention, when the execution is misspeculating, the flag is set to `-1`, and is set to `0` otherwise.

Because this flag indicates if the executing is misspeculating or not, we can use it to harden speculative loads from memory. In particular, we use it to mask speculative loads from memory – if the execution is not misspeculating, this mask has no effect, and the loaded value can be read correctly. However, if this is not the case, this mask poisons the loaded value, ensuring that we cannot read values from memory should not be accessible. This is achieved using the `#protect` primitive, which will be described in more detail below.

If we only want to insert an `LFENCE` instruction and do not need to track the value of the misspeculation flag for subsequent use, this statement can be replaced with `_ = #init_msf();`, as illustrated in Listing 3.9.

```
fn add(#transient reg u64 x y) -> #public reg u64 {
    #public reg u64 r;
    _ = #init_msf();
    r = x + y;
    return r;
}
```

Listing 3.9: Misspeculation flag initialization

Set Misspeculation Flag

The `ms = #set_msf(e, ms);` statement is used to update the value of the of the misspeculation flag. It is usually used to update the value of the misspeculation flag immediately after a branch instruction that depends on the condition `e`, although in some cases it may be possible to defer updating the flag until a later time.

Listing 3.10 illustrates a function that computes the sum of an array with 100 elements. To do so, we iterate through all of the elements with a while loop. However, because the CPU may speculatively enter the body of the loop, it is necessary to update the value of the misspeculation flag so that it can be used later to "poison" speculative loads from memory in the case of misspeculation. Here, it is also worth noting that `i` is a public variable. In fact, all branching conditions and memory indices need to be public. for the program to be CT under the Baseline (BL) leakage model. Finally, we also update the misspeculation flag at the end of the loop.

```
fn sum(#msf reg u64 ms, reg ptr u64[100] p)
    -> #msf reg u64, #public reg u64 {
    reg bool cond;
    #public reg u64 sum i;

    sum = 0; i = 0;

    while { cond = (i < 100); } (cond) {
        ms = #set_msf(cond, ms);
        sum += p[(int) i];
        i += 1;
    }

    ms = #set_msf(!cond, ms);
    sum = #protect(sum, ms);

   return ms, sum;
}
```

Listing 3.10: Setting misspeculation flag [3]

The `ms = #set_msf(e, ms);` statement compiles to the branchless conditional `ms = -1 if !e`, more specifically, to a single `cmov` instruction, rather than a branching instruction. Here, it is important to note that this statement compiles to two instructions: a `MOV`, and a `CMOVcc`. First, `MOV` initializes a register with the value -1. The following `CMOVcc` is a CT instruction that copies this value to the `ms` variable if a given condition is met, where `cc` denotes the condition code. For example, the statement `ms = #set_msf(cond, ms);` compiles to

`cmovnb` – meaning "move if not below" –, which copies the value -1 to the variable `ms` if the carry flag is not set; on the other hand, the statement `ms = #set_msf(!cond, ms);` compiles to `cmovb` – meaning "move if below" –, copies the value if carry flag is set.

Protect

To mask speculatively loaded values, selSLH is implemented via the `#protect` primitive. In this case, the instruction `x = #protect(x, ms);` is compiled to `x |= ms;`. However, if `x` is a `reg ptr` variable, then the syntax is `#protect_ptr(x, ms);`.

This approach involves "conditionally masking" the register depending on the value of the misspeculation flag. If the CPU is executing correctly, i.e., it is not misspeculating, the value of the variable `ms` is set to 0, and the register remains unchanged. However, if execution is misspeculating, the value of the variable `ms` is set to -1. This means that, when taking the bitwise OR of the value of the register `x` with the value of the misspeculation flag, we "poison" the loaded value, thus preventing speculatively executed code from leaking secret data.

As with the `#set_msf` statement, in some scenarios we can also delay the use of the `#protect` statement. For example, revisiting the snippet of code presented in Listing 3.10, instead of proctecting the `sum` variable, we could protect each loaded value individually, although this would be less efficient. In this case, instead of the `sum = #protect(sum, ms);` statement, we would protect the values loaded from memory inside the body of the for loop, as follows:

```
while { cond = (i < 100); } (cond) {
   ms = #set_msf(cond, ms);
   t = p[(int) i];
   t = #protect(t, ms);
   sum += t;
   i += 1;
}
```

Listing 3.11: Setting misspeculation flag

Annotations

In addition to the primitives discussed previously, this type system also has an explicit `#declassify` annotation for intended leakage, allowing us to treat secret values as public. This is illustrated in Listing 3.12. The `declassify_load` function takes as input a memory address, and returns the value stored at that memory address and the one immediately following. Because the type-checker assumes that all external memory is secret – even if the information stored at that address is public –, if a pointer points to external memory, it is the developer's responsability to explicitly declassify it in order to treat it as a public value, which is done with the `#declassify` annotation.

```
fn declassify_load(#public reg u64 p)
                -> #public reg u64, #secret reg u64 {
   reg u64 pub sec;
   #declassify pub = [p];
   sec = [p + 8];
   return pub, sec;
}
```

Listing 3.12: Declassifying a secret value loaded from memory [3]

Finally, there is the `#nomodmsf` annotation that is used to annotate functions that do not update the misspeculation flag.

Chapter 4

Implementation and Experimental Results

In this chapter, we describe the Jasmin implementation of a type-checked big number library which is proven to be resistant against Spectre v1 attacks. More specifically, we protect `libjbn`, a big number library implemented in Jasmin that provides both integer as well as finite field arithmetic functions against Spectre v1 attacks. We do so by using the type system discussed in the last chapter to implement Selective Speculative Load Hardening (selSLH). In addition, we also discuss modifications we made to the existing source code and provide an overview of the functions we introduced for elliptic curve arithmetic. Following this approach, we are able to protect `libjbn` against Spectre v1 attacks with a relatively low performance overhead – only 3% for point addition, 2% for mixed point addition, 4% for point doubling, and around 1% for scalar multiplication.

`libjbn` Overview

`libjbn` defines a set of basic arithmetic operations for big integers and finite field arithmetic, which include addition, subtraction, or multiplication. At this point, it is important to clarify how big numbers are represented. Because the `x86_64` registers have 64 bits, natively, the CPU performs arithmetic operations on 64-bit integers. However, due to their large size, big integers used in cryptographic implementations cannot be stored in a single machine word. Instead, a big number is represented an array of 64-bit unsigned integers. For example, an array with four 64-bit elements can represent an integer with up to 256 bits. Below, we represent a 256-bit integer:

```
u64[NLIMBS] glob_b = {
    0x5289a0cf641d011f,
    0x9b88257189fed2b9,
    0xa3b365d58dc8f17a,
    0x5bc57ab6eff168ec
};
```

Here, the constant `NLIMBS` refers to the size of the array used to represent each number, which is a global parameter. In this example, `NLIMBS` is 4, meaning that we can represent numbers of up to 256 bits. Similarly, to represent 2048-bit integers, `NLIMBS` would be 32.

In addition, the implementations in this library are generic on `NLIMBS`, meaning that a user that wishes to use the library should define the value of `NLIMBS` along with some other known constants such as the prime number in the context of finite field arithmetic, which are stored in global arrays. The user then compiles the code and ends up with an assembly file targeted for that particular set of parameters.

4.1 Source Code Modifications

This section discusses the modifications we implement in the source code in order to make it resistant to Spectre v1 attacks. At a high level, this process involves a series of steps, which are outlined below.

- Add security type annotations;
- If needed, free one register for the misspeculation flag;
- If needed, change function signature to return the misspeculation flag;
- Update the value of the misspeculation flag after branching instructions using the `#set_msf` primitive;
- If needed, protect loads from memory using the `#protect` and `#protect_ptr` primitives;
- Declassify public values loaded from memory.

These steps will be further discussed in the following sections, accompanied by illustrative examples:

Type Annotations

Initially, we introduce type annotations to each variable. We consider that big numbers are secret, with the exception of the exponent in the exponentiation function, which must be

public. This is the case because there is a conditional statement based on the value of each bit of the exponent, therefore leaking each bit of this number and, as a consequence, its binary representation. As a result, this function cannot be used for computing the exponentiation when the exponent is a secret number. Nevertheless, it can be used when the exponent is a public value – e.g. in field inversions.

Regarding export functions, all arguments must be at least transient – i.e. they can be either transient or secret. We considered these to be transient instead of secret because the arguments of these functions are the memory addresses where the numbers are stored, which are public under sequential execution, but may depend on secret data during speculative execution.

Furthermore, every export function starts by initializing the misspeculation flag, which inserts an `LFENCE` instruction. This gives us the guarantee that the function is not called in a speculative manner. It is important to note that, with SLH, we only need an `LFENCE` instruction at the beginning of each export function instead of an `LFENCE` after each branching instruction.

This is illustrated in Listings 4.1 and 4.2. In this case, it not necessary to track the value of the misspeculation flag because it is not used in subsequent computations. In other cases, however, it may be necessary to track its value, which we will discuss bellow.

```
export fn bn_set0(reg u64 rp) {
    inline int i;

    for i = 0 to NLIMBS {
        [rp + 8 * i] = 0;
    }
}
```

Listing 4.1: `bn_set0` without Spectre v1 protections

```
export fn bn_set0(#transient reg u64 rp) {
    _ = #init_msf();

    inline int i;

    for i = 0 to NLIMBS {
        [rp + 8 * i] = 0;
    }
}
```

Listing 4.2: `bn_set0` with Spectre v1 protections

Free register for the misspeculation flag & update function signature

For some functions – those that have no branching instructions and that do not need to spill values stored in register to the stack, to later retrieve them – an initial `LFENCE` instruction is sufficient to ensure that the code is SCT. If this is not the case, we need to track the value of the misspeculation flag, which is updated immediately after branching instructions. As such, in some cases, it is necessary to free one register for the misspeculation flag. For some local functions, we can trivially free one register by storing the return address in the stack, which can be achieved with the `#[returnaddress="stack"]` annotation. In other cases, however, it necessary to free one register by spilling values to stack. In such scenario, the registers we chose to spill are those that are used less often, as to maximize register usage by keeping frequently used values in register at all times.

Listings 4.3 and 4.4 illustrate this. The function signature of the `fp_inv` function, which was previously

```
inline fn __fp_inv(reg ptr u64[NLIMBS] a r) -> reg ptr u64[NLIMBS]
```

is updated to receive and return the misspeculation flag. In other words, the new function signature is

```
inline fn __fp_inv(reg ptr u64[NLIMBS] a r, #msf reg u64 ms)
-> reg ptr u64[NLIMBS], #msf reg u64.
```

However, at this point, all registers are occupied, meaning that there is not an available register to allocate the misspeculation flag. As such, one register needs to be freed for the misspeculation flag – in this case we spill the value of `ap` to the stack.

```
export
fn fp_inv(reg u64 rp ap) {
    stack u64[NLIMBS] _a _r;
    reg ptr u64[NLIMBS] a r;
    stack u64 _rp;

    _rp = rp;
    _a = __bn_load(ap);
    a = _a;
    r = _r;
    r = __fp_inv(a, r);
    rp = _rp;
    __bn_store(rp, r);
}
```

Listing 4.3: `fp_inv` without Spectre v1 protections

```
export fn fp_inv(#transient reg u64 rp
    ap) {
    #msf reg u64 ms;

    #secret stack u64[NLIMBS] _a _r;
    reg ptr u64[NLIMBS] a r;
    #public stack u64 _ap _rp;

    ms = #init_msf();

    _a = __bn_load(ap);

    _rp = rp; // spill the register
    _ap = ap; // spill the register

    a = _a;
    r = _r;

    r, ms = __fp_inv(a, r, ms);

    rp = _rp; // load from the stack
    rp = #protect(rp, ms);

    __bn_store(rp, r);
}
```

Listing 4.4: `fp_inv` with Spectre v1 protections

Finally, we need update the value of the misspeculation flag after branching instructions, which is done using the `#set_msf` primitive, and to protect speculative loads from memory, which is achieved using the `#protect` and `#protect_ptr` primitives.

Listing 4.5 illustrates the use of the `set_msf` and `protect` primitives. The function `__ecc_ branchless_scalar_mul` iterates over all of the bits of the scalar. Here, the for loop iterates all limbs, and the while loop iterates over the bits of the current limb. In this case, the misspeculation flag is updated immediately after the while loop. Subsequently, this flag is used to harden two speculative loads – first, when reading the current bit of the scalar to the variable `t`, and later when unspilling the value that was stored in the `reg` variable `k`. Considering this case when the CPU is not misspeculating, the misspeculation flag will have the value zero, and

consequently the bitwise OR resulting from the use of the `#protect` primitive will have no effect. However, if the CPU is misspeculating, then the value of the misspeculation flag will be -1, and the `#protect` primitive prevents these values from being loaded to the cache. Finally, the value of the misspeculation flag is updated immediately after the while loop.

```
inline fn __ecc_branchless_scalar_mul(#secret stack u64[NLIMBS] scalar
                                                                r0x r0y r0z
                                                                r1x r1y r1z,
                                      #msf reg u64 ms)
                                   -> #secret stack u64[NLIMBS],
                                      #secret stack u64[NLIMBS],
                                      #secret stack u64[NLIMBS],
                                      #msf reg u64 {

    inline int i; // to iterate over the limbs of a big number

    #public reg u64 k;    // to iterate over the bits of a limb
    #public stack u64 sk; // to iterate over the bits of a limb
    #public reg u64 t; #public stack u64 st; // current limb

    #public reg bool cond;
    #secret reg bool cf;

    // ... implementation omitted for brevity

    for i = 0 to NLIMBS {
        k = 64;
        while { cond = (k > 0); } (cond) {
            ms = #set_msf(cond, ms);
            sk = k; // spill k

            // ... implementation omitted for brevity

            t = scalar[(int) i];
            t = #protect(t, ms);

            _, cf, _, _, _, t = #SHR(t, 1);

            // ... implementation omitted for brevity

            k = sk;              // unspill k
            k = #protect(k, ms); // unspill k
            k -= 1;
        }
        ms = #set_msf(!cond, ms);
    }

    return r0x, r0y, r0z, ms;
}
```

Listing 4.5: `__ecc_branchless_scalar_mul` function with Spectre v1 protections

Integer Arithmetic

All of the integer arithmetic functions of `libjbn` are already Constant-Time (CT) considering both numbers secret, and do not make use of any branching instructions. In fact, the iteration over the limbs of each number is done using a for loop. Because these loops are unrolled, no branching instructions are issued. As a result, to protect these functions, it is sufficient to insert an initial `LFENCE`.

Field Arithmetic

We now discuss the implementation of arithmetic operations over a finite field $\mathbb{F}_p$. These functions are implemented by calling the integer arithmetic functions followed by a reduction modulo p. For most of them, an initial `LFENCE` is sufficient. However, in other cases, in order to free one register, we spill variables to the stack. As a result, we also require a `#protect` statement after unspilling the register.

Here, it is important to note that, unlike the integer arithmetic functions, not all field arithmetic functions can be used when the inputs are secret numbers. For example, exponentiation is implemented with the double-and-add algorithm, as illustrated in Algorithm 4. The Jasmin implementations can be found in Listings A.1 and A.2.

Algorithm 4 Exponentiation algorithm

Input: t-bit base $a = (a_0, a_1, \ldots, a_{t-1})_2 \in \mathbb{F}_p$ and exponent $b = (b_0, b_1, \ldots, t-1)_2 \in \mathbb{F}_p$

Output: $r = a^b$

1: $r \leftarrow 1$
2: $x \leftarrow a$
3: **for** i from 0 to $t-1$ **do**
4: **if** $b_i = 1$ **then**
5: $r \leftarrow r \cdot x$
6: **end if**
7: $x \leftarrow x^2$
8: **end for**
9: **return** r

Because of the conditional expression that evaluates the current bit of the scalar being processed, the binary representation of the scalar is leaked. As a result, this function should only be used when the scalar is a public value. One such example is the `fp_inv` function that computes the multiplicative inverse of a field element. To do so, we simply call the exponentiation function with -1 as the exponent. In [40], the authors implement a version of the exponentiation function that is safe to use then the base number is secret.

4.2 Elliptic Curve Arithmetic

As discussed in §2.4, arithmetic operations over elliptic curves are the foundation of Elliptic Curve Cryptography (ECC). Examples of such cryptographic protocols include the Elliptic Curve Digital Signature Algorithm (ECDSA), used to provide authentication, integrity and non-repudation, and Elliptic Curve Diffie-Hellman (ECDH), which allows two parties to establish a shared secret over an insecure channel. These protocols rely heavily on the efficient implementation of such

operations. As such, in order to implement cryptographic primitives and protocols using elliptic curves, it is necessary to have efficient algorithms for adding points on the curve, as well as computing the scalar multiplication.

In this work, we use the formulas proposed by Renes, Costello, and Batina [77]. Here, it is worth mentioning that these formulas we use are *complete*. Unlike incomplete formulas, complete formulas, despite being slower, can be used to add any two points lying on the elliptic curve, thus offering a better protection against timing side-channel attacks. This is due to the fact that, by always performing the same operations regardless of the operands, we ensure that execution time is constant, making it more difficult for potential attackers to infer information about the operands based on side-channel information.

In addition, these formulas work with points written in their projective form, which, as mentioned in §2.4, is more efficient than their affine counterparts in the sense that don't require field inversions. On another note, in [67], the author proposes a parallelized version of these addition formulas using three CPUs.

Finite field and elliptic curve arithmetic are the foundation of ECC protocols. As such, having these functions implemented, protocol implementation becomes much simpler. For example, let us consider ECDSA. In this case, three algorithms need to be implemented – key generation, signature generation and signature verification. For the key generation algorithm, it suffices to call the Jasmin `#random_bytes` system call, which is the standard way of getting random bytes in Jasmin, followed by a field reduction, to generate the secret key. For the public key, we simply multiply the base point of the curve by the previously computed secret key. For the signature generation algorithm, we first need to compute the hash of the message – which can be done using `libjade`'s SHA-3 implementation, which is resistant to Spectre v1 attacks –, and nonce generation – once again, using Jasmin's `#random_bytes`–, followed by a series of additions, multiplications, and a field inversion, which are all functions provided by `libjbn`. Finally, signature verification can be implemented following the same principle.

For the sake of optimization, multiple implementations choose curves with the parameter $a = -3$. This choice of a leads to faster explicit formulas for point doubling, resulting in more efficient computations. Consequently, the short Weierstrass form of these curves becomes:

$$y^2 = x^3 - 3x + b$$

Or, using projective coordinates:

$$Y^2Z = X^3 - 3XZ^2 + bZ^3$$

Point Addition

Algorithm 5 illustrates the addition formulas for any two points of an elliptic curve. While this algorithm can be used to compute the sum of *any* two point, the mixed addition formula discussed bellow, offers a slight performance improvement. However, it can only be used when the Z coordinate of one of the points is 1.

Algorithm 5 Complete, projective point addition for prime-order short Weierstrass curves $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$, with $a = -3$ [77]

Input: $P = (X_1 : Y_1 : Z_1), Q = (X_2 : Y_2 : Z_2), E : Y^2Z = X^3 - 3XZ^2 + bZ^3$
Output: $P + Q = (X_3 : Y_3 : Z_3)$

1: $t_0 \leftarrow X_1 \cdot X_2$
2: $t_1 \leftarrow Y_1 \cdot Y_2$
3: $t_2 \leftarrow Z_1 \cdot Z_2$
4: $t_3 \leftarrow X_1 + Y_1$
5: $t_4 \leftarrow X_2 + Y_2$
6: $t_3 \leftarrow t_3 \cdot t_4$
7: $t_4 \leftarrow t_0 + t_1$
8: $t_3 \leftarrow t_3 - t_4$
9: $t_4 \leftarrow Y_1 + Z_1$
10: $X_3 \leftarrow Y_2 + Z_2$
11: $t_4 \leftarrow t_4 \cdot X_3$
12: $X_3 \leftarrow t_1 + t_2$
13: $t_4 \leftarrow t_4 - X_3$
14: $X_3 \leftarrow X_1 + Z_1$
15: $Y_3 \leftarrow X_2 + Z_2$
16: $X_3 \leftarrow X_3 \cdot Y_3$
17: $Y_3 \leftarrow t_0 + t_2$
18: $Y_3 \leftarrow X_3 - Y_3$
19: $Z_3 \leftarrow b \cdot t_2$
20: $X_3 \leftarrow Y_3 - Z_3$
21: $Z_3 \leftarrow X_3 + X_3$
22: $X_3 \leftarrow X_3 + Z_3$
23: $Z_3 \leftarrow t_1 - X_3$
24: $X_3 \leftarrow t_1 + X_3$
25: $Y_3 \leftarrow b \cdot Y_3$
26: $t_1 \leftarrow t_2 + t_2$
27: $t_2 \leftarrow t_1 + t_2$
28: $Y_3 \leftarrow Y_3 - t_2$
29: $Y_3 \leftarrow Y_3 - t_0$
30: $t_1 \leftarrow Y_3 + Y_3$
31: $Y_3 \leftarrow t_1 + Y_3$
32: $t_1 \leftarrow t_0 + t_0$
33: $t_0 \leftarrow t_1 + t_0$
34: $t_0 \leftarrow t_0 - t_2$
35: $t_1 \leftarrow t_4 \cdot Y_3$
36: $t_2 \leftarrow t_0 \cdot Y_3$
37: $Y_3 \leftarrow X_3 \cdot Z_3$
38: $Y_3 \leftarrow Y_3 + t_2$
39: $X_3 \leftarrow t_3 \cdot X_3$
40: $X_3 \leftarrow X_3 - t_1$
41: $Z_3 \leftarrow t_4 \cdot Z_3$
42: $t_1 \leftarrow t_3 \cdot t_0$
43: $Z_3 \leftarrow Z_3 + t_1$

Mixed Point Addition

The mixed point addition offer an optimization over the standard addition formulas. If the Z coordinate of one of the points is 1, then, using the mixed addition formula, we can compute the result of the addition more efficiently.

Algorithm 6 Complete, mixed point addition for prime order short Weierstrass curves $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$, with $a = -3$ [77]

Input: $P = (X_1 \colon Y_1 \colon Z_1), Q = (X_2 \colon Y_2 \colon 1), E\colon Y^2Z = X^3 - 3XZ^2 + bZ^3$
Output: $P + Q = (X_3 \colon Y_3 \colon Z_3)$

1: $t_0 \leftarrow X_1 \cdot X_2$
2: $t_1 \leftarrow Y_1 \cdot Y_2$
3: $t_3 \leftarrow X_2 + Y_2$
4: $t_4 \leftarrow X_1 + Y_1$
5: $t_3 \leftarrow t_3 \cdot t_4$
6: $t_4 \leftarrow t_0 + t_1$
7: $t_3 \leftarrow t_3 - t_4$
8: $t_4 \leftarrow Y_2 \cdot Z_1$
9: $t_4 \leftarrow t_4 + Y_1$
10: $Y_3 \leftarrow X_2 \cdot Z_1$
11: $Y_3 \leftarrow Y_3 + X_1$
12: $Z_3 \leftarrow b \cdot Z_1$
13: $X_3 \leftarrow Y_3 - Z_3$
14: $Z_3 \leftarrow X_3 + X_3$
15: $X_3 \leftarrow X_3 + Z_3$
16: $Z_3 \leftarrow t_1 - X_3$
17: $X_3 \leftarrow t_1 + X_3$
18: $Y_3 \leftarrow b \cdot Y_3$
19: $t_1 \leftarrow Z_1 + Z_1$
20: $t_2 \leftarrow t_1 + Z_1$
21: $Y_3 \leftarrow Y_3 - t_2$
22: $Y_3 \leftarrow Y_3 - t_0$
23: $t_1 \leftarrow Y_3 + Y_3$
24: $Y_3 \leftarrow t_1 + Y_3$
25: $t_1 \leftarrow t_0 + t_0$
26: $t_0 \leftarrow t_1 + t_0$
27: $t_0 \leftarrow t_0 - t_2$
28: $t_1 \leftarrow t_4 \cdot Y_3$
29: $t_2 \leftarrow t_0 \cdot Y_3$
30: $Y_3 \leftarrow X_3 \cdot Z_3$
31: $Y_3 \leftarrow Y_3 + t_2$
32: $X_3 \leftarrow t_3 \cdot X_3$
33: $X_3 \leftarrow X_3 - t_1$
34: $Z_3 \leftarrow t_4 \cdot Z_3$
35: $t_1 \leftarrow t_3 \cdot t_0$
36: $Z_3 \leftarrow Z_3 + t_1$

Point Doubling

Algorithm 7 illustrates the doubling formulas for *any* point of an elliptic curve. Alternatively, we could use the standard addition formula with the same point for both arguments. Nevertheless, the doubling function is slightly more efficient.

Algorithm 7 Exception-free point doubling for prime order short Weierstrass curves $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$, with $a = -3$ [77]

Input: $P = (X_1 : Y_1 : Z_1), E\colon Y^2Z = X^3 - 3XZ^2 + bZ^3$
Output: $2 \cdot P = (X_3 : Y_3 : Z_3)$

1: $t_0 \leftarrow X \cdot X$
2: $t_1 \leftarrow Y \cdot Y$
3: $t_2 \leftarrow Z \cdot Z$
4: $t_3 \leftarrow X \cdot Y$
5: $t_3 \leftarrow t_3 + t_3$
6: $Z_3 \leftarrow X \cdot Z$
7: $Z_3 \leftarrow Z_3 + Z_3$
8: $Y_3 \leftarrow b \cdot t_2$
9: $Y_3 \leftarrow Y_3 - Z_3$
10: $X_3 \leftarrow Y_3 + Y_3$
11: $Y_3 \leftarrow X_3 + Y_3$
12: $X_3 \leftarrow t_1 - Y_3$
13: $Y_3 \leftarrow t_1 + Y_3$
14: $Y_3 \leftarrow X_3 \cdot Y_3$
15: $X_3 \leftarrow X_3 \cdot t_3$
16: $t_3 \leftarrow t_2 + t_2$
17: $t_2 \leftarrow t_2 + t_3$
18: $Z_3 \leftarrow b \cdot Z_3$
19: $Z_3 \leftarrow Z_3 - t_2$
20: $Z_3 \leftarrow Z_3 - t_0$
21: $t_3 \leftarrow Z_3 + Z_3$
22: $Z_3 \leftarrow Z_3 + t_3$
23: $t_3 \leftarrow t_0 + t_0$
24: $t_0 \leftarrow t_3 + t_0$
25: $t_0 \leftarrow t_0 - t_2$
26: $t_0 \leftarrow t_0 \cdot Z_3$
27: $Y_3 \leftarrow Y_3 + t_0$
28: $t_0 \leftarrow Y \cdot Z$
29: $t_0 \leftarrow t_0 + t_0$
30: $Z_3 \leftarrow t_0 \cdot Z_3$
31: $X_3 \leftarrow X_3 - Z_3$
32: $Z_3 \leftarrow t_0 \cdot t_1$
33: $Z_3 \leftarrow Z_3 + Z_3$

Scalar Multiplication

We implement scalar multiplication using the Montgomery Ladder, as illustrated in Algorithm 3. It worth mentioning that this function should only be used when the scalar is a public number. Here, the **if** $k_i = 0$ statement leaks the i-th bit of the scalar k, and eventually the whole binary representation of the scalar will be leaked.

As a result, it can not be used for computing the scalar multiplication when the scalar is secret – e.g. when, in ECDSA, we compute the public key as $Q = d \cdot G$, where $d \in \mathbb{Z}$ is the private key, and $G \in E(\mathbb{F}_p)$ is the base point of the curve. In such scenarios, we can use a branchless version of the Montgomery Ladder, which we discuss bellow.

To address this issue, we also implement the Montgomery Ladder algorithm replacing the if statement with a conditional move, thus avoiding a branch on secret data. This is illustrated in Algorithm 8. It is worth noting that this function is slightly less efficient than the the previous in the sense that, both results of the if statement must be computed. In other words, for each bit of the scalar, we compute R_0 and R_1 for both scenarios – whether the bit is set to 1 or to 0 –, and perform a conditional move accordingly.

Algorithm 8 Branchless Montgomery Ladder for scalar multiplication

Input: t-bit scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_p)$
Output: $Q = k \cdot P$

$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
for i from 0 to $t - 1$ **do**
    $t \leftarrow R_0 \oplus R_1$
    $R_1 \leftarrow t$
    $R_0 \leftarrow 2 \cdot R_0$
    $R_0 \leftarrow t$ if $k_i = 1$ ▷ Conditional move (branchless)
    $R_1 \leftarrow 2 \cdot R_1$ if $k_i = 1$ ▷ Conditional move (branchless)
end for
return R_0

Point Normalization

Converting the projective representation of a point of an elliptic curve to the corresponding affine representation consists in transforming $(X : Y : Z),\ Z \neq 0$ into (x, y), where $x = X \cdot Z^{-1}$ and $y = Y \cdot Z^{-1}$. This requires a field inversion to compute Z^{-1} and, after that, this value can be multiplied with X and Y to compute x and y, respectively.

While conversion from projective coordinates to affine coordinates requires less arithmetic operations compared to the addition formulas presented in this chapter, it is, in fact, slower. This is mainly due to the fact that field inversions are computationally expensive.

4.3 Performance Evaluation

To evaluate the performance of this library, we conducted a series of benchmarks on a machine with an AMD Ryzen 7 4700U CPU clocked at 2 GHz with hyperthreading and frequency boost disabled, running Ubuntu 22.04.2 LTS. We also used version 11.3.0 of the `gcc` compiler and the latest version (commit `d46b227b2c7a0554bd9f503df70a8fa79a277b19`) from the `glob_array3_slh`[1] branch of the repository for the Jasmin compiler[2]. We also require frequency boost to be disabled to make these measurements less susceptible to inaccuracies caused by unexpected and temporary frequency spikes caused by the frequency boost. For each benchmark, we gathered a series of 10000 timings.

Tables 4.1, 4.4 and 4.8 report the median of the measured timings for each function, both the CT and SCT implementations. In the other tables, we also report the first and third quartiles, represented by **p25** and **p75**, respectively.

Integer Arithmetic

Table 4.1 and Figure 4.1 illustrate the median of the timings we measured for the CT and SCT implementations. While looking at these tables, the performance penalty arising from the Spectre v1 protections may seem substantial when expressed as a percentage, in reality the overhead is only 40 clock cycles, which is negligible. Here, it is important to note that this high percentage is due to the small size of the code being evaluated.

In fact, the only protection applied to this code was the initial `LFENCE` instruction, and there is no workaround for it. As demonstrated in the following sections, for larger functions, the impact of is not as significant.

[1] This branch no longer exists as the speculative type system is now supported by the `main` branch.

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

Function	CT (Clock Cycles)	SCT (Clock Cycles)	Overhead (%)
bn_addn	60	100	66.67
bn_copy	60	100	66.67
bn_eq	60	100	66.67
bn_muln	100	140	40
bn_set0	60	100	66.67
bn_sqrn	100	140	40
bn_subn	60	100	66.67
bn_test0	60	100	66.67

Table 4.1: Performance comparison of integer arithmetic functions for 4 limbs

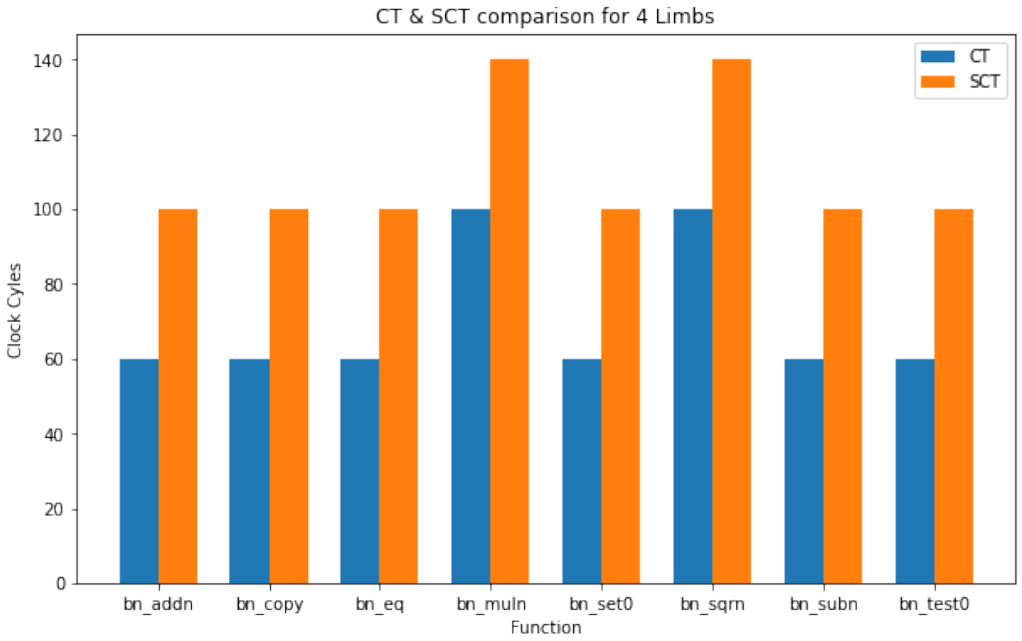


Figure 4.1: Performance comparison of integer arithmetic functions for 4 limbs

In addition, in Tables 4.2 and 4.3, we report the median, and the first and third quartiles of the measured timings for both the CT and SCT implementations, respectively. These statistical measures allows us to have a better understanding of the distribution of the values. As we can see, the values for the median, and the first and third quartile are relatively close to each other. These suggests that the number of clock cycles that each function takes to execute is relatively consistent across different executions.

Function	Median (Clock Cycles)	p25 (Clock Cycles)	p75 (Clock Cycles)
bn_addn	60	40	60
bn_copy	60	40	60
bn_eq	60	40	60
bn_muln	100	100	100
bn_set0	60	40	60
bn_sqrn	100	80	100
bn_subn	60	40	60
bn_test0	60	40	60

Table 4.2: Performance of CT integer arithmetic functions for 4 limbs

Function	Median (Clock Cycles)	p25 (Clock Cycles)	p75 (Clock Cycles)
bn_addn	100	100	120
bn_copy	100	80	100
bn_eq	100	100	100
bn_muln	140	140	140
bn_set0	100	80	100
bn_sqrn	140	140	160
bn_subn	100	100	120
bn_test0	100	100	100

Table 4.3: Performance of SCT integer arithmetic functions for 4 limbs

Field Arithmetic

Table 4.4 and Figure 4.2 illustrate the median of the timings we measured for the CT and SCT implementations. As already discussed, for relatively small functions – e.g. `fp_add`, the cost of these protections may appear substantial when expressed as a percentage. However, it is important to note that this corresponds to only a few dozen cycles. For larger functions – e.g. `fp_inv` – the impact of these protections is not as significant.

Here, it is also worth noting that the slight performance increase in `fp_expm_noct` can be attributed to how we choose to free one register for the misspeculation flag. While in the remaining functions we free one register by spilling one `reg` variable to the stack, in this case we free one register by iterating over the bits of each limb using a for loop instead of a while loop. As such, the iteration variable is an `inline int` instead of a `reg` variable. The main disadvantage of this approach is that it leads to a considerable increase in the number of lines in

the compiled assembly code due to loop unrolling (cf. Table 4.5)[3]. By unrolling the loop, we can prevent a few clock cycles from being wasted due to mispredictions that would have otherwise potentially occurred, and due to updates and spill to this variable. Overall, the `fp_expm_noct` function has a negligible performance increase instead of the expected decrease because this outweighs the penalty imposed by the initial `LFENCE` instruction. However, it should be noted that this may have limited impact outside of the benchmarking environment.

Function	**CT (Clock Cycles)**	**SCT (Clock Cycles)**	**Overhead (%)**
`fp_add`	80	160	100
`fp_expm_noct`	35760	35560	-0.56
`fp_fromM`	120	180	50
`fp_inv`	63180	63520	0.54
`fp_mul`	220	280	27.27
`fp_sqr`	220	280	27.27
`fp_sub`	80	140	75
`fp_toM`	140	200	42.86

Table 4.4: Performance comparison of finite field arithmetic functions for 4 limbs

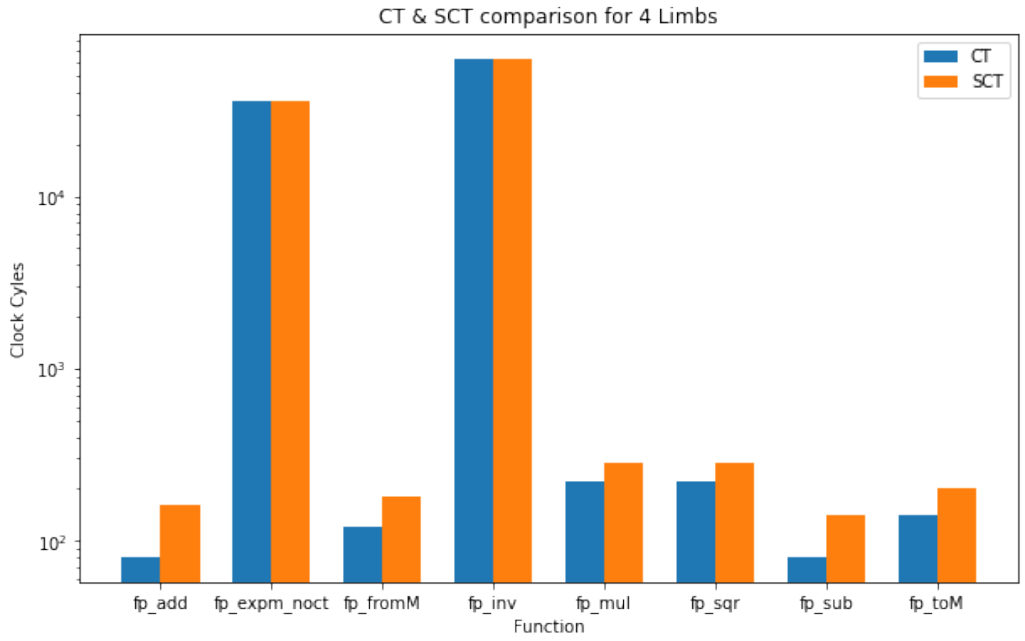


Figure 4.2: Performance comparison of finite field arithmetic functions for 4 limbs

[3]When compiling only the `fp_expm_noct` function for 4 limbs, the CT implementation consists of 885 lines of assembly, while the SCT implementation consists of 6439 lines.

	CT (Lines of Code)	**SCT (Lines of Code)**
Integer Arithmetic	472	488
Finite Field Arithmetic	1973	8023
Elliptic Curve Arithmetic	18466	37436

Table 4.5: Comparison of the lines of assembly between the CT and SCT implementations for 4 limbs

In addition, Tables 4.6 and 4.7 report report the median, and the first and third quartiles of the measured timings for both the CT and SCT implementations. As before, these values are relatively close to each other, suggesting that the number of cycles each function takes to execute does not vary significantly across different executions.

Function	**Median (Clock Cycles)**	**p25 (Clock Cycles)**	**p75 (Clock Cycles)**
`fp_add`	80	60	80
`fp_expm_noct`	35760	35720	35820
`fp_fromM`	120	120	120
`fp_inv`	63180	63120	63220
`fp_mul`	220	200	220
`fp_sqr`	220	220	220
`fp_sub`	80	60	80
`fp_toM`	140	140	140

Table 4.6: Performance of CT finite field arithmetic functions for 4 limbs

Function	**Median (Clock Cycles)**	**p25 (Clock Cycles)**	**p75 (Clock Cycles)**
`fp_add`	160	140	160
`fp_expm_noct`	35560	35520	35640
`fp_fromM`	180	160	180
`fp_inv`	63520	63460	63580
`fp_mul`	280	280	280
`fp_sqr`	280	280	300
`fp_sub`	140	140	160
`fp_toM`	200	200	200

Table 4.7: Performance of SCT finite field arithmetic functions for 4 limbs

Elliptic Curve Arithmetic

Table 4.8 and Figure 4.3 illustrate the median of the timings we measured for the CT and SCT implementations. The values illustrated here emphasize that it is possible to proctect cryptographic implementations against Spectre v1 with a relatively low performance overhead.

Furthermore, the small performance increase observed in `ecc_normalize` can be attributed to the use of the `fp_expm_noct` function to compute the multiplicative inverse of the Z coordinate of the projective representation of the point being processed. As mentioned previously, the protected version of this function is slightly more efficient than the unprotected counterpart.

Function	CT (Clock Cycles)	SCT (Clock Cycles)	Overhead (%)
`ecc_add`	2480	2560	3.23
`ecc_branchless_scalar_mul`	1732380	1741960	0.55
`ecc_double`	2200	2280	3.64
`ecc_mixed_add`	2180	2220	1.83
`ecc_normalize`	63980	63380	-0.94
`ecc_scalar_mul`	1178220	1187860	0.82

Table 4.8: Performance comparison of elliptic curve arithmetic functions for 4 limbs

In addition, Tables 4.9 and 4.10 report the median, and the first and third quartiles of the measured timings for both the CT and SCT implementations. Here, for the addition and normalization functions, these values are relatively close to each other, indicating that the number of cycles these functions take to execute is consistent across executions. However, these values are not as close for the scalar multiplication functions. This can by explained by the large number of point addition and doubling it requires to compute the result of a scalar multiplication.

Function	Median (Clock Cycles)	p25 (Clock Cycles)	p75 (Clock Cycles)
`ecc_add`	2480	2460	2500
`ecc_branchless_scalar_mul`	1732380	1730780	1735340
`ecc_double`	2200	2180	2220
`ecc_mixed_add`	2180	2160	2180
`ecc_normalize`	63980	63920	64080
`ecc_scalar_mul`	1178220	1177100	1179700

Table 4.9: Performance of CT elliptic curve arithmetic functions for 4 limbs

Function	Median (Clock Cycles)	p25 (Clock Cycles)	p75 (Clock Cycles)
ecc_add	2560	2560	2560
ecc_branchless_scalar_mul	1741960	1740920	1743940
ecc_double	2280	2260	2280
ecc_mixed_add	2220	2220	2220
ecc_normalize	63380	63360	63440
ecc_scalar_mul	1187860	1186900	1189160

Table 4.10: Performance of SCT elliptic curve arithmetic functions for 4 limbs

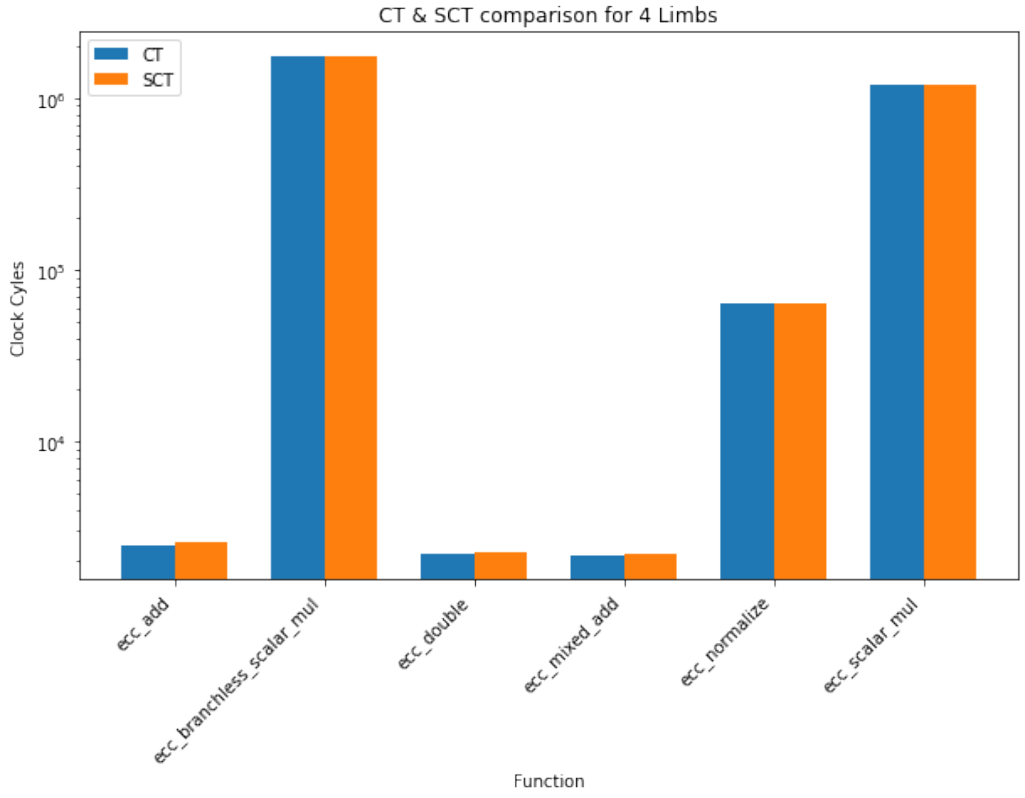
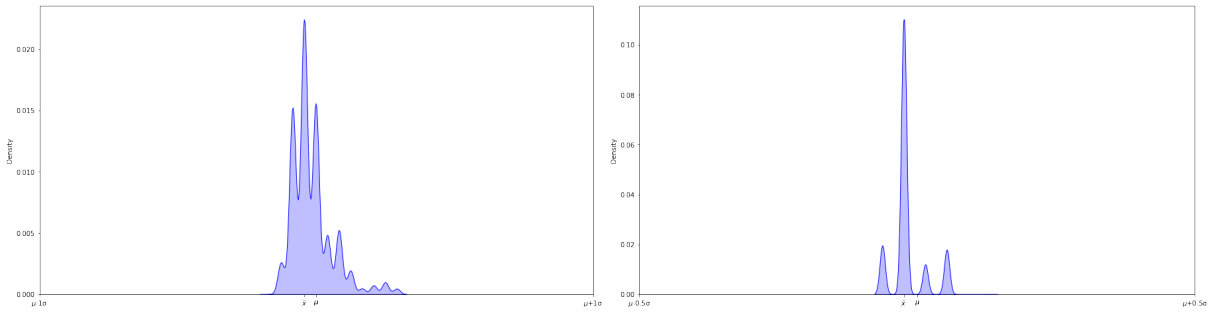


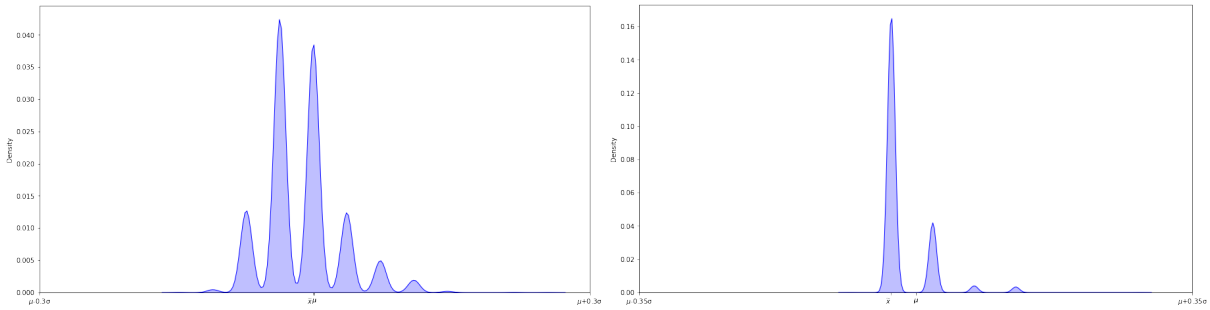
Figure 4.3: Performance comparison of elliptic curve arithmetic functions for 4 limbs

Finally, looking at Figures 4.4 to 4.9, we can see that the majority of the data points are concentrated around the $\mu \pm \sigma$, meaning that there are no significant skewness or outliers. Exceptions to these are the functions for scalar multiplication, for which some data points around $\mu + 3\sigma$ – for the branchless implementation – and $\mu + 4\sigma$ – for the implementation using the Montgomery Ladder – also exist, and the function for normalization, for which some data points around $\mu + 2\sigma$ also exist. Here, μ denotes the mean of the measured values, and σ denotes the respective standard deviation.



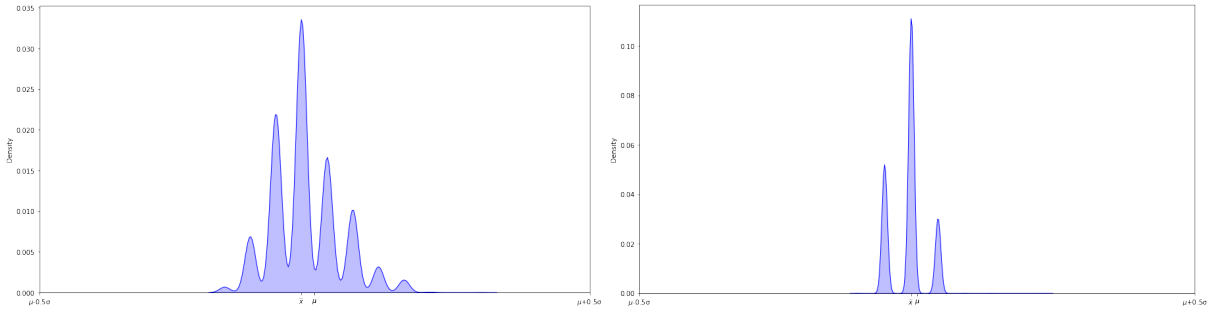
(a) CT (b) SCT

Figure 4.4: Distribution of cycle count for the `ecc_add` function



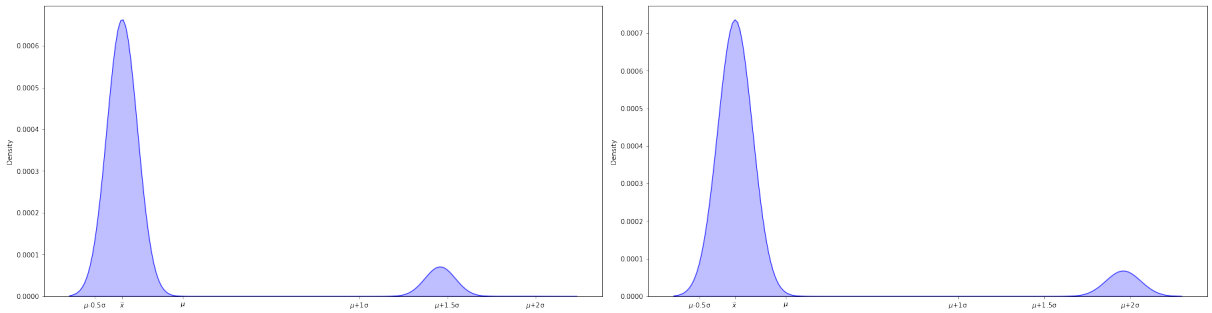
(a) CT (b) SCT

Figure 4.5: Distribution of cycle count for the `ecc_mixed_add` function



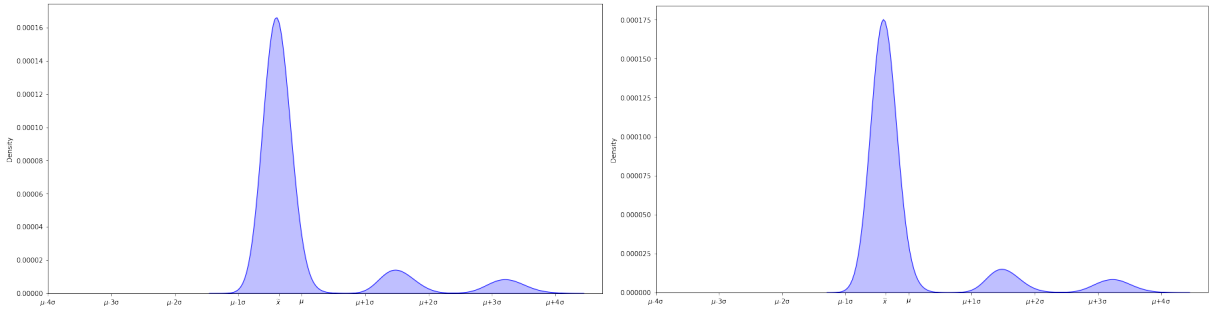
(a) CT (b) SCT

Figure 4.6: Distribution of cycle count for the `ecc_double` function



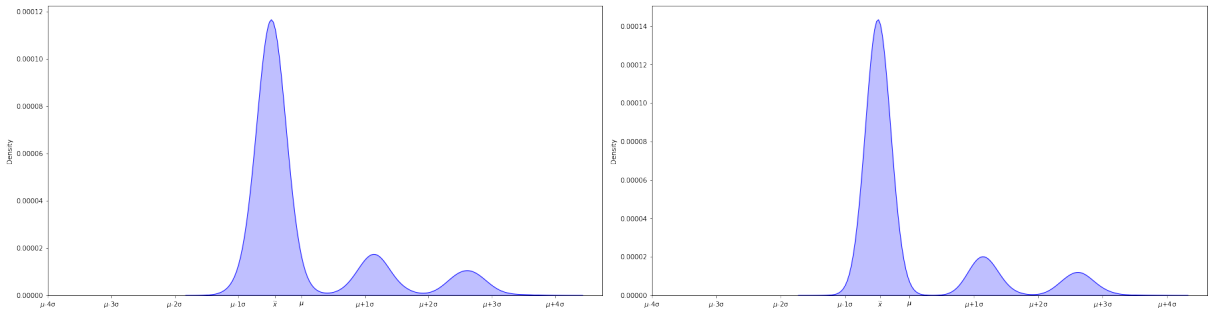
(a) CT (b) SCT

Figure 4.7: Distribution of cycle count for the `ecc_normalize` function



(a) CT (b) SCT

Figure 4.8: Distribution of cycle count for the `ecc_scalar_mul` function



(a) CT (b) SCT

Figure 4.9: Distribution of cycle count for the `ecc_branchless_scalar_mul` function

Furthermore, as the number of limbs increases, the difference in performance between the CT and the SCT implementations decreases. This happens because the performance penalty imposed by the initial `LFENCE` instruction is amortized by the subsequent code. Nonetheless, it is important to note that the impact of this effect is not considerable. This is illustrated in

Figure 4.10, where we plot the median number of cycles of the `fp_toM` for the interval between 1 and 12 limbs. Here, it is worth noting that the choice of this function has no significance; it is just illustrative. In fact, all other functions exhibit the same behaviour.

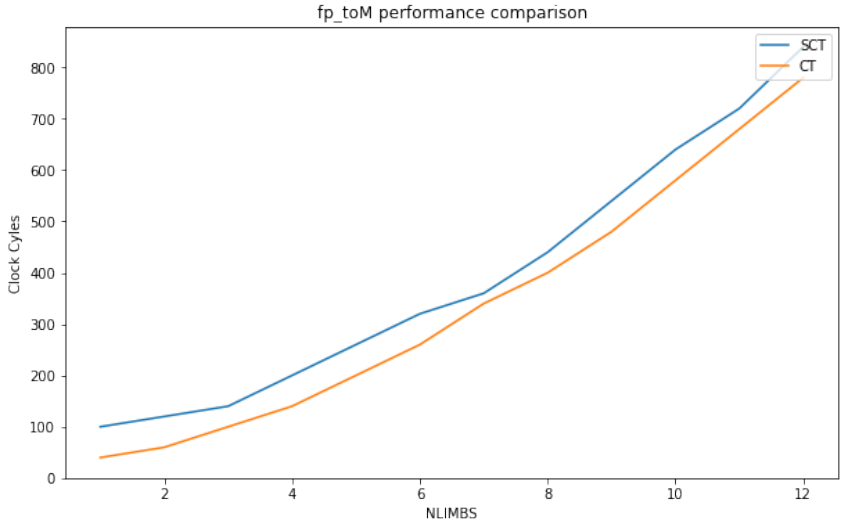


Figure 4.10: Cycle Count in terms of the number of limbs for the `fp_toM` function

Even though all function exhibit this behaviour, for integer arithmetic functions this trend is not so evident in such small intervals due to their relatively small code size. However, if we consider a larger interval, such as from 1 to 100 limbs as illustrated in Figure 4.11, this trend becomes clear. Here, it is important to note that the example of the `bn_addn` function serves only to exemplify this pattern. Upon examining the behavior of other functions, we can see that they all follow the same pattern.

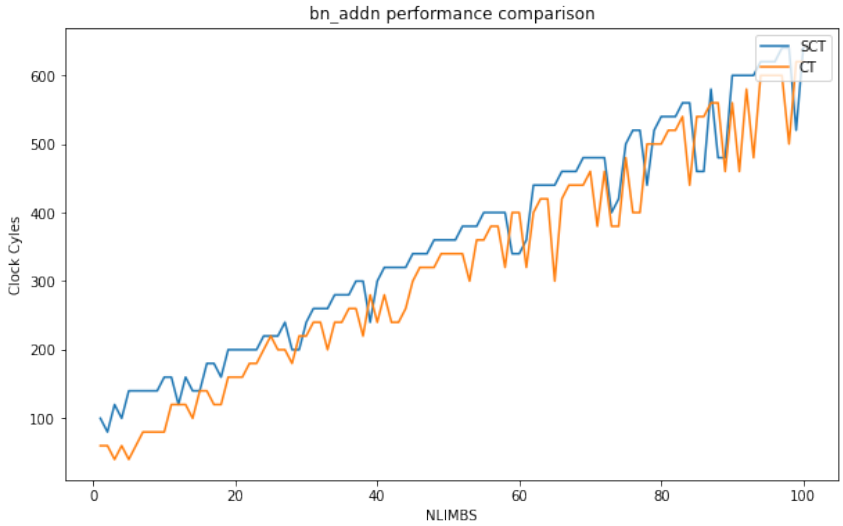


Figure 4.11: Cycle Count in terms of the number of limbs for the `bn_addn` function

Chapter 5

Conclusion & Future Work

In this dissertation, we explored speculative execution attacks, their potential impact and possible solutions to mitigate them. By considering an information flow type system, we were able to protect `libjbn`, a Jasmin big number library, against these type of attacks. Finally, we also discussed the generic implementation of arithmetic operations over prime-order short Weierstrass elliptic curves defined over finite fields, which can be used to implement *all* prime-order curves where $a = -3$. Notably, the implementation of these protections incurs minimal overhead, effectively ensuring protection against Spectre attacks while maintaining an acceptable level of performance.

5.1 Limitations & Future Work

The major limitation of this work is the fact that the Jasmin compiler is not proved to preserve the protections enforced by the type system. In other words, even though the speculative type checker does not reject our code, we do not have the guarantee that the compiled assembly is SCT. However, instead of proving that cryptographic code is SCT at the source level, we can, instead, prove it after certain a certain compilation pass[1]. Intuitively, the closer to the assembly level we check that a given program is SCT, the stronger our confidence in the protections enforced by the type system becomes.

Regarding the arithmetic operations over prime-field elliptic curves, we only considered speculative execution attacks. However, our implementation is vulnerable to other kinds of side-channel attacks – e.g. DPA attacks. Considering this, in [30], for computing the scalar multiplication, the authors protect against DPA attacks by adding a *randomized scalar splitting*. In other words, the computation of $k \cdot P,\ k \in \mathbb{Z},\ P \in E(\mathbb{F}_p)$ is split into two scalar multiplications: $r \cdot P$ and $(k - r) \cdot P$, where $r \in \mathbb{Z}$ is a random number. The result $k \cdot P$ is then computed by

[1]The compiler flag `-checkSCT` checks if the code is SCT at the source level. while the flag `-checkSCTafter` checks if the code is SCT after a given compilation pass – e.g. running `jasminc -checkSCTafter ralloc` checks if the code is SCT after the register allocation pass.

adding these intermediate results.

Another countermeasure is the use of *Randomized Projective Coordinates*, proposed in [31]. Essentially, instead of representing the point $P \in E(\mathbb{F}_p)$ as $(X : Y : Z)$, we represent them as $(\lambda X : \lambda Y : \lambda Z)$, where λ is a random element of the finite field $\mathbb{F}_p$.

Despite the encouraging results we have achieved, there is still room for improvement. This improvement is twofold, encompassing both optimized implementations and formal proofs of correctness, which we discuss in more detail below.

Optimized Implementations

As discussed in Chapter 4, the performance overhead imposed by the Spectre v1 protections stems from two main factors: the initial `LFENCE` instruction at the beginning of every function, and register spilling. While the penalty arising from the use of fence instructions is inevitable, we can reduce the impact of spilling registers by spilling them to MMX registers instead of spilling them to the stack. In this case, by spilling values to these register instead of spilling them to the stack, it would not be necessary to insert `#protect` statements after unspilling them. This is left as future work.

In addition, the work presented in this dissertation is not an optimized implementation. It serves only to demonstrate that we can protect Jasmin source code against Spectre v1 attacks with minimal overhead. Nevertheless, it serves as a starting point for the development of optimized implementations, which can be achieved by taking advantage of SIMD instructions such as the AVX/AVX-2 vectorized instructions.

Formal Proof of Correctness

As discussed in Chapter 3, EasyCrypt plays an important role in the Jasmin workflow. After ensuring that the speculative type checker accepts this code, the next logical step is to prove that the source code correctly computes the results. While we have tested the results computed by the Jasmin implementation against a C implementation for 5000 random points, stronger guarantees are needed. In this case, this would require extracting the Jasmin implementation to an equivalent EasyCrypt model, which is then used to prove certain properties of the source code, namely that the program is functionally correct.

Appendix A

Jasmin Source Code

```
#[returnaddress="stack"]
fn _fp_exp(reg ptr u64[NLIMBS] a _b r) -> reg ptr u64[NLIMBS] {
    inline int j;
    reg u64 k t;
    reg bool cf;
    stack u64[NLIMBS] _x;
    reg ptr u64[NLIMBS] x;
    stack u64 ss;
    stack ptr u64[NLIMBS] rr bb;
    reg ptr u64[NLIMBS] b;
    reg ptr u64[NLIMBS] glob_oneMp;

    x = _x;
    glob_oneMp = glob_oneM;

    x = __bn_copy2(a, x);
    r = __bn_copy2(glob_oneMp, r);

    _x = x;
    bb = _b;
    rr = r;

    for j = 0 to NLIMBS {
        b = bb;
        t = b[(int) j];
        k = 64;

        while (k != 0) {
            ss = k;
            _, cf, _, _, _, t = #SHR(t, 1);

            if (cf) {
                r = rr;
                x = _x;
                r = _fp_mulU(r, x);
                _x = x;
                rr = r;
            }

            x = _x;
            x = _fp_sqrU(x);
            _x = x;
            k = ss;
            k -= 1;
        }
    }

    r = rr;
    return r;
}
```

Listing A.1: `_fp_exp` function without Spectre v1 protections

```
#[returnaddress="stack"]
fn _fp_exp(reg ptr u64[NLIMBS] a _b r,
           #msf reg u64 ms) -> reg ptr u64[NLIMBS], #msf reg u64 {
    #secret stack u64[NLIMBS] _x;

    reg ptr u64[NLIMBS] x;
    reg ptr u64[NLIMBS] glob_oneMp;

    stack ptr u64[NLIMBS] glob_oneMps;

    stack ptr u64[NLIMBS] rr bb; // used to spill r and _b
    reg ptr u64[NLIMBS] b;

    inline int j k;
    #public reg bool cf;
    #public reg u64 t;

    x = _x;

    glob_oneMp = glob_oneM;

    x = __bn_copy2(a, x);
    r = __bn_copy2(glob_oneMp, r);

    _x = x;

    bb = _b;
    rr = r;

    for j = 0 to NLIMBS {
        b = bb;
        b = #protect_ptr(b, ms);

        t = b[(int) j];
        t = #protect(t, ms);

        for k = 64 downto 0 {

            _, cf, _, _, _, t = #SHR(t, 1);

            if (cf) {
                ms = #set_msf(cf, ms);

                r = rr;
                r = #protect_ptr(r, ms);

                x = _x;
                x = #protect_ptr(x, ms);

                r = _fp_mulU(r, x);
                _x = x;
                rr = r;
            } else {
                ms = #set_msf(!cf, ms);
            }

            x = _x;
            x = _fp_sqrU(x);
            _x = x;
        }
    }

    r = rr;
    r = #protect_ptr(r, ms);

    return r, ms;
}
```

Listing A.2: `_fp_exp` function with Spectre v1 protections

Bibliography

[1] CompCert – The CompCert C compiler. https://compcert.org/compcert-C.html#perfs. (Accessed on 18/02/2023).

[2] `golang/go`: The Go programming language. https://github.com/golang/go. (Accessed on 14/02/2023).

[3] `jasmin-lang/jasmin`: Language for high-assurance and high-speed cryptography. https://github.com/jasmin-lang/jasmin. (Accessed on 14/02/2023).

[4] Ayush Agarwal, Sioli OConnell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 699–715, 2022. doi:10.1109/SP46214.2022.9833711.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 18071823, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 9781450349468. doi:10.1145/3133956.3134078.

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Gregoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. A Machine-Checked Proof of Security for AWS Key Management Service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 6378, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450367479. doi:10.1145/3319535.3354228.

[7] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 16071622, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450367479. doi:10.1145/3319535.3363211.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, and Tiago Oliveira. Certified Compilation for Cryptography: Extended x86 Instructions and Constant-Time Verification. In *Progress in Cryptology INDOCRYPT 2020: 21st International Conference on Cryptology in India, Bangalore, India, December 1316, 2020, Proceedings*, page 107127, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN: 978-3-030-65276-0. doi:10.1007/978-3-030-65277-7_6.

[10] José Bacelar Almeida, Manuel Barbosa, Manuel L Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. Machine-checked ZKP for NP-relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head. Cryptology ePrint Archive, Paper 2021/1149, 2021. https://eprint.iacr.org/2021/1149. doi:10.1145/3460120.3484771.

[11] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber Part I: Implementation Correctness. Cryptology ePrint Archive, Paper 2023/215, 2023. https://eprint.iacr.org/2023/215.

[12] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Enforcing Fine-Grained Constant-Time Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 8396, New York, NY, USA, 2022. Association for Computing Machinery. ISBN: 9781450394505. doi:10.1145/3548606.3560689.

[13] Utsav Banerjee. *Efficient Algorithms, Protocols and Hardware Architectures for Next-Generation Cryptography in Embedded Systems*. Thesis, Massachusetts Institute of Technology, June 2021.

[14] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795, 2021. doi:10.1109/SP40001.2021.00008.

[15] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. EasyPQC: Verifying Post-Quantum Cryptography. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 2564–2586, New York, NY, USA, 2021. Association for Computing Machinery. ISBN: 9781450384544. doi:10.1145/3460120.3484567.

[16] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and

Pierre-Yves Strub. *EasyCrypt: A Tutorial*, pages 146–166. Springer International Publishing, Cham, 2014. ISBN: 978-3-319-10082-1. doi:10.1007/978-3-319-10082-1_6.

[17] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi:10.1145/3371075.

[18] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-Assurance Cryptography in the Spectre Era. In *2021 IEEE Symposium on Security and Privacy (S&P)*, pages 1884–1901, 2021. doi:10.1109/SP40001.2021.00046.

[19] Daniel J. Bernstein. `qhasm`: tools to help write high-speed software. https://cr.yp.to/qhasm.html. (Accessed on 18/02/2023).

[20] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. miTLS: Verifying Protocol Implementations against Real-World Attacks. *IEEE Security & Privacy*, 14(6):18–25, 2016. doi:10.1109/MSP.2016.123.

[21] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 785800, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450367479. doi:10.1145/3319535.3363194.

[22] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, pages 82–96, 2001. doi:10.1109/CSFW.2001.930138.

[23] Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. You Really Shouldn't Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries, 2021.

[24] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, August 2019. USENIX Association. ISBN: 978-1-939133-06-9.

[25] Chandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html. (Accessed on 21/02/2023).

[26] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 913926, New York, NY, USA, 2020. Association for Computing Machinery. ISBN: 9781450376136. doi:10.1145/3385412.3385970.

[27] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical Foundations for Software Spectre Defenses, 2022.

[28] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A Formal Approach to Secure Speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 288–28815, 2019. doi:10.1109/CSF.2019.00027.

[29] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157, 2019. doi:10.1109/EuroSP.2019.00020.

[30] Łukasz Chmielewski, Pedro Maat Costa Massolino, Jo Vliegen, Lejla Batina, and Nele Mentens. Completing the Complete ECC Formulae with Countermeasures. 7(1), 2017. ISSN: 2079-9268. doi:10.3390/jlpea7010003.

[31] Jean-Sébastien Coron. Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, pages 292–302, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN: 978-3-540-48059-4.

[32] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. Machine-Checked Proofs for Electronic Voting: Privacy and Verifiability for Belenios. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 298–312, 2018. doi:10.1109/CSF.2018.00029.

[33] Craig Costello. A gentle introduction to elliptic curve cryptography. https://static1.squarespace.com/static/5fdbb09f31d71c1227082339/t/5ff376bc0db4f45ccb096682/1609791167623/2019-SACtutorial1.pdf. (Accessed on 29/06/2023).

[34] Formosa Crypto. `libjbn`: BigNums library for Jasmin. https://github.com/formosa-crypto/libjbn, . (Accessed on 14/03/2023).

[35] Formosa Crypto. `libjade`: Crypto library. https://github.com/formosa-crypto/libjade, . (Accessed on 14/03/2023).

[36] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Hunting the Haunter -Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *NDSS 2021 - Network and Distributed Systems Security*, Virtual, France, 2021. doi:10.14722/ndss.2021.24286.

[37] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1): 238268, Feb. 2018. doi:10.13154/tches.v2018.i1.238-268.

[38] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises.

In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019. doi:10.1109/SP.2019.00005.

[39] Denis Firsov and Dominique Unruh. Zero-Knowledge in EasyCrypt. Cryptology ePrint Archive, Paper 2022/926, 2022. https://eprint.iacr.org/2022/926.

[40] Denis Firsov, Tiago Oliveira, and Dominique Unruh. Schnorr protocol in Jasmin. Cryptology ePrint Archive, Paper 2023/752, 2023. https://eprint.iacr.org/2023/752.

[41] Jean-Pierre Flori, Jérôme Plût, Jean-René Reinhard, and Martin Ekerå. Diversity and Transparency for ECC. Cryptology ePrint Archive, Paper 2015/659, 2015. https://eprint.iacr.org/2015/659.

[42] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. 2019.

[43] Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. Swoosh: Practical Lattice-Based Non-Interactive Key Exchange. Cryptology ePrint Archive, Paper 2023/271, 2023. https://eprint.iacr.org/2023/271.

[44] Google. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886. (Accessed on 10/03/2023).

[45] Miguel Grilo, João Campos, João F. Ferreira, José Bacelar Almeida, and Alexandra Mendes. Verified Password Generation from Password Composition Policies. In Maurice H. ter Beek and Rosemary Monahan, editors, *Integrated Formal Methods*, pages 271–288, Cham, 2022. Springer International Publishing. ISBN: 978-3-031-07727-2.

[46] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2020. doi:10.1109/SP40000.2020.00011.

[47] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg, 2003. ISBN: 038795273X.

[48] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level. In *Proceedings of the 14th European Workshop on Systems Security*, EuroSec '21, page 814, New York, NY, USA, 2021. Association for Computing Machinery. ISBN: 9781450383370. doi:10.1145/3447852.3458721.

[49] Mark D. Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L. Hennessy. On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro*, 39(2):9–19, 2019. doi:10.1109/MM.2019.2897677.

[50] Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Formal Verification of Saber's Public-Key Encryption Scheme in EasyCrypt. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 622–653, Cham, 2022. Springer Nature Switzerland. ISBN: 978-3-031-15802-5.

[51] Rémi Hutin. *Compilation vérifiée et sécurisée contre les canaux cachés temporels*. PhD thesis, École normale supérieure de Rennes, 12 2021.

[52] Intel. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html, . (Accessed on 20/04/2023).

[53] Intel. Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html, . (Accessed on 06/03/2023).

[54] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security*, pages 573–582, Cham, 2016. Springer International Publishing. ISBN: 978-3-319-48965-0.

[55] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[56] Neal Koblitz and Alfred Menezes. Critical Perspectives on Provable Security: Fifteen Years of "Another Look" Papers. Cryptology ePrint Archive, Paper 2019/1336, 2019. https://eprint.iacr.org/2019/1336. doi:10.3934/amc.2019034.

[57] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html. (Accessed on 27/03/2023).

[58] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 1–19, 2019. doi:10.1109/SP.2019.00002.

[59] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[60] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.

[61] Xavier Leroy. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL '06, page 4254, New York, NY, USA, 2006. Association for Computing Machinery. ISBN: 1595930272. doi:10.1145/1111037.1111042.

[62] Congmiao Li and Jean-Luc Gaudiot. Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 25–28, 2018. doi:10.1109/CAHPC.2018.8645918.

[63] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 264–276, 2019. doi:10.1109/HPCA.2019.00043.

[64] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[65] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 21092122, New York, NY, USA, 2018. Association for Computing Machinery. ISBN: 9781450356930. doi:10.1145/3243734.3243761.

[66] Paulo Sérgio Alves Martins. Public-key Cryptography on SIMD Mobile Devices. Master's thesis, Instituto Superior Técnico, 2014.

[67] Pedro Maat C. Massolino, Joost Renes, and Lejla Batina. Implementing Complete Formulas on Weierstrass Curves in Hardware. Cryptology ePrint Archive, Paper 2016/1133, 2016. https://eprint.iacr.org/2016/1133.

[68] Victor S. Miller. Use of Elliptic Curves in Cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. ISBN: 978-3-540-39799-1.

[69] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading Kernel Writes From User Space, 2019.

[70] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[71] Nicky Mouha and Christopher Celi. A Vulnerability in Implementations of SHA-3, SHAKE, EdDSA, and Other NIST-Approved Algorithms. Cryptology ePrint Archive, Paper 2023/331, 2023. https://eprint.iacr.org/2023/331.

[72] National Institute of Standards and Technology (NIST). Post-quantum cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-

cryptography-standardization/evaluation-criteria/security-(evaluation-criteria). (Accessed on 10/04/2023).

[73] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498. USENIX Association, August 2020. ISBN: 978-1-939133-17-5.

[74] Tiago Oliveira. *High-speed and High-assurance Cryptographic Software*. PhD thesis, Faculdade de Ciências da Universidade do Porto, 2022.

[75] Zhixin Pan and Prabhat Mishra. Automated Detection of Spectre and Meltdown Attacks Using Explainable Machine Learning. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 24–34, 2021. doi:10.1109/HOST49136.2021.9702278.

[76] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1274, 2019. doi:10.1109/SP.2019.00064.

[77] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. Cryptology ePrint Archive, Paper 2015/1060, 2015. https://eprint.iacr.org/2015/1060.

[78] Peter Schwabe. Presentation. Slide Presentation, December 2022.

[79] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[80] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 279–299. Springer, 2019.

[81] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre Declassified: Reading from the Right Place at the Wrong Time. Cryptology ePrint Archive, Paper 2022/426, 2022. https://eprint.iacr.org/2022/426.

[82] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing High-Speed Cryptography against Spectre v1. Cryptology ePrint Archive, Paper 2022/1270, 2022. https://eprint.iacr.org/2022/1270.

[83] Laurent Simon, David Chisnall, and Ross Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15, 2018. doi:10.1109/EuroSP.2018.00009.

[84] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi:10.1109/SP.2013.13.

[85] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 395410, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450362405. doi:10.1145/3297858.3304060.

[86] The Coq Development Team. The Coq Proof Assistant. https://coq.inria.fr/. (Accessed on 18/04/2023).

[87] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.*, 23(1):3771, jan 2010. ISSN: 0933-2790.

[88] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[89] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019. doi:10.1109/SP.2019.00087.

[90] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi:10.1145/3434330.

[91] Jack Wampler, Ian Martiny, and Eric Wustrow. ExSpectre: Hiding Malware in Speculative Execution. In *NDSS*, 2019.

[92] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 47(11):2504–2519, 2021. doi:10.1109/TSE.2019.2953709.

[93] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman & Hall/CRC, 2 edition, 2008. ISBN: 9781420071467.

[94] Wenjie Xiong and Jakub Szefer. Survey of Transient Execution Attacks and Their Mitigations. *ACM Comput. Surv.*, 54(3), may 2021. ISSN: 0360-0300. doi:10.1145/3442479.

[95] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441, 2018. doi:10.1109/MICRO.2018.00042.

[96] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. Cryptology ePrint Archive, Paper 2014/140, 2014. https://eprint.iacr.org/2014/140.

[97] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 954968, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450369381. doi:10.1145/3352460.3358274.

[98] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. Cryptology ePrint Archive, Paper 2022/715, 2022. https://eprint.iacr.org/2022/715.