High-speed and High-assurance Cryptographic Software

Tiago Filipe Azevedo Oliveira
Programa Doutoral em Informática (MAP-i)
Departamento de Ciência de Computadores
2022

Orientador
Manuel Bernardo Martins Barbosa, Professor Associado, Faculdade de Ciências da Universidade do Porto

Coorientador
José Carlos Bacelar Ferreira Junqueira Almeida, Professor Auxiliar, Departamento de Informática da Universidade do Minho

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

Dedication and acknowledgements

First of all, I want to express my gratitude to the supervisor and co-supervisor of this thesis, Manuel Barbosa and José Almeida, first, for convincing me to embark on this fascinating journey that was doing this Ph.D. and, second, for all the assistance and dedication into creating what turned out to be the ideal conditions for this work to happen the way it did.

Words of appreciation are also due to the whole Jasmin team, with a particular remark going to Benjamin Grégoire for the fantastic collaboration and countless hours we've spent together since this project started. I would also like to thank all my co-authors and research colleges for their support.

I would like to thank Peter Schwabe for all the support, motivation, and guidance always expressed every day without exception long before I joined the Max Planck Institute for Security and Privacy. A note of gratitude is also due to Gilles Barthe, whose observations, comments, and advice, allow me to improve.

A warm thanks to my family and friends for all the support given throughout these years.

Finally, the most important acknowledgment is to my beloved wife Verónica, that unconditionally supported me and whose advice always revealed itself to point in the right direction. This thesis is dedicated to you Verónica[1].

[1] I will spend more time with you from now on. Throw this to my face if I don't :-)

Abstract

The implementation of cryptographic primitives is a complex task that requires a very particular skill set. Several cryptography-specific requirements must be fulfilled, with these being mainly related to efficiency and security. Since cryptographic code is being used at large, a mere performance penalty of 10%, in comparison to the fastest implementations, can impede its adoption even if advantageous safety guarantees are provided. For this reason, cryptography is often implemented using low-level programming languages such as assembly, qhasm, and C, with some implementations taking advantage of specific CPU extensions to improve performance. The produced machine code should also be provably functionally correct and protected against side-channel attacks to ensure that there are no unexpected behaviors, such as an overflow occurring under certain conditions or variances in the execution time that could cause a secret data leakage.

In this thesis, we first explore how formally verified C compilers impact the efficiency of cryptographic libraries. This first study concludes that a low-level programming language and corresponding formally verified compiler are required to achieve the desired properties. In this context, we present, from a practitioner point-of-view, the Jasmin programming language, which is part of the Jasmin framework. The Jasmin programming language provides a high level of control over the resulting assembly code, which is required to produce high-speed implementations comparable with the fastest hand-written assembly implementations. The thesis concludes with an example base validation of the Jasmin framework. We cover several cryptographic primitives, showing that their Jasmin implementations match the performance of the best unverified code. We also discuss how these implementations were formally verified.

Resumo

A implementação de primitivas criptográficas é uma tarefa que requer conhecimento e proficiência em diversas áreas dado que estas devem cumprir diversos requisitos relacionados com eficiência e segurança. Sobre a eficiência, e considerando que o código criptográfico é utilizado em grande escala, um decréscimo de performance na ordem dos 10% de uma dada implementação, por exemplo, pode ser um factor que impossibilite a adopção desta por parte dos seus utilizadores, mesmo no caso em que garantias adicionais de segurança sejam providenciadas. Em consequência desta necessidade por elevado desempenho, a implementação de código criptográfico é realizada com recurso a linguagens de programação com baixo nível de abstracção, tais como assembly, qhasm ou C, recorrendo também a extensões específicas de certas arquitecturas de computador. A par dos apertados requisitos de eficiência encontram-se os requisitos de segurança, no sentido que, idealmente, todo o código máquina que origina de implementações criptográficas deveria ser formalmente verificado por forma a garantir, por exemplo, a correcção e a ausência de variabilidades no tempo de execução dependentes de parâmetros secretos.

Neste contexto, o da conciliação de implementações de elevado desempenho com metodologias de verificação formal, foi realizada esta tese. Inicia por apresentar uma visão global sobre as metodologias, ferramentas e necessidades inerentes ao desenvolvimento de código criptográfico e de que forma uma possível solução baseada em compiladores de código C, formalmente verificados, impacta a performance quando comparada com as melhores alternativas não verificadas. Consequentemente, justifica-se a necessidade de uma abordagem alternativa que permita replicar, ou suplantar, os melhores resultados de eficiência já obtidos em contextos não formais. Apresenta-se então a linguagem de programação Jasmin, sendo esta apresentação particularmente orientada para programadores versados em linguagens de programação com diminuta abstração. As implementações criptográficas desenvolvidas no contexto desta tese são de seguida discutidas estando incluídas as primitivas ChaCha20, Poly1305 e Curve25519. A *framework* Jasmin inclui uma infraestrutura que permite a verificação formal das técnicas desenvolvidas, sendo esta baseada em EasyCrypt. As provas de correcção dos algoritmos apresentados são também abordadas.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cryptography is a multi-disciplinary area with the ultimate goal of developing primitives, schemes, and protocols to build secure systems. Mathematics, computer science, software, and electrical engineering are vital disciplines involved in designing, validating, and implementing cryptographic constructions. To be accepted by the community and its potential users, and eventually get standardized, two essential requirements must be satisfied: security and efficiency. Formal methods can play an important role in obtaining this assurance. However, one particular issue remains to be solved: a sound connection between formal verification methods and the fastest cryptographic implementations (usually written using low-level programming languages) that can be used to relate high-level specifications with the actual machine code that gets executed. Ideally, this connection between formal methods and machine code should not be limited by the complexity of the implementations: it should be possible to prove the correctness of such implementations and also to prove the existence of any specific properties that are relevant in cryptography domain, such as the code being constant-time under some model of execution. As such, this thesis aims to improve the state-of-the-art in this area: the development and formal verification of high-speed cryptographic implementations.

The introduction follows with some discussions related to high-speed, high-assurance cryptographic implementations, which expose the requirements that justify the need for a new approach. Section 1.1 focuses on the standardization of cryptographic components. One of the criteria used to guide the standardization process, besides security claims, corresponding proofs, and the absence of significant attacks against a particular cryptographic construction, is the ability of a given component to be implemented and executed efficiently in a wide range of architectures. From a candidate's point of view, having formal guarantees in the submitted code can increase the candidate's chances of being selected, given that low-level code is not easily auditable and it may contain subtle defects. Security proofs should also be strongly tied with the evaluated machine code. An interested reader might start by consulting the

extended material on the subject, on Appendixes A.1 and A.2, which presents a more detailed overview of two standardization processes, DES and AES, as additional motivation.

The deployment of non-optimized code can have a non-negligible economical impact. Additionally, it is also not trivial to systematically measure the performance of cryptographic software. Because of this, it is important that the development of new methodologies and tools that bridge the gap between high-speed and high-assurance should allow the practitioner to evaluate the performance of both verified and non-verified implementations. Section 1.2 presents some considerations on this matter.

Section 1.3 contains an overview of the current approaches used to implement high-speed cryptographic software. It then discusses how alternative approaches, some recently published, conciliate formal methods with performance requirements.

The previous section establishes context for our contributions, discussed in section 1.4, along with the list of publications directly related to this document and how they relate to the thesis' structure.

1.1 Cryptography Standards

In 1973, the National Bureau of Standards (NBS), known as the National Institute of Standards and Technology (NIST) since 1988, published the first solicitation of proposals for an algorithm that could be standardized and used to protect computer data in transmission or storage, mainly motivated by the increasing amount of digital communication [oS73]. This solicitation initiated the process that led to the Data Encryption Standard (DES), an encryption algorithm used until 1999 in its original form [oST99a, pg.4 item 12]. One of the main problems of DES was the secret-key length, which had only 56 bits and, because of this, at the end of its life-cycle DES could be easily brute-forced [Cur05]. Even at the time of the standard's publication, in 1977, some cryptographers presented convincing arguments to justify why 56-bit secret keys did not provide enough security [DH77]. Initially, only hardware deployments of DES were allowed, and, in 1993, software implementations were permitted [oST93, pg. 3, item 12]. Many implementation concerns that exist today did not exist back then, and the most valuable lesson that can be learned from this event is that the development of new cryptographic standards should be as open to the community as possible. Appendix A.1 presents a more detailed overview of these events.

In 1997, NIST announced the beginning of the Advanced Encryption Standard (AES) standardization process to replace the 20-year old DES [oST97a]. In the submissions call, the evaluation criteria covered the following topics: *Security*, *Cost*, and *Algorithm and Implementations Characteristics*. The *Cost* section enumerated some properties that would be evaluated, such as speed (for instance, the number of CPU cycles taken to perform a

given operation or throughput), memory requirements, and implementation complexity and flexibility. The AES standardization process was organized into two rounds, each focusing on some particular properties of the submissions. In the first round's documentation, it is possible to observe some inconsistencies in the evaluation process. Appendix A.2 describes these in more detail. Measuring and comparing software is an intricate task, sometimes seen as reasonably simple, whose minor or overlooked details often affect results. The comparison complexity significantly increases when evaluation must be done on multiple platforms and using different environments. This observation is still valid. The constant-time policy, where cryptographic code should not branch or, more generically, perform any variable-timing operation (including memory accesses), that depend on secret data was also not correctly addressed [Ber05a, TOS10]. From the five finalists, Rijndael (selected for standardization) and Serpent were the top two candidates in a survey conducted by NIST during the last AES conference. NIST selected only one algorithm. Rijndael, as it was evaluated, was consistently faster than Serpent.

In early 2007, NIST announced the development of new algorithms to be included in the Secure Hash Standard [oST07]. This standardization process, referred to as SHA-3 Project on NIST's website, was a response to the published attacks on SHA-1. The "*Proposed Draft Submission Requirements*" section from this announcement was very similar to the one previously used for AES, but it included an additional mention to security proofs: "*(...) any security argument that is applicable, such as a security reduction proof;*". There is also a call for an "*ANSI C (...) reference implementation and an optimized implementation*". In the equivalent announcement for AES, it is stated that the ANSI C submitted implementation will "*be used to compare software performance and memory requirements with respect to other algorithms*". This update approximates the development practices of the cryptography community with the standardization process: security proofs are used to reason about specific properties of the proposed designs, and multiple implementations for the same primitive, targeting multiple CPU architectures and architectures extensions, are also commonly used in practice to use as most as possible the available resources. The number of candidates also grew: AES had 15 submissions considered valid (i.e., submissions that met all requirements) and SHA-3 had 51 submissions in the same condition. Although these standardization processes relate to different types of cryptographic operations, it can be assumed that the interest and motivation from the community grew as well between these ten years that separate the beginning of AES and SHA-3. The SHA-3 standardization process had three rounds in total. The Round 3 final report contains an extensive comparison between all five finalists: this comparison uses data from the ECRYPT Benchmarking of Cryptographic System (eBACS) [BL], to study the performance of the candidates on multiple CPU architectures and for different message lengths [CPB+12]. In 2012, the winner of SHA-3 competition was announced: Keccak. Although the winner did not deliver the best performance in software, it did excel on its hardware implementation capabilities. One aspect that also favored this decision was the

design's flexibility and fundamentally different design principles compared to SHA-2. From this event, it is possible to observe that performance is not the only critical factor. In an ideal scenario, security proofs and specifications should be soundly related to the machine code that is used in real-world deployments.

1.2 Efficiency and Cost

Efficiency is an essential property of any cryptographic implementation, and it is primarily defined by design choices, for instance, the number of rounds of a cipher. Secondly, cryptographic implementations can perform poorly when inappropriate languages or compilers are used, and, even for similar evaluation environments, significant performance differences can be observed. Considering that cryptography is practically omnipresent in modern systems and that most applications use a limited set of primitives and protocols, it makes sense to optimize the cryptographic implementations that are commonly used. It also seems reasonable to assume that even a relatively modest performance improvement in a CPU-intensive cryptographic implementation deployed worldwide over countless servers can directly translate into the improvement of the response times of such servers and contribute to reducing server costs.

In a Cloudflare blog post from 2017 [Kra17], Krasnov presented a study that detailed how much CPU time is spent by cryptography-related computations in one of Cloudflare's servers. The profiled server was located in a data center in Germany, where, and at that time, 73% of the requests were performed using TLS[1]. The used cryptographic library was BoringSSL, "*a fork of OpenSSL that is designed to meet Google's needs*", as it can be read on the project's GitHub page [bor21]. It is stated in this study that only 1.8% of CPU time was spent doing cryptographic operations. More specifically, Curve25519 and ChaCha20-Poly1305 consumed 0.06% and 0.03%, respectively, of the total CPU time. For context, 30.5% of the TLS requests performed the key exchange using Curve25519.

In a study published in 2020 [MSL+20], it is estimated that, in 2018, data centers represented 1% of the global energy consumption, which corresponds to roughly 205 TWh. If we assume an average cost of $0.10 per KWh, for simplicity, then data centers' energy consumption in 2018 corresponded to $20.5 billions. Depending on the overall efficiency of the data centers facilities, for instance, the efficiency of the cooling systems, servers can represent 50% to 70% of the total energy consumption of the data center [KBK12, MSL+20]. If we also assume that 40% of a server's energy consumption is directly related to the CPU, then 20% to 28% of the $20.5 billion is spent by CPUs. If one-third of these servers are performing TLS connections with a workload similar to the profiled Cloudflare's server, then one can estimate that roughly $25 million were spent during 2018 doing cryptographic operations in data centers. To

[1] According to Google Transparency Report, 95% of the traffic was encrypted in early of 2022.

conclude, the discussion from this paragraph should be taken lightly, as it contains a high number of assumptions, and several factors are ignored. Nonetheless, it serves its purpose by showing that, even with what can be considered conservative assumptions, cryptographic implementations can be directly associated with non-negligible costs (depending on the view). On the other hand, estimating the negative economic impact caused by a security breach related to some implementation defect is even more challenging. However, it is easy to imagine several scenarios where this cost is orders of magnitude greater than the cost of running cryptographic software. For this reason, cryptographic implementations should be as high-assurance as possible, and performance should not be improved at the expense of security.

1.3 Cryptography Implementations

As it was mentioned during the SHA-3 discussion, from section 1.1, there can be different implementations of same cryptographic primitive for performance reasons. After prototyping, which can be done using tools/programming languages with a higher level of abstraction, such as Sage or Python, each primitive is usually implemented using the C programming language. This first implementation is often called *reference* implementation and is frequently written as close as possible to the mathematical description of the algorithm. Readability is usually the primary concern when developing reference implementations. The goal is to provide an accurate description of the algorithm that can simultaneously be compiled for different CPU architectures while being reasonably efficient and used as a basis for developing optimized versions of the algorithm.

The next step is to produce optimized versions of the primitive. Such optimizations can be generic, meaning that the developed code still compiles in a wide range of architectures, or they can be architecture or micro-architecture dependent. In the latter case, where the code is non-portable, and for this discussion, three different approaches can be taken: 1) the implementation is still written in C, but it uses some low-level machine instructions that are only available in some architectures, and, as such, the produced code is non-portable. Such low-level instructions can be accessed using intrinsics, which, essentially, are functions that the compiler handles differently and usually correspond to specific machine instructions; 2) the usage of C compilers is avoided by writing the implementation directly in low level-languages, such as assembly or qhasm; 3) the implementation is written using a mixture of C and assembly, which can be inlined in the C code or be independently defined and be compatible with the target calling convention. In these contexts, it is pretty common to find optimized implementations that use instructions that simultaneously process multiple pieces of data. These are often referred to as vectorized implementations.

To provide an initial intuition on vectorization: some CPU extensions specify an additional

set of registers and corresponding instructions that allows the processing of multiple values at once, often referred to as SIMD (Single Instruction, Multiple Data). For example, with Intel AVX2 (Advanced Vector Extensions, version 2), it is possible to use 256-bit registers that can be used to perform, for instance, some arithmetic operations on four 64-bit or eight 32-bit values using only one instruction. Many other sub-sizes and types of instructions (bitwise or for reorganizing the internal state of the registers) are available. When compared to non vectorized code, that in a 64-bit CPU can use up to 64 bits of the available set of *native* registers, the performance can be improved up to 4 times, depending on the algorithm's complexity and many other aspects.

It is generally true that optimized implementations are more prone to contain bugs than the corresponding reference implementation due to increased complexity. It is also common practice to use test vectors to check if an implementation performs a given computation as expected. Intuitively, a test vector can be seen as a map between inputs and corresponding outputs: the implementation is given each of these inputs to check if it produces the expected outputs. For primitives that require randomness, the standard approach for testing is to replace the function that produces the *real* randomness with one that produces a predefined stream of data, for instance, by encrypting zeros with a fixed key and nonce. Although it may seem intuitive to consider that using a test-based approach to reason about an implementation's correctness is more than sufficient in most cases, that is not usually true. One can always argue that if test vectors are well-chosen, they should cover all different scenarios and cause all types of behaviors to occur. That is, however, very difficult (if not impossible) to achieve, for instance, in primitives that require computations on big-numbers. At the core of the trust base of the described development model are: the developer, who writes and optimizes the code; the tester, which designs the test vectors; and the compiler, if the implementation needs to be compiled.

In addition to functional correctness and memory safety, to ensure that the program behaves as expected and it does not read or write from invalid memory space, side-channel attacks should also be considered during the development of cryptographic implementations, and corresponding countermeasures should be implemented. For instance, the execution time of cryptographic implementations should not depend on secret data. Otherwise, an attacker can exploit this behavior to acquire information about it. Implementations whose execution time does not depend on secret data are often called constant-time implementations. An attack that takes advantage of timing variations is a timing attack. If we assume that the execution time of the arithmetic instructions of a given architecture does not depend on the corresponding input data, for instance, a given multiplication instruction always takes the same execution time even if both inputs are zero, then it should suffice for the control-flow and memory addresses to be independent of secret data. Nonetheless, any methodology and tools produced for constant-time verification should be designed to allow for more general leakage models and be flexible enough to capture different architectures.

Overall, formal methods can be employed to tackle these problems: the ideal scenario would be to have formal guarantees on the machine code corresponding to a given cryptographic implementation that performs as efficiently as possible. The following subsections describe some tools, methodologies, and approaches that allow implementing high-speed and/or high-assurance cryptographic software. This establishes a basis for comparison with our approach, presented in section 1.4.

qhasm

qhasm is a set of tools that allow the writing of high-speed software [Ber07]. The latest version is from 2007. qhasm uses machine-description files to define the programming language for a given target architecture. This means that it can be easily extended for different needs. For example, the machine-description files define the available types, and, if applicable, to which registers do they map, and the supported set of instructions. For instance, in AMD64, the qhasm instruction "`r += s`" directly translates to an `add` assembly instruction where `r` is a register and `s` may be a register or a memory operand. In the available version of the tool, the configuration file needs to have two entries specifying "`r += s`" for the case where `s` is a register and for the case where `s` is a stack variable. If a developer writes a program that, in a given segment, contains more live register variables than the registers that are available, then the program does not compile. Nevertheless, control-flow structures are typically implemented using explicit `goto`-like instructions to a given label. Control-flow structures are implemented by the qhasm developers using well-defined patterns, which allows for automatic translation to other low-level programming languages [Oli12, ABB+17a].

The intuition of qhasm is that each statement (one line) corresponds to a predefined set of assembly instructions (usually one). The compiler inserts no unexpected instructions during the compilation of a qhasm program to assembly. The main advantage of using qhasm, compared to directly programming in assembly, is that the developer does not need to perform the register allocation and manage the stack variables manually. Also an advantage, is the usage of a syntax closer to the C programming language, for instance. qhasm has been used to implement numerous speed-record-breaking cryptographic implementations [Ber06a, BS08, CS09, NNS10, BCC+12, BCS13, Cho15].

In a work published in 2014 [CHL+14], a new approach to achieve formally verification of qhasm written programs was presented. The verification was performed at the source level and, as the paper clearly mentions it, "*The obvious disadvantage is that we rely on the correctness of qhasm translation*" (to assembly). Given an annotated qhasm file, a Boolector (an SMT-solver [BB09]) specification is produced, and most properties are automatically checked. For the goals that Boolector fails to check, the Coq proof assistant [BC13] was used. The methodology was validated using two different implementations (targeting different micro-architectures that used different internal representations/radix) of the Curve25519 [Ber06b].

Vale

In 2017, Vale, a tool/programming language that enables the automatic verification of high-speed cryptographic assembly code, was presented [BHK+17]. Vale tool transforms a given annotated assembly implementation into an abstract syntax tree (AST) and generates the corresponding proof goals to be verified by an SMT solver. One of the main motivations of this work was to provide a viable alternative (in the sense that no computational overhead is introduced) to the development methodologies that are used in several cryptographic implementations of OpenSSL, which are described by the authors as not to be easily analyzed. As a quick overview, instead of writing the assembly in its final form, OpenSSL developers leverage the text processing capabilities of Perl, together with preprocessing macros, to write slightly more generic assembly code. For instance, instead of using registers by their names (rax,r10,r11,...), a *renaming* mechanism allows for intuitive variable names. This development approach can be seen as an extended macro mechanism. There is no automatic register/stack allocation in OpenSSL's approach as it is done in qhasm.

Vale is built upon Dafny[Lei10], an automatic program verifier, and it was validated with optimized and architecture-dependent (ARM and AMD64) implementations of SHA-256, Poly1305, and AES-CBC. The conclusions from the presented evaluation are clear: Vale produces code whose performance matches OpenSSL's best implementations (hand-written assembly, also commonly referred to as Perlasm) and, in some specific contexts, surpasses the throughput of some of those implementations, for instance, a complete Vale implementation of Poly1305 (assembly) versus a mixture of C and assembly as it is reportedly implemented in OpenSSL. One of the main advantages of Vale is that functional correctness and memory safety are checked at, essentially, the machine level, which means that it is not necessary to rely on non-formally verified compilers to achieve competitive levels of performance.

In 2019, Vale was extended via an embedding of AMD64 assembly in F* [FGH+19]. F* is a functional programming language with native support for semi-automatic verification that relies on SMT solvers. By using this approach, the authors verified, for the first time, an optimized implementation of AES-GCM. One of the advantages of this solution is that it enables the practitioner to write highly-efficient code for the performance-critical cryptographic routines, which can be easily integrated with HACL*.

HACL* and HACLxN

HACL*, a verified cryptographic library used on Mozilla's NSS project, was presented in 2017 [ZBPB17]. The distributed code is functionally correct, memory safe, and includes mitigations to protect against timing attacks. The solution relies on several tools and different techniques. Specifications are written using a higher-order functional subset of F*. Optimized code is implemented using Low*, a low-level subset of F*. The optimized

implementations (Low*) are verified to match the corresponding high-level F* specifications. The verification process is semi-automatic and leverages the capabilities of the Z3 SMT solver [MB08]. Low* implementations are compiled to C using the KreMLin tool [PZR+17], and C implementations can be compiled to machine code using CompCert [Ler09], a formally verified compiler. Alternatively, Clang or GCC can be used if performance is a concern. HACL* provides implementations whose performance is competitive with alternative, non-formally-verified, C implementations. The approach was validated with ChaCha20 [B+08], Poly1305 [Ber05b], Curve25519 [Ber06b], Ed25519 [BDL+12a], and some other examples. This work also provided the first verified vectorized implementation. A comprehensive performance evaluation of the resulting source code is presented, and includes different architectures and compilers. There are ongoing efforts to reduce the number of components under the trusted computing base.

In 2020, HACL* was extended to support systematic C extraction of vectorized code for different architectures and multiple CPU extensions [PBP+20]. The approach was validated with ChaCha20, Poly1305, SHA-256, and Blake2, the second version of a final candidate from the SHA-3 standardization process. The results of this extension to HACL* cannot be compiled with CompCert, due to the lack of support for vectorization instructions on this formally verified compiler.

Fiat Cryptography

In 2019, the project Fiat Cryptography was presented [EPG+19b]. It defines a new approach to building a formally-verified C library of arithmetic operations commonly used in the cryptography domain, mainly focused on elliptic curve computations. The approach is instantiated with a Coq-verified compiler that takes high-level specifications and outputs the corresponding optimized C implementations. The results of this project have been integrated into BoringSSL, a publicly available cryptography library from Google. When compiled with a generic C compiler, the Fiat Cryptography implementations' performance is comparable with hand-written assembly and faster than equivalent computations implemented using GMP, a generic multiple precision arithmetic library. Hence, the trusted computing base is only composed of the C compiler and a simple pretty printer. A clear advantage of this approach is the ability to generate optimized C implementations for different prime fields very simply. Additionally, it is mentioned that there is an interest in further reducing the trust base by removing the C compiler from it.

1.4 Contributions, Sources, and Structure

This section contextualizes and describes the contributions of this thesis, chapter by chapter, stating the related publications, where applicable. Almost all writing in this thesis is entirely novel, except for chapter 3, which includes material from the corresponding publication.

Chapter 2

The evaluation of cryptographic software's performance is critical for several reasons. Cryptographic primitives are used at scale, and introducing non-negligible performance overheads can be unacceptable. Without exception, all the previously discussed contexts and works, from the establishment of cryptography standards to the different approaches to implementing high-assurance cryptographic routines, demonstrate a big concern with the resulting software's performance. Although evaluation methodologies seem to be an already solved problem, that is sometimes not the case, considering the number of debates that frequently arise around this topic. For that reason, chapter 2 presents the methodologies used to validate our approach.

Chapter 3

Generic C compilers are commonly found in the trust base of many solutions which rely on them to produce code whose performance is sometimes comparable to hand-written assembly. As was previously discussed, one way of improving the performance of cryptographic implementations is to take advantage of specific CPU extensions that allow for vectorization. CompCert, a formally verified C compiler, does not include support for those extensions, and as such, it is not possible to take advantage of the correctness proof of this compiler, which provably preserves the semantics of the input programs until the last step of the compilation, assembly, in speed-critical contexts.

In this context, in 2017, we concluded the validation of a methodology that allowed to add support for vectorized instructions on formally verified C compilers. Our approach was instantiated with CompCert, which, at the beginning of this research project, was in version 2.2. The available CompCert did not include support for AMD64, so we extended and evaluated the x86 (32-bit) backend. This work was later published in 2020 [ABB+20b]. The conclusions from this study point to the need for a different approach to ultimately bridge the gap between high-speed and high-assurance implementations. The following publication resulted from this work.

[ABB+20b] Certified Compilation for Cryptography: Extended x86 Instructions and Constant-Time Verification

(Indocrypt 2020)

This paper presents an extension to the CompCert compiler to support instruction extensions in the x86 architecture. This work also describes a type system for constant-time verification. The extended CompCert was evaluated within the SUPERCOP toolkit.

Chapters 4 and 5

Several requirements related to cryptography development have been presented until now. A set of methodologies and tools to facilitate the deployment of high-quality cryptographic code must: allow replicating the performance (and overall behavior) of non-formally verified alternatives such that the final user does not incur in extra-costs by using formally-verified implementations; it must allow checking the correctness, memory safety, and absence of leakages under different models of execution of any implementation; these properties should be propagated to the produced assembly or machine code; and, to conclude, the level of control that is required to achieve the best performance should not implicate extra complexity in the development tasks. For instance, it is not sustainable for a developer to implement all components that a vast cryptography routine requires in pure assembly (and this is why, very frequently, and for complex routines, a mixed approach that uses C and assembly is used).

Jasmin was developed to address these requirements. Jasmin is a framework that includes a programming language, a Coq-verified compiler that compiles Jasmin source code to assembly. The compiler preserves the semantics of the input programs and the constant-time property. The compiler currently supports the AMD64 architecture, AVX/AVX2, and some additional extensions such as ADX/BMI2. ARM will be the next supported architecture, and it is a work in progress.

For the source-level formal verification tasks, we rely on an embedding of Jasmin to EasyCrypt (the translation from Jasmin programs to EasyCrypt modules is essentially a one-to-one correspondence and is currently in the trusted computing base) [BDG+13]. Three extraction models are supported: functional verification, verification of the constant-time property, and memory safety verification. The idea behind the functional verification is the following, a given specification of a cryptographic primitive (that can be written using a functional-programming style, for instance) is proven equivalent to the extracted EasyCrypt representation of a given Jasmin program. The compiler also includes an automatic safety-checker and a constant-time type-checking system.

As previously mentioned, a cryptographic primitive can have several different implementa-

tions, some vectorized. All implementations correspond to the same specification. Consider the following scenario where it exists: an EasyCrypt specification, a reference implementation, and a vectorized implementation. Depending on the primitive characteristics, the specification can be proven equivalent to the reference implementation, and the reference implementation can be proven equivalent to the vectorized implementation. Alternatively, it might be easy to prove the specification equivalent to each implementation if, for instance, the vectorization techniques used introduce significant differences to the reference implementation. Also an advantage of following this approach is that security proofs developed in EasyCrypt, can be related with practical implementations.

Jasmin is a low-level programming language in the sense that an experienced developer knows exactly the assembly instructions that the compiler generates during compilation. In Jasmin, it is the developer's responsibility to decide which variables are placed in registers or memory. This approach is similar to the one taken by qhasm. On the other hand, Jasmin includes several *high-level* features such as function calls, control flow statements, and arrays.

Chapter 4 presents a comprehensive review of the language, and its features, from a practical point of view. There is no available documentation for this programming language, and this chapter aims to cover precisely that. The corresponding publication, where the Jasmin framework was presented, is detailed next [ABB+17a]. The first approach for the formal verification tasks of the compiler was updated, and the previous one is no longer used and was replaced by the EasyCrypt extraction.

[ABB+17a] Jasmin: High-Assurance and High-Speed Cryptography

(CCS 2017)

> This paper describes the Jasmin framework in its early stages. This work briefly presents the Jasmin programming language, and it provides some insight into how the compiler works. It discusses the proofs methodologies for the compiler and how constant-time preservation is achieved. The safety and constant-time analysis were done through an embedding of the Jasmin language into Dafny[Lei10]. The motivating example was Curve25519, and the performance evaluation of the implementations was done using SUPERCOP. Many other cryptographic implementations were considered to validate the proposed approach and corresponding tool. Most of these were obtained by automatically translating existing qhasm code to Jasmin [Oli12], to create a significant code base quickly.

The first version of Jasmin supported the AMD64 architecture, but it did not include support for vectorization instructions. The second Jasmin publication explains how the tools were extended, and how the verification infrastructure was improved [ABB+20a]. It included state-of-the-art implementations with competitive (for some input lengths, slightly better) than all available implementations. They were formally verified using the new approach,

which relied on EasyCrypt. The discussion on Chapter 4 covers vectorization.

[ABB+20a] The Last Mile: High-Assurance and High-speed Cryptographic Implementations

(S&P 2020)

This paper describes an improvement to the Jasmin framework that allows the development of cryptographic implementations that perform as efficiently as hand-written assembly code. To achieve this, the language and the compiler were extended to support new types and instructions to allow for vectorized implementations. The framework was also extended with an embedding into the EasyCrypt framework to allow for functional correctness and constant-time verification. The approach was validated by implementing and verifying ChaCha20 and Poly1305, two cryptographic algorithms used in one of the two ciphersuites recommended in TLS 1.3.

Chapter 5 presents, in detail, the design and optimization process of several different implementations of ChaCha20 and Poly1305. The work described in this chapter defined the shape of the Jasmin programming language and its compiler by clearly setting the goals: only the best-performing code should be considered a viable option to demonstrate the quality of our approach. In addition to the mentioned implementations, this chapter describes a novel implementation of the Curve25519, which has outstanding performance. All implementations are extensively evaluated and compared with alternative implementations, formally-verified or not, with many of them being provided by the previously discussed projects such as HACL*, Vale, or Fiat Cryptography. Regarding the formal verification, in the context of ChaCha20 and Poly1305, an overview of the proof methodology is presented. For Curve25519, the current status of the proof (which is a work in progress) is comprehensively detailed.

An additional paper was published in this context: it demonstrates how the Jasmin framework can be used to relate machine-checked security proofs with optimized implementations. This work is not included for extended discussion in this document.

[ABB+19] Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3

(CCS 2019)

This paper presents a high-assurance and high-speed implementation of the SHA-3 hash function. Two Jasmin high-speed implementations were developed, one non-vectorized and another that takes advantage of vectorization instructions. These were formally verified for functional correctness, timing attack resistance, and provable security using the EasyCrypt proof assistant.

Chapter 6

Chapter 6 is dedicated to providing an overview, in hindsight, of the developed approaches and how they can be extended. It overviews the impact of Jasmin in the community and sets a series of challenges for future work.

Chapter 2

Evaluation of Cryptographic Software

Developing new methodologies and tools to produce high-speed and high-assurance cryptography software must be accompanied by reliable evaluation methodologies. The number of CPU cycles required to perform a computation on a given CPU architecture, or sometimes micro-architecture, is frequently used to compare alternative approaches. Depending on the target environment, there could be other concerns besides speed, such as stack-frame space or binary size. The latter considerations are especially valid in heavily restricted contexts, such as IoT applications where the devices have limited resources. Currently, we are mainly interested in contexts where memory is not a practical limitation and, as such, all the discussions throughout this chapter are CPU-time oriented.

We use the SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives) toolkit to conduct our analysis. SUPERCOP can be downloaded on the project's web page[1]. The toolkit is actively maintained and includes thousands of cryptographic implementations and a building system that produces a cryptographic library containing the implementations that perform the best in a given environment. During the build, which can take several days to a couple of weeks depending on the target's hardware, implementations are tested and measured. Information about each step of the building, including testing and measurement results, is kept in a log file. This log, which contains some other environment information such as the CPU model and corresponding frequency being used, can be submitted through the eBATS mailing list for the results to be later published on the SUPERCOP website, making the collected data available for public consultation.

Section 2.1 discusses the SUPERCOP toolkit in more detail. Section 2.2 presents an evaluation methodology for cryptographic implementations.

[1] `https://bench.cr.yp.to/supercop.html`

2.1 SUPERCOP

SUPERCOP allows building a cryptographic library targeted for a given environment. It is also commonly used to measure the performance of cryptographic implementations. This section starts by discussing how implementations are organized within SUPERCOP. The discussion follows with an overview of the setup process and how the building system works to establish ground knowledge for section 2.2.

2.1.1 Operations, Primitives and Implementations

SUPERCOP's implementations are organized in three hierarchical levels: operations, primitives, and implementations.

At the top-level directory of the toolkit, a file named OPERATIONS contains a list of all cryptographic operations supported by the toolkit. Each operation is essentially an abstract interface for a cryptographic component. For instance, crypto_hash and crypto_stream are two operations that specify the API for hash functions and stream ciphers, respectively. Each operation has a corresponding directory at the top-level directory of the toolkit. These contain primitives, which are instantiations of their parent operation. sha256 and chacha20 (primitives) implement crypto_hash and crypto_stream (operations). Each primitive can have several implementations. Some of these implementations are platform-independent (frequently named ref and written in C), while others target specific architectures or CPU extensions. Platform-dependent implementations can: 1) be written in C and use intrinsics to access platform-specific data types and instructions; or 2) written in assembly, qhasm, or similar.

Table 2.1 presents the API for 7 out of the 19 operations included in SUPERCOP version 20210604. SUPERCOP builds the operations' primitives in the same order as they appear in OPERATIONS. For example, crypto_hashblocks primitives are sometimes used by the corresponding crypto_hash primitive (sha256), and, as such, the former is processed first. Table 2.1 also includes the set of MACROS that each implementation must define in a file named api.h. The api.h file must be defined for SUPERCOP to consider a given implementation in the evaluation process. For brevity, "unsigned char" is written as "u8", "const unsigned char" as "cu8", and "unsigned long long" as "ull". The following paragraphs provide a small overview of each operation from Table 2.1.

crypto_verify This operation specifies an interface to verify (in constant-time) if two given arrays x and y are equal. The length of the arrays is statically known, and the BYTES macro specifies it. Supporting different lengths requires different primitives. SUPERCOP has several primitives for this operation, including the following three: 32; 16; and 8. In this

Operation	API	MACROS
crypto_verify	int crypto_verify(cu8 *x, cu8 *y);	BYTES
crypto_core	int crypto_core(u8 *out, cu8 *in, cu8 *k, cu8 *c);	OUTPUTBYTES INPUTBYTES KEYBYTES CONSTBYTES
crypto_hashblocks	int crypto_hashblocks(u8 *statebytes, cu8 *in, ull inlen);	STATEBYTES BLOCKBYTES
crypto_hash	int crypto_hash(u8 *out, cu8 *in, ull inlen);	BYTES
crypto_stream	int crypto_stream(u8 *c, ull clen, cu8 *n, cu8 *k); int crypto_stream_xor(u8 *c, cu8 *in, ull inlen, cu8 *n, cu8 *k);	NONCEBYTES KEYBYTES
crypto_onetimeauth	int crypto_onetimeauth(u8 *out, cu8 *in, ull inlen, cu8 *k); int crypto_onetimeauth_verify(cu8 *h, cu8 *in, ull inlen, cu8 *k);	BYTES KEYBYTES
crypto_scalarmult	int crypto_scalarmult(u8 *q, cu8 *n, cu8 *p); int crypto_scalarmult_base(u8 *q, cu8 *n);	BYTES SCALARBYTES

Table 2.1: SUPERCOP API overview.

case, the name of the primitive refers to the size in bytes of both arrays. An implementation of this operation is expected to return 0 if inputs are equal or -1 otherwise.

crypto_core This is an interface for *core* cryptographic components. It expects four arguments, out for the output, in for the input, k for a key, and c for additional inputs. All arguments have statically known lengths, defined by the corresponding macros. Example primitives are aes256encrypt and salsa20: the implementations of aes256encrypt do not use c and, as such, CONSTBYTES is defined as 0; in salsa20, c is used to provide the nonce. Additionally, keccakf160064bits is a primitive of this operation, where KEYBYTES and CONSTBYTES are defined as 0.

crypto_hashblocks This operation isolates the hash block computation from the complete computation. Given a state pointer statebyte, in, and corresponding length inlen, implementations under this operation are expected to perform an update to the state. sha256 is an example of a crypto_hashblocks primitive, and it defines STATEBYTES and BLOCKBYTES as 32 and 64, respectively.

crypto_hash This operation provides an interface for hash functions. The contents from the input pointer in, with length inlen, are hashed, and the result is written in out, a memory region with BYTES bytes. This API only supports statically known lengths for the hash output. For instance, in recent versions of SUPERCOP, the primitive shake256, an extendable-output function (XOF), is included. In this case, the corresponding implementations define

BYTES as 136 even though these functions can be called for different output lengths.

crypto_stream This operation defines two functions. The first, crypto_stream, receives a pointer to an array c with clen bytes, a nonce n, and a secret key k. NONCEBYTES and KEYBYTES define the sizes of n and k. The produced keystream, with length clen, is written in c. The second, crypto_stream_xor, allows encrypting an input in with length inlen by performing a XOR with the produced keystream, parameterized by n and k.

crypto_onetimeauth This operation defines two functions related to one-time usage authentication tags. The first, crypto_onetimeauth, allows computing and writing an authentication tag in out, given an input in and corresponding length inlen, and a secret-key k. The second, crypto_onetimeauth_verify allows verifying if a given tag h corresponds to the expected value for the tag. crypto_onetimeauth_verify can be implemented by calling crypto_onetimeauth to compute a tag h', and then crypto_verify can be used to check if h is equal to h'. BYTES and KEYBYTES define the tag and key length in bytes, respectively.

crypto_scalarmult This operation defines two functions. crypto_scalarmult performs the multiplication between the scalar n and a point p, and the result is written in q, whose size is defined by BYTES. For instance, in the case of curve25519, a primitive of this operation, crypto_scalarmult_base performs the multiplication between a given secret key n (scalar) and the x-coordinate of the base point, in this case 9. The result is written in q. Even though crypto_scalarmult_base can be defined as a call to crypto_scalarmult with p set to 9, the computation can be optimized because the base point is statically known.

Table 2.2 shows the number of primitives (column P) and implementations (column I) for each operation of SUPERCOP. The next set of columns, $\rightarrow$ O, $\rightarrow$ P, and $\rightarrow$ I, are explained next. The first of this set ($\rightarrow$ O) shows how many other operations include code from the current operation. The second ($\rightarrow$ P) and third ($\rightarrow$ I) columns show how many primitives and implementations are included by implementations from other operations. For instance, crypto_verify implementations are used by 67 implementations from the following operations (the crypto_ prefix is omitted): aead, auth, encrypt, kem, onetimeauth, scalarmult, and sign. crypto_hash is the most included operation: 19 times by crypto_encrypt, 36 times by crypto_sign, and 92 times by crypto_kem implementations.

The next three columns show how many implementations under each operation have ref, sse, and avx in the path. More frequently than not, a reference implementation contains ref in the directory name. The presented data not not only includes /ref/ but also /*ref*/. The same applies for sse and avx, where sse2, sse4, and avx2 implementations are counted. Please note that there are no guarantees that an implementation named ref cannot contain AVX2

instructions. For the purpose of this overview, this heuristic is sufficient. It is possible to observe that more than 25% in SUPERCOP of the implementations are, likely, reference implementations. Roughly 10% are specifically designed for the AVX/AVX2 extensions, and roughly 20% of the implementations include at least one .s or .S file (the presented count includes symbolic links).

Operation	P	I	→ O	→ P	→ I	*ref*	*sse*	*avx*	*.s *.S
crypto_verify	16	29	7	36	67	16	0	12	0
crypto_decode	35	90	2	31	51	35	0	19	0
crypto_encode	41	90	2	37	63	41	0	31	0
crypto_sort	2	18	1	18	23	0	0	2	1
crypto_core	56	194	6	35	47	63	3	59	25
crypto_hashblocks	4	34	2	7	8	4	1	4	12
crypto_hash	162	961	4	86	147	91	92	24	298
crypto_stream	41	299	5	42	78	16	19	33	141
crypto_onetimeauth	1	14	1	1	1	1	2	5	12
crypto_auth	7	19	1	1	1	6	4	0	0
crypto_secretbox	2	2	1	2	2	2	0	0	0
crypto_aead	479	1074	1	3	3	519	42	34	82
crypto_rng	3	3	0	0	0	3	0	0	0
crypto_scalarmult	3	18	2	5	5	4	0	3	10
crypto_box	2	2	0	0	0	2	0	0	0
crypto_dh	41	188	0	0	0	11	1	1	126
crypto_sign	182	428	0	0	0	151	25	77	10
crypto_kem	188	396	1	2	2	138	13	85	47
crypto_encrypt	61	92	0	0	0	38	0	13	0
	1326	3951				1141	202	402	764

Table 2.2: SUPERCOP operations overview.

2.1.2 Setup and Build

SUPERCOP relies on performance evaluations to determine which implementation performs best in a given environment. Before running the building script do (or do-part in recent versions of the toolkit), some configurations must be checked to reduce variability in benchmarks. SUPERCOP's website[2] contains a section covering this topic: "Reducing randomness in benchmarks". For instance, in the context of Intel CPUs, it may be necessary to disable Turbo-Boost, a feature that increases the CPU frequency depending on the CPU

[2] `https://bench.cr.yp.to/supercop.html`

load. Additionally, it might also be helpful to disable hyper-threading, a feature in which, for instance, two logical processors are associated with just one physical processor. The remaining discussion provides an intuition on how to build any SUPERCOP version and, as such, some comparisons between versions are included.

After the machine has been checked for adequate running conditions, the main building script, do, which contains the building code written in shell, can be used to run the toolkit. In comparison with recent versions of SUPERCOP, the do script is just a call to the do-part script with the argument all, "exec do-part all", and all the machinery is placed in do-part. The tasks performed by the building script are, essentially, the following:

1. Directory bench/machine-name (where machine-name is the actual machine name given by the command hostname minus some cleaning up) is created at the top-level directory of SUPERCOP, and it is in this directory that the produced results are put.

2. Some local scripts are created in the directory bench/machine-name/bin. Some of these scripts output the list of compilers that work on a given setup. The complete combination of compilers and compilers' options can be consulted in directory okcompilers/. For example, "gcc -O1" and "gcc -O3" could be two possible compiler configurations. The lists included by default in the toolkit can be pretty extensive, because many combinations between different flags are considered, but, in a scenario where the user is fine with just using one compiler configuration (to save time), these files can be edited before running the do main script. For example, in a context where AMD64 is supported as target architecture, the script okc-amd64 will be put in the aforementioned directory and, when executed, it outputs the list of C compilers that produce machine code suitable for that target architecture.

3. Some third-party libraries distributed with the toolkit are built as part of the initialization process, for instance, NTL[3] and GMP[4].

4. For each operation, for each primitive, and for each supported ABI (for instance, x86 and amd64 can be simultaneously considered if the specified compiler configuration list includes these as targets), the script tries to compile each implementation with each compiler configuration (for example, the output of okc-amd64). If the compilation is successful, the exported functions are tested. For each operation, there exists a file named try.c containing the testing code. Briefly, the implementation is tested by executing it with a series of deterministically-generated inputs and maintaining a checksum of the outputs. At the end of the testing procedure, the computed checksum is compared with the expected checksum[5]. Some other checks happen while the checksum

[3] `https://libntl.org/`

[4] `https://gmplib.org/`

[5] Expected checksums are placed in files named checksumsmall and checksumbig, for *small* and *long* sequences of testing, respectively.

is being computed. For instance, input pointers (i.e., message or secret key pointers) are unaligned, and a canary, in this case, a pseudo-random sequence of bytes, is copied to before and after the memory addresses where the implementation is expected to write. After execution, the canary is checked. An interested reader might also want to read file try-anything.c that sits at the top-level directory. This file contains the generic part of the testing code. If it passes all tests and is the fastest implementation so far (the execution time collected by one of the tests is considered) then the implementation benchmarking executable is created and put in a specific directory for later execution.

5. After testing, the fastest implementations are benchmarked. For operations that support variable-length inputs, such as crypto_hash, the number of CPU cycles for each considered input length is collected. The logs containing information about the machine, tests results, used compiler options, and benchmarks results are placed in a file named data. The main script compresses this file, and it can be uploaded to be included on the SUPERCOP website through the eBATS mailing list. A static library containing the best implementations is built.

The previous enumeration omits many details but it provides a general intuition of how SUPERCOP works. Recent versions of the SUPERCOP toolkit include new features, for instance, to create two different branches (constbranchindex or timingleaks) of execution, to handle implementations declared to be constant-time, or not.

Recent versions of SUPERCOP include a do-part script that accepts the option used, "./do-part used", where only essential components are built. Under some crypto_* operation, many implementations contain an empty file named used to mark them as part of the essential components that need to be built. A user who wants to use SUPERCOP as part of its development workflow and does not need all *used* implementations, can delete some of these files[6] to speed up the initial setup (but not all, as some are essential for the toolkit to work).

After the initial setup has been performed, the do-part can be invoked to evaluate a given primitive. This causes all the implementations to be evaluated as previously described: all implementations of a given primitive are tested and the one with the best execution time during tests is measured. It is possible to extend the do-part script to indicate which implementation should be considered, for instance, to fully evaluate it, even if it is slower than the alternatives[7]. The results are placed on the previously mentioned data file.

[6] https://github.com/tfaoliveira/libjc/blob/a884df7313cbe9967f71d8457c4e8cabf4ae0104/bench/patch/20210604/remove

[7] https://github.com/tfaoliveira/libjc/blob/a884df7313cbe9967f71d8457c4e8cabf4ae0104/bench/patch/20210604/do-part

2.2 Evaluation of new Cryptographic Implementations

For this work we use SUPERCOP, version 20210604. The machine used for the benchmarking has an Intel i7-6500U Skylake processor, clocked at 2.5GHz (Turbo Boost was disabled), Ubuntu 20.04, and GCC 9.3.0.

In SUPERCOP, under each cryptographic operation directory, there is a file named measure.c, which defines the code that is used to benchmark all primitives' implementations of that operation. measure.c defines a function named measure, which measures how the code performs. For variable-input implementations, this function measures TIMINGS times, the performance of the implementations, for each input length up until MAXTEST_BYTES. The inputs are initialized with random bytes. It outputs the collected number of cycles that the primitive took to execute and the corresponding median. For instance, measure.c from crypto_stream, defines TIMINGS as 15 and MAXTEST_BYTES as 4096. This evaluation is executed LOOPS times. As such, for each input length, if applicable, there are LOOPS measurements available for consultation in the log file. LOOPS is defined as a compiling flag in the do-part script. LOOPS is usually defined as 3, but if can be redefined in measure.c. For operations whose inputs' sizes do not vary, the evaluation occurs TIMINGS times.

The corresponding measure.c files were updated to reduce benchmarks execution time and, simultaneously, produce results that are compatible with the plots included throughout this document: for the primitives with variable-size inputs, the presented plots contain the number of cycles per processed byte in the y-axis; and the input length in bytes, using a logarithmic scale of 2, in the x-axis. Figure 2.1 shows the cycles per byte that four different ChaCha20 implementations take to process messages of different lengths. The execution time can be roughly estimated by multiplying the cycles per byte by the corresponding input length.

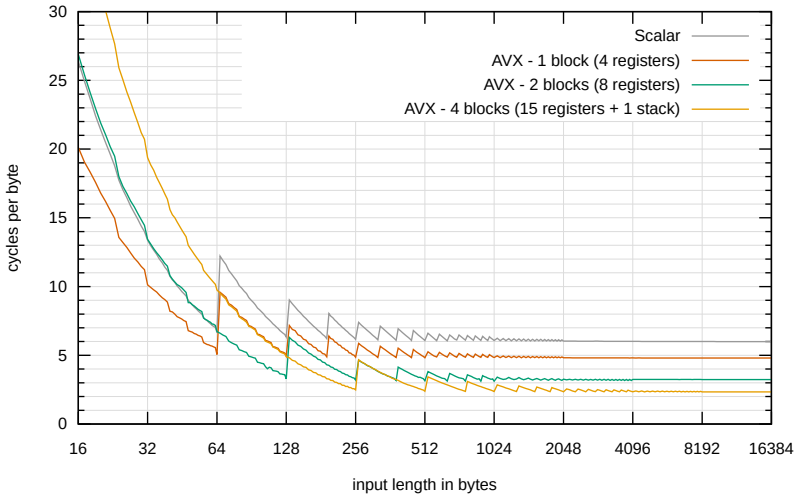

Figure 2.1: Benchmarking: Example plot.

The main reason for using a logarithmic scale in the x-axis is that too many data points usually do not allow for the production of easily readable plots. As the input length increases, the number of bytes that are skipped (during performance evaluation) increases accordingly: for inputs lengths of 2^n, the increment is defined as $max(2^{(n-6)}, 1)$. For instance, for input lengths between 1 and 64, the increment factor is 1 (with the primitive being measured for every possible input length on that range), from 64 (2^6) to 128 (2^7) the increment factor is 2, and from 8192 (2^{13}) to 16384 (2^{14}) is 128. Each plot interval, between powers of 2, contains 64 data points.

For the context of the primitives implemented during this work, where the performance behavior can be classified as linear, the presented method allows clear visualization of the implementation's performance and benchmark results availability (it is fast to collect measurements to guide the optimization process). MAXTEST_BYTES is defined as 16384, instead of 4096, to be consistent with other publications [ZBPB17, BHK+17]. To remove outliers, which become particularly visible when drawing plots, TIMINGS was increased, and the plots include the best-reported median for each considered point.

Chapter 3

Certified Compilation for Cryptography

Several cryptographic implementations are written in assembly, qhasm, or at a similar low level of abstraction, while many others are purely written in C. To achieve the best performance, some of these C implementations are vectorized and, as such, use intrinsics. Some of these optimized implementations exhibit competitive performance figures when compiled with non-verified compilers and compared with the corresponding low-level implementations. In order to understand if one could leverage all the research developments in formal verification of C implementations, an extension to a certified compiler, CompCert, was developed, the first with support for intrinsics [ABB+20b]. Section 3.1 presents an overview of this extension.

Section 3.2 presents the evaluation of this extended version of CompCert, conducted using version 20170105 of SUPERCOP. As a brief overview, SUPERCOP is a toolkit that allows building a cryptographic library targeted for a given environment: several implementations for each supported cryptographic primitive are distributed in this toolkit, and each implementation is tested and measured for a predefined set of compilers and compiler flags; the produced library includes the implementations that perform best for a given context; in addition to that, the toolkit collects extensive performance measurements. The considered version of the toolkit, 20170105, contains 2153 different implementations covering many CPU architectures. Some of these implementations are portable, meaning that they can be compiled for any architecture when a C or C++ compiler is available, while others are platform-dependent and use intrinsics. The number of available implementations is growing in each new release of SUPERCOP and, in more recent versions, for instance, 20210604, 3951 implementations are available. Although more updated versions of this toolkit are currently available, the primary source for this chapter was initially written in 2017 and later published in 2020.

3.1 CompCert for Cryptography

Our extension to CompCert was adapted from the 2.2 distribution of CompCert, and it focuses only on the part of the distribution that targets the ia32 architecture. There is no particular reason for our choice of CompCert version, except that this was the most recent release when our project started. Equivalent enhancements can be made to more recent versions of CompCert with some additional development effort.

3.1.1 Relevant CompCert Features

The architecture of CompCert is depicted in Figure 3.1. We follow [Ler09] in this description. The source language of CompCert is called CompCertC, which covers most of the ISO C 99 standard and some features of ISO C 2011 such as the `_Alignof` and `_Alignas` attributes. Some features of C not directly supported in CompCertC v2.2, such as struct-returning functions, are supported via rewriting from the C source during parsing. The semantics of CompCertC is formalized in Coq and it makes precise many behaviors unspecified or undefined in the C standard, whilst assigning other undefined behaviours as "bad". Memory is modeled as a collection of byte-addressable disjoint blocks, which permits formalizing in detail pointer arithmetic and pointer casts.

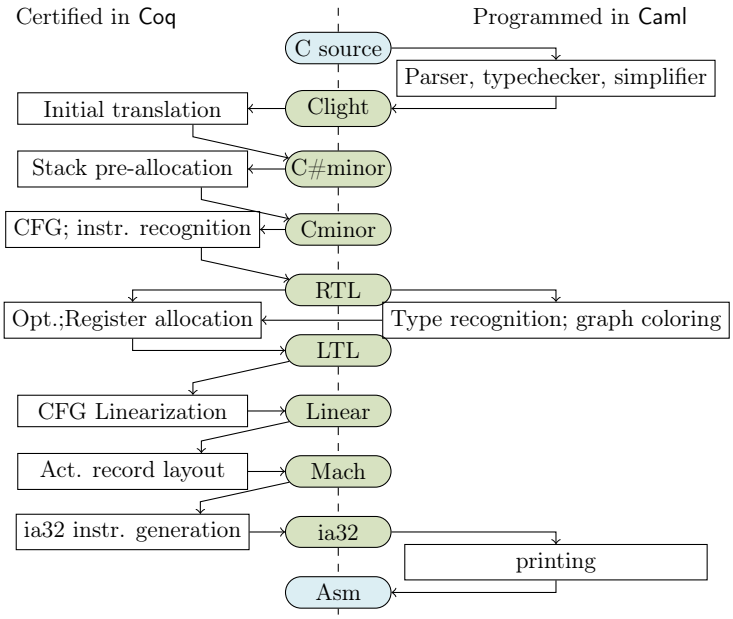

Figure 3.1: CompCert architecture.

CompCert gradually converts the CompCertC input down to assembly going through several intermediate languages. Parts of CompCert are not implemented directly in Coq. These include the non-certified translator from C to CompCertC and the pretty-printer of the

assembly output file. Additionally, some internal transformations of the compiler, notably register allocation, are implemented outside of Coq, but then subject to a *translation validation* step that guarantees that the transformation preserves the program semantics.

The front-end of the compiler comprises the translations down to Cminor: this is a typeless version of C, where side-effects have been removed from expression-evaluation; local variables are independent from the memory model; memory loads/stores and address computations are made explicit; and a simplified control structure (e.g. a single infinite loop construct with explicit block exit statements).

The backend starts by converting the Cminor program into one that uses processor specific instructions, when these are recognized as beneficial, and then converted into a standard Register Transfer Language (RTL) format, where the control-flow is represented using a Control Flow Graph (CFG): each node contains a machine-level instruction operating over pseudo-registers. Optimizations including constant propagation and common sub-expression elimination are then carried out in the RTL format, before the register allocation phase that produces what is called a LTL program: here pseudo-registers are replaced with hardware registers or abstract stack locations. The transformation to Linear format linearizes the CFG, introducing labels and explicit branching. The remaining transformation steps comprise the Mach format that deals with the layout of stack frames in accordance to the function calling conventions, and the final Asm language modeling the target assembly language.

The generation of the executable file is not included in the certified portion of CompCert – instead, the Asm abstract syntax is pretty-printed and the resulting programs is assembled/linked using standard system tools.

3.1.2 Semantic Preservation

CompCert is proven to ensure the classical notion of correctness for compilers known as *semantic preservation*. Intuitively, this property guarantees that, for any given source program S, the compiler will produce a compiled program T that operates *consistently* with the semantics of S. Consistency is defined based on a notion of *observable behaviour* of a program, which captures the interaction between the program's execution and the environment. Let us denote the evaluation of a program P over inputs $\vec{p}$, resulting in outputs $\vec{o}$ and observable behaviour B as $P(\vec{p}) \Downarrow (\vec{o}, B)$. Then, semantic preservation can thus be written as

$$\forall B, \vec{p}, \vec{o},\ T(\vec{p}) \Downarrow (\vec{o}, B) \quad \Longrightarrow \quad S(\vec{p}) \Downarrow (\vec{o}, B)$$

meaning that any observable behaviour of the target program is an admissible observable behaviour of the source program. Observable behaviours in CompCert are possibly infinite sequence of events that model interactions of the program with the outside world, such as accesses to volatile variables, calls to system libraries, or user defined events (so called annotations).

3.1.3 High-level view of our CompCert Extension

Our extension to CompCert is consistent with the typical treatment of instruction extensions in widely used compilers such as GCC: instruction extensions appear as *intrinsics*, i.e., specially named functions at source level. Calls to intrinsics are preserved during the first stages of the compilation, and eventually they are mapped into (typically) one assembly instruction at the end of the compilation. Intrinsic-specific knowledge is added to the compiler infrastructure only when this is strictly necessary, e.g., to deal with special register allocation restrictions; so transformations and optimizations treat intrinsic calls as black-box operations.

We have extended CompCert with generic intrinsics configuration files. Our current configuration was automatically generated from the GCC documentation[1] and the machine-readable x86 assembly documentation from x86asm.net.[2] This configuration file allows the CompCert parser to recognize GCC-like intrinsics as a new class of built-in functions that were added to the CompCert semantics. For this, we needed to extend the core libraries of CompCert with a new integer type corresponding to 128-bit integers; in turn this implies introducing matching changes to the various intermediate languages and compiler passes to deal with 128-bit registers and memory operations (e.g., a new set of alignment constraints; calling conventions; etc.). The new built-ins associated with intrinsics are similar to other CompCert builtins, apart from the fact that they will be recognized by their name, and they may carry immediate arguments (i.e., constant arguments that must be known at compile-time, and are mapped directly to the generated assembly code). These extended built-in operations are propagated down to assembly level, and are replaced with the corresponding assembly instructions at the pretty-printing pass. All changes were made so as to be, as much as possible, intrinsics-agnostic, which means that new instruction extensions can be added simply by modifying the configuration file. Overall, the development modified/added approx. $6.3k$ lines of Coq and ML, spread among 87 files from the CompCert distribution. We now present our modifications to CompCert in more detail.

Modifications to the CompCert front-end Modifications at the compiler front-end are generally dedicated to making sure that the use of intrinsics in the source file are recognized and adequately mapped into the CompCertC abstract syntax tree, and that they are subsequently propagated down to the Cminor level. This includes modifications and extension to the C parser to recognize the GCC-style syntax extensions for SIMD vector types (e.g., the vector_size attribute), as well as adapted versions of intrinsics header files giving a reasonable support for source-level compatibility between both compilers. These header files trigger the generation of the added builtins, whose specification is included on the configuration file. For each new builtin, the following data is specified:

[1] http://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html
[2] http://ref.x86asm.net

- the function identifier that is used to recognize the intrinsic by name;
- the signature of the intrinsic, i.e., the types of the input parameters and return type;
- an instruction class identifier that is used to group different intrinsics into different sets that can be activated/deactivated for recognition in different platforms (this is linked to a set of command-line option switches);
- the assembly instruction(s) that should be used when pretty-printing an assembly file in which that particular built-in operation appears;
- a Boolean value indicating whether the associated assembly instruction is two-address, which is relevant for register allocation later on.

Translation into CompCertC maps all vector types/values into a new 128-bit scalar type. Subsequent transformations were extended to support this data type.

Modifications to the CompCert backend The most intrusive modifications to CompCert were done at the back-end level, most prominently in the register allocation stage. CompCert uses a non-verified graph-coloring algorithm to compute a candidate register allocation, whose output is then checked within Coq for correctness. We added the eight 128-bit xmm register-bank to the machine description, taking into account that floating point operations in CompCert were already using 64-bit half of these registers. This implied extending the notion of interference used during register allocation and adapting the corresponding proof of correctness. During the constructions of the stack-frame layout, the calling convention for vector parameters/return-values was implemented supporting up to 4 parameters and the return-value passed on registers. The final component of our extensions was the addition to the assembly pretty-printer, supporting a flexible specification of the code to be produced by each built-in.

Consequences for semantics preservation Our new version of CompCert comes with an extended semantics preservation theorem that has essentially the same statement as the original one. The difference resides in the fact that the machine model now explicily allows built-in functions to manipulate 128-bit values. Note that, although we did *not* add a detailed formalization of the semantics of all instruction extensions, this is not a limitation when it comes to the correctness of the compiler itself: indeed, our theorem says that, whatever semantics are associated by a machine architecture to a particular extended instruction, these will have precisely the same meaning at source level. This is a powerful statement, since it allows us to deal with arbitrary instruction extensions in a uniform way. Such detailed semantics would be important if one wished to reason about the meaning of a program at source level, e.g., to prove that it computes a particular function. In these cases a formal semantics can be given just for the relevant instructions.

3.2 Evaluating CompCert on SUPERCOP

This section studies how the developed CompCert extension performs in SUPERCOP. Other compilers are included in this study for comparison.

3.2.1 Coverage

The SUPERCOP release we considered contained 2153 such implementations for 593 primitives. In Table 3.1 we present a more detailed summary of these counts, focusing on some interesting categories for this work. In particular, we detail the following successive refinements of the original implementation set: 1) the number of implementations that target the x86-32 architecture (x86), which we identified by excluding all implementations that explicitly indicate a different target architecture; 2) how many of the above implementations remain (x86-C) if we exclude those that are given (even partially) in languages other than C, such as assembly; and 3) how many of these use instruction extensions (x86-ext). Additionally, we give an implementation count that extends the x86 one by including also implementations that explicitly target 64-bit architectures (amd64); this gives an idea of how much coverage is lost by restricting attention to 32-bit architectures.

operations	x86	x86-C	x86-ext	amd64
aead	644	548	118	660
auth	19	19	6	19
box	2	2	0	2
core	25	25	0	29
dh	123	17	11	185
encrypt	13	11	4	13
hash	550	464	144	664
hashblocks	16	15	5	21
onetimeauth	9	2	0	11
scalarmult	7	5	0	14
secretbox	2	2	0	2
sign	62	47	11	65
stream	168	95	31	228
verify	3	3	0	3
total count	1643	1255	330	1916

Table 3.1: SUPERCOP implementation histogram.

One can see that 330 implementations resorting to x86 instruction extensions can be found in SUPERCOP, corresponding to 168 primitives — out of a total of 576 primitives that come

equipped with a x86 implementation. This set of primitives represents the universe over which the new formal verification tools that we put forth in this work will provide benefits over pre-existing tools.

Before moving to this detailed analysis, we conclude this subsection with a high-level view of the data we collected in SUPERCOP that permits comparing certified compilers to general-purpose compilers. This statistic is a byproduct of our work and we believe it may be of independent interest, as it gives us an indication of what the state-of-the-art in certified compilation implies for cryptography. Table 3.2 gives coverage statistics, i.e., how many implementations each compiler was able to successfully convert into executable code accepted by SUPERCOP in the machine we used for benchmarking. This machine has the following characteristics: Intel Core i7-4600U processor, clocked at 2.1 GHz, with 8 GB of RAM, running Ubuntu version 16.04. We note that SUPERCOP exhaustively tries many possible compilation strategies for each compiler in a given machine. The baseline here corresponds to implementations in the set tagged as x86-C in Table 3.1 that were successfully compiled with GCC version 5.4.0 or CLANG version 3.8.0.

architecture	x86-32				amd64	
operations	baseline	ccomp-2.2	ccomp-ext	ccomp-3.0	baseline	ccomp-3.0
aead	343	258	290	178	506	269
auth	16	10	10	8	19	10
box	2	2	2	2	2	2
core	21	25	25	25	29	25
dh	2	2	2	2	7	3
encrypt	4	5	**1**	5	5	6
hash	471	323	356	239	562	380
hashblocks	12	8	8	8	16	11
onetimeauth	5	5	5	6	7	7
scalarmult	6	6	6	6	13	9
secretbox	2	2	2	2	2	2
sign	12	0	0	2	18	3
stream	121	91	114	91	152	19
verify	3	3	3	3	3	3
total count	1020	740	**824**	577	1341	749

Table 3.2: SUPERCOP coverage statistics for various compilers.

The apparent discrepancy (1020 versus 1643) to the number of possible x86 implementations indicated in Table 3.1 is justified by the fact that some implementations omit the target architecture and incompatibility with x86 is detected only at compile-time.[3] In the table,

[3]The degenerate red value in the table is caused by implementations that use macros to detect intrinsic

ccomp refers to CompCert and ccomp-ext refers to the CompCert extension we developed. One important conclusion we draw from this table is that, at the moment, the version of CompCert we present in this paper has the highest coverage out of all certified compiler versions, due to its support for intrinsics. Nevertheless, we still do not have full coverage of all intrinsics, which justifies the coverage gap to the baseline. In particular, we do not support the `__m64` MMX type nor AVX operations.Furthermore, we do not use any form of syntactic sugar to hide the use of intrinsics, e.g., allowing XOR operations (^) over 128-bit values, which is assumed by some implementations fine-tuned for specific compilers.

3.2.2 Methodology for Performance Evaluation

We will be measuring and comparing performance penalties incurred by using a particular compiler. These penalties originate in two types of limitations: 1) the compiler does not cover the most efficient implementations, i.e., it simply does not compile them; or 2) intrinsic limitations in the optimization capabilities of the compiler. In particular, we will evaluate the trade-off between assurance and performance when compiling cryptographic code written in C for different versions of CompCert. Our metric will be based on average timing ratios with respect to a baseline measurement. In all cases, the timing ratio is always reported to the fastest implementation overall, often given in assembly, as compiled by a non-verified optimizing compiler in the best possible configuration selected by SUPERCOP. We now detail how we compute our metrics.

Performance Metrics We consider each SUPERCOP operation separately, so let us fix an arbitrary one called $o \in O$, where O is the set of all operations in SUPERCOP. Let C be the set of compilation tools activated in SUPERCOP and $P(o)$ a set of primitives that realize o. Denote $I(p)$ as the set of all implementations provided for primitive $p \in P(o)$. Let also t_C^p denote the fastest timing value reported by SUPERCOP over all implementations $i \in I(p)$, for primitive $p \in P(o)$, when compiled with all of the compilers in C. Note that, if such a value t_C^p has been reported by SUPERCOP, then this means that at least one implementation $i \in I(p)$ was correctly compiled by at least one of the configured compilers in C. Furthermore, t_C^p corresponds to the target code that runs faster over all the implementations given for p, and over all compilation options that were exhaustively attempted over C.

To establish a baseline, we run SUPERCOP with GCC version 5.4.0 and CLANG version 3.8.0 and collected measurements for all primitives. Let us denote this set of compilers by C^*. We then independently configured and executed SUPERCOP with different singleton sets of compilers corresponding to different versions of CompCert. Let us designate these by $C_{2.2}$, $C_{3.0}$ and $C_{2.2\text{-ext}}$, where the last one corresponds to our extension to CompCert described in section 3.1. Again we collected information for all primitives in SUPERCOP.

support; ccomp-ext activates these macros, but then launches an error in a GCC-specific cast.

For a given operation $o \in O$ we assess a compiler configuration C by computing average ratio:

$$R^o_C = \frac{1}{|P|} \cdot \sum_{p \in P} \frac{t^p_C}{t^p_{C^*}},$$

where we impose that t^p_C and $t^p_{C^*}$ have both been reported by SUPERCOP, i.e., that at least one implementation in $I(p)$ was successfully compiled via C and one (possibly different) implementation in $I(p)$ was successfully compiled by C^*.

When we compare two compiler configurations C_1 and C_2 we simply compute independently $R^o_{C_1}$ and $R^o_{C_2}$. However, in this case we first filter out any primitives for which either C_1 or C_2 did not successfully compile any implementations. The same principle is applied when more than two compiler configurations are compared; hence, as we include more compiler versions, the number of primitives considered in the rations tends to decrease. In all tables we report the number of primitives $|P|$ considered in the reported ratios.

Finally, since we are evaluating the penalty for using certified compilers, we introduced an extra restriction on the set of selected primitives: we want to consider only the performance of implementations covered by the correctness theorems. Our approach was heuristic here: if SUPERCOP reports that the most efficient implementation compiled by a CompCert version (including our new one) includes assembly snippets, we treat this primitive as if no implementation was successfully compiled.

3.2.3 Performance Boost from Certified Instrinsics-aware Compilation

In this section we measure the performance improvements achieved by our new version of CompCert supporting instruction extensions. Table 3.3 shows two views of the collected results: the top table compares three versions of CompCert, whereas the bottom table compares only the vanilla version of CompCert 2.2 with our extended version of it. In the bottom table we list only the lines where the set of considered primitives differs from the top table. The results speak for themselves: for operations where a significant number of primitives come equipped with an intrinsics-relying implementation, the performance penalty falls by a factor of 5 when comparing to CompCert 2.2, and a factor above 3 when comparing to CompCert 3.0.

In Table 3.3 we are including primitives for which no implementation relying on instruction extensions is given. In that case our new version of CompCert does not give an advantage, and so the performance gain is diluted. To give a better idea of the impact for primitives where instruction extensions are considered, we present in Table 3.4 the average ratios that result from restricting the analysis to primitives where instruction extensions are used. These

operation	$\|P\|$	ccomp-2.2	ccomp-3.0	ccomp-ext
aead.decrypt	120	24.78	18.75	5.23
aead.encrypt	120	28.04	20.85	5.32
auth	5	3.50	1.76	3.58
box.afternm	2	1.90	1.52	1.83
box.open	2	1.80	1.50	1.84
dh	2	5.59	4.67	5.81
dh.keypair	2	4.65	3.93	4.65
encrypt	6	3.09	2.68	3.23
encrypt.open	6	5.04	4.08	5.09
hash	25	7.55	6.27	3.51
scalarmult.base	2	5.29	4.29	5.16
scalarmult	2	5.72	4.66	5.79
secretbox	2	2.24	1.74	2.09
secretbox.open	2	2.09	1.64	2.03
sign	25	4.87	3.64	4.55
sign.open	25	3.73	2.87	3.55
stream	10	1.91	1.42	1.93
stream.xor	10	1.62	1.35	1.66
global	494	21.00	15.83	4.79

operation	$\|P\|$	ccomp-2.2	ccomp-ext
aead.decrypt	166	22.03	5.39
aead.encrypt	166	24.70	5.49
hash	32	8.13	5.01
global	639	20.00	5.08

Table 3.3: Performance ratios aggregated by instantiated operation.

results show that, as would be expected, intrinsics-based implementations allow a huge speed-up when compared to implementations in plain C. The most significant improvements are visible in the AEAD operations, where one important contributing factor is the enormous speed boost that comes with relying on an AES hardware implementation, rather than a software one.

The work presented in this chapter initiates a systematic study of the coverage of formal methods tools for cryptographic implementations. The statistics are encouraging, but there is significant room for improving coverage and performance. The development is available at `https://github.com/haslab/ccomp-simd`. We are currently porting our work to version 3.7 of CompCert, which will allow us to benefit from the new features that have been added since. Most notably, support to 64 bit architectures (in particular amd64), which by itself

operation	$\|P\|$	ccomp-2.2	ccomp-3.0	ccomp-ext
aead.decrypt	50	56.60	42.84	7.89
aead.encrypt	50	64.29	47.77	7.94
auth	2	4.23	3.88	4.06
hash	43	6.63	5.42	4.21
stream	3	1.43	1.05	1.34
stream.xor	3	1.31	1.19	1.25
global	201	48.08	36.25	6.92

operation	$\|P\|$	ccomp-2.2	ccomp-ext
aead.decrypt	60	55.10	7.51
aead.encrypt	60	62.18	7.56
hash	46	6.40	4.14
global	234	48.63	6.70

Table 3.4: Performance ratios aggregated by instantiated operation, restricted to primitives including at least one implementation relying on instruction extensions.

widens the applicability of the tool, and opens the way to support intrinsics for new vector extensions such as AVX, AVX2 and AVX-512. Finally, we are also updating our benchmarking set-up to the most recent versions of SUPERCOP, GCC and CLANG. We do not expect the main conclusions to change, but the number of assessed implementations will grow significantly.

Although the work presented during this chapter is certainly interesting from a research point of view and, to some extent, provides enough evidence that the support for vectorization is a key feature for this domain, there are still some significant challenges that need to be addressed. For instance, formally verifying a compiler that produces machine code that performs as fast (or with an almost negligible performance overhead) as machine code from unverified compilers is perhaps the major one. More concretely, proving the correctness of several optimizations' heuristics that can be used within this context can be very challenging tasks by themselves and which, most probably, require a massive investment at several levels.

In order to provide a solution for this problem, which is essentially finding a way of having some guarantees that a given piece of machine code performs as expected and, if it is compiled, the compilation process is guaranteed to yield code semantically equivalent to the source code, one possible approach is to leave optimization to the developer. This often comes with an assurance penalty, and so the following chapters focus on a new approach that aims to provide the best of both worlds: maximum developer control and formal verification capabilities rivaling those available for high-level languages.

Chapter 4

Jasmin

Jasmin is a low-level programming framework that enables the development of highly efficient, high-assurance cryptographic implementations [ABB+17a, ABB+20a]. This framework includes the Jasmin programming language, to write cryptographic implementations, and the Jasmin compiler, to compile Jasmin code into AMD64 assembly. Additionally, the Jasmin compiler allows to extract EasyCrypt *modules* from Jasmin implementations. These *modules* can be used within the EasyCrypt framework to prove the functional correctness or check the constant-time property of a given implementation. The Jasmin compiler is, itself, formally verified for functional correctness in the Coq proof assistant, meaning that all compilation steps preserve the semantics of the original Jasmin program.

The Jasmin programming language enables to write highly efficient implementations due to the level of control that it provides to the developer: most Jasmin statements map into one assembly instruction or, in the case of control-flow structures for instance, a predefined set of instructions. To provide this level of control, where the compiler does not automatically insert instructions of any kind, it is the developer's responsibility to choose which variables should be in registers or in the stack frame during the program's execution. Intuitively, the Jasmin programming language can be seen as lying in between qhasm and C. It provides the same level of control of qhasm, in the sense that the compilation output is entirely predictable and no unexpected instructions are introduced during compilation. On the other hand, Jasmin includes high-level features that simplify the development of cryptographic routines. Examples of this are the support for functions, C-like control-flow structures, and intuitive syntax for non-standard data types.

To verify the functional correctness of a given Jasmin program, the compiler allows to extract an EasyCrypt *module* that is semantically equivalent to the original program. Then, a high-level specification, also in the form of a *module* with a set of *procedures*, or a set of *operators*, can be written and proven equivalent to the extracted implementation using the EasyCrypt proof assistant. Depending on the complexity of the specification and corresponding im-

plementation, the proof can be separated into several logical steps, usually referred to as hops. For instance, an (EasyCrypt) specification can be a representation of an RFC or a mathematical description of a given algorithm: if a specification is written according to an RFC, and the corresponding Jasmin implementation is somehow similar to the specification, the equivalence proof between the specification and extracted implementation will be simple, and it will, most probably, require just one hop; if we consider a mathematical description as specification and a corresponding vectorized Jasmin implementation, the equivalence proof between these two can, and should, be separated into several logical steps, mostly for convenience and improved legibility. Regarding the verification of the constant-time property, it is possible to extract an EasyCrypt *module* specifically designed to achieve this purpose and, generally speaking, checking this property is a simple task.

This chapter's main goal is to present the Jasmin programming language from a user point of view. As an overview of the most important features, Jasmin supports three different types of functions: inline; local; and export. An inline function is fully inlined at the caller site, with no jumps or stack-setup instructions being generated in the resulting assembly code. These functions are particularly useful in contexts where one does not want to incur in the overhead associated with a function call but wants to logically isolate the code being called. Local functions may be used to reduce the size of the compiled assembly, since the same code is shared by all caller sites. Export functions can be called from external implementations where the System V AMD64 calling convention is supported. In the context of control-flow structures, Jasmin supports *if* statements and two kinds of loops: one specifically designed to unroll the loop body; and another that preserves the loop structure in the resulting assembly. Jasmin also supports the data types defined by the AVX and AVX2 extensions or, more specifically, 128-bit and 256-bit variables. These extensions define a set of registers and corresponding instructions that, intuitively and somehow informally, allow for the computation of multiple values during the execution of single instruction. Whenever possible, the syntax to use those instructions in Jasmin is improved, when compared to the approach used in the C programming language.

The discussion in this chapter includes language features that are currently available in the main branch of the Jasmin repository[1] which, in addition to the supported AMD64 instructions and AVX/AVX2 extensions, also supports the declaration of global constant arrays, a useful feature in many cryptographic implementations. If such branch happens to be no longer available at the time of this reading, it was, most probably, merged into the main branch of the repository. Floating types are not supported and ARM architecture is planned to be supported in the near future[2]. The following paragraphs describe how this chapter's organization.

[1] https://github.com/jasmin-lang/jasmin/tree/main
[2] https://github.com/jasmin-lang/jasmin/tree/arm

Section 4.1 starts by presenting the available storage classes and types which can be used to declare variables. It then discusses how arrays of different storage classes can be declared and used, how global variables can be declared and initialized, and how to use memory pointers.

Section 4.2 presents the available operators/instructions which always map into a predefined set of assembly instructions. Some instructions can be called using an intuitive syntax, such as "+=", while others must be accessed using their name.

Section 4.3 presents the available control-flow structures. It starts by presenting the available test conditions to use in the context of control-flow statements. It follows by discussing the available control-flow structures: *if*; *for*; and *while*.

Section 4.4 presents the available functions' types. It first presents *inline* functions, which are fully inlined at the caller site, then *export* functions, which implement the System V calling convention, and, to conclude, *local* functions, whose code is shared by all caller sites.

Section 4.5 focuses on the embedding of the Jasmin programming language in EasyCrypt, which allows to verify the functional correctness of a given implementation and to verify the constant-time property. It provides an overview of the typical proof infrastructure.

4.1 Variables

In Jasmin, a variable declaration starts with the specification of its storage class and is followed by the corresponding type and variable name — or set of names separated by a space if more than one variable with the same type is necessary. This section presents: the supported storage classes in §4.1.1; basic types in §4.1.2; declaration of arrays in §4.1.3; global variables in §4.1.4; memory pointers in §4.1.5; and a section's overview in §4.1.6.

4.1.1 Storage Classes

There are four storage classes in Jasmin: stack, reg, inline, and global. The following paragraphs describe each storage class.

stack A stack variable is allocated in the program's stack frame, and the compiler controls its relative address to the stack pointer, rsp, which is aligned according to the variables' types being used. Since most instructions in the AMD64 architecture allow one operand to be in memory, stack variables are not exclusively used to manage the number of live registers. Generally speaking, a live variable is a variable whose value can still be read by an expression or instruction that can be reached from the current context. The developer also does not need to be highly cautious when declaring stack variables, to save stack frame space for instance, given that the compiler analyses theirs' life span and merges them if possible. Merging in this

context means that the same memory space may be used for any stack variables that are not live at the same moment. It is impossible to access stack variables using pointer arithmetic.

reg A reg variable is allocated in a register chosen by the compiler's register allocator. In no circumstances a variable declared as reg will be automatically stored or loaded from the program's stack frame or any other memory region. That means that the compilation of a semantically valid Jasmin program may fail, for instance, if there are more live reg variables at any given line-of-code than there are registers. In this case, it is the developer's responsibility to choose which reg variables should be stored in the stack frame to free the necessary number of registers and, eventually, restore them with a load operation. The store and load from the stack frame can be performed by copying the reg variable into a stack variable and vice-versa. The assurance that a reg variable is placed in a register is necessary in the context of developing high-speed code, as it enables to write code that frequently outperforms the best C implementations.

inline and global An inline variable is always initialized with a statically known value. This value can be represented by a constant, expression, or even by the result of a statically known computation. As such, it can be used where an immediate value is expected. The global storage class specifies a variable whose value is also statically known and will be placed in the .data section of the resulting assembly file. Global variables are discussed in more detail in §4.1.4.

4.1.2 Basic Types

Jasmin's basic types can be grouped into three categories: words, booleans, and integers. The following subsections describe each category and how they relate with the available storage classes.

Words

The available types for representing words are u8, u16, u32, u64, u128, and u256. For instance, a variable declared with type u8 can hold 8 bits, and an u16 variable holds 16 bits. The u stands for unsigned, and there are no signed types, only operators or instructions which are discussed in more detail during 4.2. Word types can be used in combination with all storage classes.

As a first example, a reg 64-bit variable a can be declared as "reg u64 a;". Table 4.1, presented in the context of this section's overview, details which registers are used for each word type. It is important to highlight that some registers do overlap: if the previously declared variable a is allocated into rax in a given compiled program, it means that another reg variable declared

as u8, u16, or u32, and which is live at the same moment, cannot be allocated into al, ax or eax, since these correspond to slices of rax. Hence, the maximum number of live register variables for the word types u8, u16, u32, and u64 is 15. In total, there are 16 available registers in the AMD64 architecture, but in the context of Jasmin programming rsp is reserved for stack frame management. The remaining types, u128 and u256, also map in registers that overlap. All 16 xmm/ymm registers can be used.

Regarding word variables declared as stack: it was previously mentioned that the stack pointer is aligned according to the types being used. In this context, a local variable declared as "stack u256 a;" requires 32 bytes of memory space and the variable's memory address is guaranteed to be aligned at 32. The same applies to the remaining word types. There can be as many stack word variables as one wishes as long as the memory space that is required is compatible with the target environment, for instance, the operative system configuration can set a maximum size for the stack frame.

Regarding inline word variables: these are resolved during compilation. For instance, if we declare c as "inline u64 c;", and initialize it with the statement "c = ((1«27)-1);", which defines a value where only the first 26 bits are set, we can then copy this value into a variable x, declared as "reg u64 x;", with the statement "x = c;". The previous statement would be compiled as "movq $67108863, %rax" where 67108863 is the decimal representation of the expression ((1«27)-1), which corresponds to 0x3FFFFFF in hexadecimal. The destination register, in this case rax, depends upon the source code being compiled.

Booleans

The Boolean type is supported in Jasmin to provide an intuitive mechanism for handling the arithmetic flags or, more concretely, the carry, parity, zero, sign, and overflow flags. Flags are supported by the rflags register which, generally speaking, cannot be directly manipulated and whose values are set according to the result of arithmetic instructions. To access flags values in Jasmin, it is necessary to use reg bool variables. For instance, the statement "reg bool of cf sf pf zf;" declares five bool variables, one for each available flag.

It is common to name, for instance, the zero flag as zf, but any variable name can be used. Hence, the position of a bool variable in a given statement determines which flag it corresponds to. Boolean variables are different from other variables mainly because each flag is associated with a specific bit in the rflags register, which, to some extent, can be thought of as a global and shared variable. This means that, if we have two bool variables declared as "reg bool cf1 cf2;", and add four u64 variables x0, x1, y0 and y1 in the following way, "cf1, x0 += y0; cf2, x1 += y1;", where cf1 and cf2 hold the resulting carry of each addition, cf1 cannot be read after "cf2, x1 += y1;" is executed, because its value was overwritten. To propagate cf1 into the next addition, the previous statements could be updated to "cf1, x0

+= y0; cf2, x1 += y1 + cf1;". The previous example could also be written using a single bool variable, cf for instance. There could be cases where it is advantageous to use different names for the same logic bool value, for instance, to self-document the code in some particular case. Overall, flags are mostly used to manage control-flow, carry propagation in addition or subtraction chains, or also in the context of conditional moves.

Integers

To represent integers, the type int is available. This type specifies an unbounded integer that should be statically known. When used in the context of Jasmin functions, it should be declared as inline. When declared in the global scope, outside of any function context, the param keyword should precede the int type. Variables declared as int are mostly used for array indexing and in some loop structures. int variables can also be casted into words. For instance, if we declare an inline int variable i and a reg u32 variable x, and initialize i with the value $2^{33} - 1$, which requires 33 bits to be encoded and corresponds to the hexadecimal 0x1FFFFFFFF, then the statement "x = i;" compiles into the assembly instruction "movl $-1, %eax", where $-1, in this context, corresponds to the value 0xFFFFFFFF. When copying this inline int variable, whose value is unbounded, into the 32-bit variable x, only the first 32 bits are considered and the remaining bits are discarded. Generically, for any n-bit variable x and inline int variable i, statements such as "x = i;" correspond to $x = i \mod 2^n$.

4.1.3 Arrays

Jasmin supports the declaration of reg and stack arrays, for all word types. The array size must be statically known and specified in between square brackets, right after the type. Indexing starts at 0.

As an example, the statement "reg u64[2] x y;" allows to declare two register arrays with two u64 elements each. When accessing reg arrays, the index should be statically known as it does not exist a way of encoding a run-time index to access the available registers in AMD64. An element of a register array can be used whenever a register is expected, and, as such, the previously presented example to compute an addition with carry propagation could be rewritten to "cf, x[0] += y[0]; cf, x[1] += y[1] + cf;".

As a curiosity, it is possible to declare and use register arrays with more elements than the registers that are available, as long as not all elements are live at the same moment. In the context being discussed, u64 variables, it would be possible to declare x as having, for instance, 20 elements, using the statement "reg u64[20] x;", and throughout the Jasmin implementation up to 15 elements of such array could be live at any given line of code — this number, 15, depends on the existence of other live variables which are mapped into the same set of registers.

In the context of accessing stack arrays, the index can be a statically known value, or a run-time value placed in a reg u64 variable. For instance, "stack u64[25] state;" declares a stack array with 25 64-bit elements. If index is a reg u64 variable initialized with the value 8, then "state[8]" returns the same value as "state[(int) index]". In Jasmin, indexes are treated as integers and, as such, the cast "(int)" is necessary.

Stack arrays can also be accessed as if they have a different type. For instance, by slightly changing the previous statement "state[(int) index]" to "state[u8 (int) index]", it is possible to access the ninth byte of the state array. This feature is particularly useful, but not exclusively, to implement cryptographic primitives that read inputs and write outputs with arbitrary lengths.

Overall, arrays are considered regular values and can be used as function arguments and return values. This topic will be discussed in more detail during section 4.4.

4.1.4 Global Variables and Initialization

Jasmin allows the declaration of global values and arrays of values. These are usually declared in the global scope of a Jasmin program but can also be declared and initialized inside a function's body. The following paragraphs describe the declaration and initialization of scalars and arrays in the global scope. The initialization considerations presented in this subsection also apply to non-global variables, if the corresponding mov instruction is available for the types and values used in the initialization.

Global Scalars The declaration of a global variable is similar to the variables' declarations being discussed so far. The main differences are: the usage of the global keyword, which specifies its storage class, is not necessary when such variables are declared in the global scope; only word types are supported; only one variable can be declared for each statement; and each global variable must be initialized. For instance, the statement "u8 p = 0x5C;" declares a global variable p that contains the value 0x5C. The initialization value can be written as a decimal, hexadecimal, simple expressions which may include additions, subtractions or multiplications, and also as an array: the previous example could be rewritten as "u8 p = (8u1)[0,1,0,1,1,1,0,0];" which corresponds to the binary representation of 0x5C. Alternative ways of defining the same value are: "(4u2)[1,1,3,0]"; "(2u4)[5,11+1];"; "(2u4)[6-1,0x6*2];". Any word variable, global or not, can be initialized by an array whose total size corresponds to the size of the variable being initialized.

Global Arrays The declaration and initialization of a global array is similar to the declaration and initialization of a global scalar. To declare a global array, the size and all values of the array must be specified. As an example, "u8[2] p2 = {0x5C,0x5C};" declares

an array with name p2 with two u8 elements, both initialized as 0x5C. Each individual value of the array can be initialized as it was previously discussed, for instance, "u8[2] p2 = {(2u4)[5,12],(2u4)[5,12]};".

.data section All global variables, declared in a function or global scope, are placed in the .data section, which is aligned at 32 bytes with the .p2align 5 directive. The order in which variables are declared and initialized in Jasmin does not influence their final position in the .data section: the compiler sorts the variables by their type, from the largest to the smallest (from u256 to u8), and then it *prints* all values. This means that, if there is an u256 global variable, it is guaranteed to be aligned at 32 bytes, an u128 variable can be aligned at least at 16 bytes or 32 bytes, depending if there are u256 global variables or not. The same applies to all types.

Global Parameters In addition to the global variables that are placed in .data section, it is possible to declare global parameters using the unbounded integer type int. For instance, the statement "param int ROUNDS = 10;" declares the constant ROUNDS, which can be used in any expression where an int is expected. This feature is particularly interesting in the context of writing generic implementations.

4.1.5 Memory Pointers

Jasmin is a programming language designed to implement cryptographic primitives. As such, the implemented routines can be called from other programming languages that adopt the same calling convention. The API for cryptographic primitives often includes memory pointers in its specification. For instance, and for C APIs, function arguments declared as "unsigned char *in" for inputs, and "unsigned char *out" for outputs, with an additional argument that specifies the length of the inputs and outputs, are quite common. Since it is not possible to call externally defined functions in Jasmin programs, calling memory allocation routines such as malloc is not an option. Considering this, we can classify the memory that is used in Jasmin programs in two different types: external memory, which is managed by the caller function; and internal memory, that is controlled by the Jasmin compiler which corresponds to stack and global variables. These are discussed next.

External Memory

There is not a dedicated type in Jasmin to represent pointers. Although this distinction cannot be made through the available types, there are cases where it is advantageous to make this clear through the variable name, for instance, by using a prefix or suffix in the variable name when considered appropriate. The previously discussed function argument

"unsigned char *in" can be declared in Jasmin as "reg u64 in;". The u64 in the declaration is not related with the data that is pointed by in but, instead, it corresponds to the number of bits that are necessary to hold a pointer in 64-bit code. Considering this, all external pointers must be declared as u64. As an analogy, the equivalent type in C is the void*.

Since in points to a memory region and it can be seen as an array, an array-like notation is available. By default, accesses are made in 64-bit words. For instance, "[in]" allows to access the first u64 value that is pointed by in, and "[in + 8]" the second. To read or write the second byte, the following expression can be used: "(u8)[in + 1]". Memory accesses can be performed for any word type and the access type, in the previous expression "(u8)", must be specified for any type that is different from u64. For instance, the expression "(u256)[in]" allows to access the first u256 value that is pointed by in. The offset for this type of memory access (8 and 1 in the previous examples) is always specified in bytes. More details on how the offset can be built are provided during the next section, in §4.2.1.

Although it may be tempting to use external memory for other tasks than reading inputs and writing outputs, instead of using stack arrays, it is recommended to limit the usage of external memory in Jasmin programs as much as possible. The first reason for this is that, since stack variables accesses are done through the rsp register, which is already reserved, there is no need to occupy other registers to hold additional pointers. The second reason is that stack arrays allow for the same set of operations to be performed when compared with external pointers, if we exclude pointer's arithmetic, which can be avoided in most cases.

There could be, however, some cases where it is not feasible to use stack arrays at all. Given that the size of all Jasmin arrays should be statically known, in a scenario that requires the allocation of a stack whose size depends on a run-time value, and where it is not viable to declare such stack array with a fixed amounted of elements that covers all expected cases, an external memory pointer should be given to the Jasmin program by its caller.

Overall, and whenever possible, the programmer should take advantage of the Jasmin semantics, in which arrays are treated as regular values, to write much more readable code when compared to an equivalent implementation that only uses external memory. Readability plays an import role in the context of formally verifying the functional correctness of the developed Jasmin code.

Internal Memory

In the context of memory that is managed by the Jasmin compiler, more concretely, memory corresponding to stack arrays or global (read-only) arrays, there is an additional type that was not introduced so far: ptr. The purpose of this type is, currently, twofold: 1) reading global arrays elements using a run-time index; 2) provide the necessary infrastructure for local (non-inlined) Jasmin functions to receive stack arrays as arguments. The following two

paragraphs describe in more detail the ptr type for each context.

Global Arrays As previously discussed, global arrays are placed in the .data section. More concretely, all global variables and global arrays are placed under a common label, glob_data, sequentially, and according to their type. For instance, if we declare the following two global arrays, "u8[4] g8 = {1,2,3,4};" and "u64[4] g64 = {1,2,3,4};", the first array to appear in the .data section is g64 and the second g8. If we also declare, in a function's context, two u8 variables "reg u8 a b;", and read the first position of g8 into a with "a = g8[0];", the following assembly instruction is generated: "movb glob_data + 32(%rip), %al".

To access g8 using a run-time offset, the base address of g8 needs to be in a register. To achieve this, it is possible to declare a local variable "reg ptr u8[4] g8p;", and *copy* the array with "g8p = g8;". According to the Jasmin's semantic, and since all arrays are treated as values, a full copy of the array is performed. In practice, only a copy of the base address of g8 is made: "leaq glob_data + 32(%rip), %rcx". Given a variable declared as "reg u64 i;", the assignment "b = g8p[(int)i];" generates the follow instruction: "movb (%rcx,%rdi), %cl". In the previous assembly instruction the variable i corresponds to the register rdi. Figure 4.1 shows a small Jasmin program containing the presented example and corresponding assembly code. It is important to notice that a reg ptr variable requires one register. If, for some reason, the register that holds a reg ptr variable is needed for some other computation, a corresponding stack variable can be declared to store the address in the stack frame. In the context of this example, a variable "stack ptr u8[4] g8ps;" could be declared and g8p could be copied into it: "g8ps = g8p;".

```
u8[4] g8 = {1,2,3,4};
u64[4] g64 = {1,2,3,4};

export fn example1(reg u64 i) → reg u8
{
  reg ptr u8[4] g8p;
  reg u8 a b;

  a = g8[0];
  g8p = g8;
  b = g8p[(int)i];
  a += b;
  return a;
}
```

```
# ...
example1:
        movb    glob_data + 32(%rip), %al
        leaq    glob_data + 32(%rip), %rcx
        movb    (%rcx,%rdi), %cl
        addb    %cl, %al
        ret
        .data
# ...
_glob_data:
glob_data:
      .byte 1
      .byte 0
      # ...
```

Figure 4.1: Jasmin global arrays and the ptr type.

Stack Arrays In comparison to global arrays, stack arrays are held in the stack frame and the stack frame's base address is already in a register, rsp. Hence, to calculate the address of a given element of a stack array, rsp can be added with a constant offset if the index is statically known, or added with a register if the index is a run-time value. As such, the previous discussion regarding the relationship between global arrays and the ptr type does not apply to stack arrays. Instead, a stack array address needs to be copied into a corresponding reg ptr variable to allow for local functions to receive stack arrays as arguments. Given that local functions, as well as many other Jasmin features, are yet to be presented, the following discussion will be kept at a high level of abstraction.

Consider a case where there are two stack arrays x and y which are declared and used in a given function f1: "stack u64[25] x y;". At a certain point, function f1 needs to call function f2 two times, to perform the same computation on both arrays: "x = f2(x); y = f2(y);". Given that arrays are considered regular values, they need to be returned for the *update* to happen. In this case, and assuming that the generated assembly code of f2 is being shared by these two calls, f2 needs to have access to one of the following: 1) the offset of the array in relation to the rsp register; 2) or the memory address itself. Both options would require one register. Option 2) was chosen to allow for an uniform treatment of global and stack arrays.

In this context, two reg ptr variables could be declared, "reg ptr u64[25] xp yp;", and initialized with the corresponding addresses before the call happens. For instance, the code for the first call could be: "xp = x; xp = f2(xp);". Finally, to conclude this first call to f2, it is necessary to perform an additional *copy*, since that, according to Jasmin's semantic, xp and x are different arrays: "x = xp;". It is important to highlight that this last copy does not produce any assembly instruction. The complete Jasmin code for the two calls would then be: "xp=x; xp=f2(xp); x=xp; yp=y; yp=f2(yp); y=yp;". Since the presented example constitutes a common scenario, the compiler can handle all those moves automatically, and the code could be rewritten as: "x=f2(x); y=f2(y);". Nonetheless, it is important to be aware about which instructions are being generated when taking advantage of such features.

4.1.6 Overview

Table 4.1 presents an overview of the variables that can be declared in the context of Jasmin programs. Word variables declared with the reg storage class are mapped into the registers that are presented on the rightmost column of this table. In this context, reg variables declared with types u8, u16, u32, u64, and ptr, share the same physical space and, as such, no more than 15 variables with the mentioned types should be live at each moment or the compilation to assembly will fail. To recall, a variable is live if it contains a value that can still be read by any instruction that is reachable from the given context. reg u64 variables can also be used to hold external memory pointers.

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

In Type column, u8-u256 denotes all word types.
Indexing column refers to the supported indexes for accessing arrays. 'S' denotes statically-known indexes, and 'D' dynamic or run-time indexes.

Table 4.1: Jasmin variables declaration overview.

In the context of reg variables with types u128 and u256, up to 16 variables can be in a live state at any given moment. Register arrays can be declared with all word types and indexes should be statically known – denoted in column Indexing from table 4.1 as 'S'. It is not possible to declare reg arrays of bool's. reg ptr variables are used to reference Jasmin's internal memory (stack and global arrays) and the index can be a statically known value, or a run-time value placed in a reg u64 variable. Although it is not currently implemented, there are plans to extend the ptr type to support external memory pointers since it can benefit the formal verification process of the developed cryptographic primitives.

Regarding the stack storage class, all stack variables are placed in the stack frame. Similarly to the reg storage class, it is possible to declare scalar or arrays for any word type. Indexes can be statically known or run-time values. The stack ptr type allows to store a memory address from a reg ptr variable in the stack. It is not possible to access the elements of a given array using an stack ptr variable: the address needs to be in a register and, as such, it must be loaded into a corresponding reg ptr variable first. A stack ptr requires the same space as a stack u64 variable.

It is possible to declare global scalars or arrays for any word type. These are placed in the .data section of the produced assembly and, in the case of arrays, only statically known indexes are supported. To access a global array using a run-time index the array must be *copied* into a corresponding reg ptr variable. The copy is only logical and only a load of the

corresponding address is performed. Data is guaranteed to be aligned by the corresponding size. For instance, a u256 scalar or array is aligned at 32 bytes, u128 at 16 bytes, and so forth.

The inline storage class can be used to declare variables that are initialized with statically known values. The int type should be used to declare a variable containing an unbounded integer. Word types can also be used in combination with this storage class. Currently, the inline storage class does not allow for the declaration of arrays.

4.2 Operators and Instructions

In Jasmin, each statement corresponds to a predefined, and thus predictable, set of assembly instructions. Each operator is usually mapped in just one assembly instruction and operands must be declared with storage classes and types that are compatible with the corresponding assembly instruction. For instance, in the AMD64 architecture, most instructions require the destination operand to be a register. In those cases, the destination operand in Jasmin must be declared using the reg storage class. As a first example, if there are two reg u64 variables a and b, the statements "a += b;" and "a -= b;" correspond to addq and subq assembly instructions, respectively.

Operators provide an intuitive syntax to access a given assembly instruction. There are, however, some specific instructions whose behavior cannot be easily captured by an intuitive syntax such as "+=" or "-=". For that reason, Jasmin also supports direct calls to instructions. This feature is mostly useful in the context of AVX and AVX2 extensions. As an example, consider the instruction vperm2i128, which allows to combine two source operands depending on how an 8-bit immediate value is set, with the result being written in the destination operand. If a, b, and c are declared as reg u256 variables, then the Jasmin statement "c = #VPERM2I128(a, b, 0x20);" corresponds to the assembly instruction vperm2i128, with the 8-bit value being set as 0x20.

This approach is similar to the one used in C programming, where such instructions can be accessed via intrinsics. Intuitively, intrinsics are functions that are specially handled by the compiler, with those corresponding to a specific assembly instruction or, less often, a sequence of instructions. For instance, vperm2i128 can be used in C by performing a call to the function _mm256_permute2x128_si256. It is worth mentioning that, even though very specific machine instructions can be accessed through this mechanism in C, it is still the C compiler's responsibility to select whose variables are placed in registers or memory. Most C compilers also take advantage of the semantics of such instructions to optimize the resulting assembly code. This means that the instruction could change, or be replaced, during compilation. As an example of this behavior, consider a scenario where a given C compiler decides that a should be kept in memory and b in a register. Since vperm2i128 allows the

second source operand to be a memory operand, the position of a and b in the instruction could be swapped, and the immediate value changed from 0x20 to 0x02. The instruction would still compute the same result.

In Jasmin, optimizations of such kind are not performed, with the instruction being included in the produced assembly code exactly as it was written by the Jasmin programmer. As such, when performing direct calls to instructions in Jasmin, it is also necessary to declare operands with storage classes that are compatible with the target assembly instruction. Overall, it is the Jasmin programmer responsibility to be aware of such restrictions, and the architecture documentation should be consulted when necessary.

This section is organized as follows. Memory operands are presented in §4.2.1, as there are details regarding this type of operands that are worth discussing in more detail. Register and immediate operands do not have their own dedicated subsection as their usage is quite intuitive. The assignment operator is independently discussed in §4.2.2, as its behavior depends on the operands' types being used. Then, in §4.2.3, the remaining operators are presented, first for the types u8, u16, u32, and u64, and then for u128, and u256. To conclude the section, an overview of instructions that can be directly called is presented in §4.2.4.

4.2.1 Memory Operands

In section 4.1, some insights on how memory can be accessed in a Jasmin program were provided. For instance, given a reg u64 variable in that contains a valid pointer to external memory, the expression "[in]" allows to access the first u64 element pointed by in. It was also stated that, given a stack array declared as "stack u64[25] state;" and a variable index declared as reg u64 and initialized with 8, the expressions "state[8]" and "state[(int) index]" allow to access the same element in the array. Finally, to access global arrays using run-time indexes, it is necessary to load its address into a reg ptr variable with an equivalent type. Each or the previous expressions can be used as an operand of an operator, or in a direct call to an instruction, where a memory operand is allowed. This subsection details how memory operands can be written in more detail.

In the context of the AMD64 architecture, memory addresses can be computed as $b+(i*s)+d$, where b is a register that contains the base address, i is a register that contains an index, s is the scale factor which can be 2, 4 or 8, and d is the displacement, which can be an 8, 16, or 32-bit value. In Jasmin, only b is mandatory.

When dereferencing a pointer using the square bracket notation, [ ], all displacements are in bytes and, by default, 64-bits of data are accessed. For different word sizes, the corresponding word type should precede the expression. As an example, the expression "(u256)[in + 32]" allows to access the second u256 element that is pointed by in. In this example, only the base address, in, and displacement, 32, are used.

Some examples on how the index i and the scale factor s can be used are shown in figure 4.2. On the left side, several Jasmin expressions that allow to access the data that is pointed by in are shown. The preceding type, for example "(u256)" or "(u8)", is not shown in any of the examples as it is only relevant in the context of a complete statement. On the right side, the corresponding assembly operand is shown. In this figure, in corresponds to register rsi, and index to rdx. The displacement d is set to 16 for demonstration purposes.

[in]	(%rsi)
[in + 16]	16(%rsi)
[in + index]	(%rsi,%rdx)
[in + index∗2]	(%rsi,%rdx,2)
[in + index∗4]	(%rsi,%rdx,4)
[in + index∗8]	(%rsi,%rdx,8)
[in + index + 16]	16(%rsi,%rdx)
[in + index∗8 + 16]	16(%rsi,%rdx,8)

Figure 4.2: Jasmin addressing examples for memory pointers.

While for memory pointers dereferenced through the square bracket notation the offset is always specified in bytes, the index, when accessing stack arrays (or global arrays using a statically known value or through a reg ptr variable using a run-time index), refers to the position of the element in the array. This means that the Jasmin compiler uses the scale factor to perform typed accesses. For instance, the previous expression "state[(int) index]" corresponds to the assembly operand (%rsp,%rdx,8) where rsp contains the stack pointer, rdx corresponds to the index variable, and, since state is an array of u64's, the scale factor is 8. The displacement is not shown in this example, but, if more than one stack variable, or array, is used throughout a given Jasmin program, it is used by the compiler to specify the variable's position in the stack frame.

It was also previously mentioned that stack arrays can be accessed using a different type. The expression "state[u8 (int) index]" corresponds to the assembly operand (%rsp,%rdx), which means that no scale factor is applied, because for this case it would be 1 and, as such, it can be omitted. For u16 and u32 accesses, the assembly operands would be (%rsp,%rdx,2) and (%rsp,%rdx,4), respectively. The scale factor is only necessary when accessing arrays using run-time known indexes, because (almost[3]) any statically known index can be included in the displacement. Considering this, and also that the scale factor is only available for the values 2, 4, and 8, it is not possible to access u128 and u256 elements within stack arrays and using run-time indexes. Such types would require the scale factor to be 16, or 32, depending on the type. To circumvent this architecture limitation, an additional notation was introduced. If we consider that index contains the value 32, then the expression "state.[u256 (int) index]"

[3] Acordingly to Intel's documentation, the displacement can be an 8-bit, 16-bit, or 32-bit value.

state[0]	(%rsp)
state[8]	64(%rsp)
state.[8]	8(%rsp)
state[(int) index]	(%rsp,%rdx,8)
state.[(int) index*8]	(%rsp,%rdx,8)
state[(int) index + 16]	128(%rsp,%rdx,8)
state.[(int) index*8 + 16]	16(%rsp,%rdx,8)
state.[u256 (int) index]	(%rsp,%rdx)
state.[u256 (int) index + 32]	32(%rsp,%rdx)

Figure 4.3: Jasmin addressing examples for stack.

allows to access the second u256 element in the stack array state. When a dot, ".", is included in between the array name and the first square bracket, the compiler does not use the scale factor, and the corresponding assembly operand is just (%rsp,%rdx). The dot notation can be used when accessing stack arrays or with reg ptr variables, for any word type. The offset in these cases is always specified in bytes.

Figure 4.3 shows some examples of the available notations for comparison. The corresponding assembly for each expression is also shown. As an example, and since state is an array of u64's, with each element requiring 8 bytes of stack space, the expression state[8], which refers to the ninth element of state, corresponds to 64(%rsp) in assembly. When using the standard notation, without the dot, all displacements are multiplied by the elements' size. That can also be observed in the expression "state[(int) index + 16]", where the value 16 corresponds to 128 in assembly.

4.2.2 Assignment Operator

The assignment operator, "=", expects two operands and allows to copy one variable, which can be a scalar or an array, into another variable with a compatible type. This operator usually maps into a predictable number of move instructions and, if one of the variables is a reg ptr, it can correspond to a leaq instruction or no instruction at all.

As a first example, if two reg u64 variables a and b are declared, the statement "a = b;" compiles into one movq instruction. If those variables were declared with the u8 or u256 types, the corresponding instructions would be movb and vmovdqu, respectively.

In the context of converting between different types, it is also possible to access the movsx and movzx instructions using the assignment operator. The first instruction, movsx, allows to copy the source operand into the destination operand while performing a sign-extension. The movzx instruction performs a zero-extend copy instead. These instructions can be used for the types u8, u16, u32, and u64, where the source operand is smaller than the destination

operand. For instance, if the source operand is an u8 variable, then the destination operand can be an u16, u32, or u64 variable.

As an example, if c is an u8, u16, or u32 variable, and a an u64 variable, then the statement "a = (64s) c;" compiles into one of the following instructions: movsbq, movswq, or movslq. Other casts can be performed depending on the type of the destination operand, (32s) or (16s). The cast expression allows to distinguish between the sign-extend and the zero-extend move. To perform a copy of the variable using the zero-extend move, the previous statement could be changed to "a = (64u) c;", which would compile to the assembly instruction movzbq, assuming that c is an u8 variable (informally, cast operations have their 'u' swapped into the *end*).

Another interesting feature of the assignment operator is that it allows to copy arrays. Consider, for instance, the case where there are two register arrays x and y, declared as "reg u256[4] x y;", and x is copied to y with the statement "x = y;". This statement usually compiles into four vmovdqu instructions.

There is, however, a special case where mov instructions can be removed by the Jasmin compiler. Considering the last example, if y is not live after the assignment is performed, there is no need to perform the copy given that x can be allocated in the registers that previously belonged to y. This feature is useful, for instance, to perform a variable renaming. This works for any two variables, including arrays, with compatible storage classes and types. It is worth to note that this feature is not exclusive of the assignment operator.

In the context of stack arrays, if xs and ys are declared with the same type and size of x and y, "stack u256[4] xs ys;", the statements "xs = x; ys = y;" can correspond to eight vmovdqu instructions, from register to memory. Move instructions expect two operands and they support at most one memory operand. This means that, to perform a copy from memory to memory, each value should be copied into a register first, and only then into the destination memory address. As such, statements such as "xs = ys;" are not supported.

Regarding the usage of the vmovdqu instruction, instead of vmovdqa which assumes that the given address is aligned at the corresponding size: the Jasmin compiler can be updated to use this instruction whenever it is possible to ensure that the final address is aligned; nonetheless, since Jasmin memory variables with types u128 and u256 are already aligned by their size, no significant performance penalty is introduced by this missing feature.

In assignments where there is a reg ptr involved, there are two different scenarios, as it was mentioned during subsection 4.1.5, where this type was first presented. Consider, for instance, a scenario where there is a need to declare a reg ptr variable yp, to hold a reference of the previously declared ys stack array. Such variable could be declared as "reg ptr u256[4] yp;" and its initialization, "yp = ys;", could correspond to the assembly instruction "leaq 128(%rsp), %rcx". The actual registers, in this case rcx, and displacement, 128, depend upon

the complete source code being compiled. Since arrays are treated as regular values, if any element of an array is updated using yp, and if, at a later stage in the Jasmin implementation one wishes to continue to use ys because the reg ptr is no longer necessary, then the assignment "ys = yp;" should be performed before using ys again. This assignment, "ys = yp;", does not correspond to any assembly instruction. To conclude, if a reg ptr is on the left side of "=", the statement corresponds to a leaq instruction and, if it is on the right side, no assembly instruction is generated.

4.2.3 Operators

Operators for u8, u16, u32, and u64 types

Jasmin arithmetic operators corresponding to the addition, subtraction, multiplication, and division, for the types u8, u16, u32, and u64, are presented in table 4.2. The rightmost column of the mentioned table shows how Jasmin statements can be written, and, for each statement, the corresponding assembly instruction followed by a small description is also shown. The presented assembly includes the Jasmin variables, using the GAS syntax, to provide an intuition on how each statement is compiled. Each operand must be compatible with the corresponding assembly instruction.

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

Table 4.2: Jasmin arithmetic operators overview for u8, u16, u32, and u64.

Depending on the types being used, each assembly instruction will have a different suffix when compiled. Although this was left implicit in the previous section, the same applies for

mov instructions. For instance, the first statement in the table, "a += 1;", corresponds to the assembly instruction "inc a", which increments by 1 its operand. If a is an u8 variable, then the corresponding assembly instruction is "incb a". For the types u16, u32, and u64, the suffixes are w, l, and q, respectively. All presented instructions can be used with all types, with the exception of lea (load effective address), which is not defined for u8's.

Operands can have different storage classes, depending on the corresponding assembly instruction. For instance, inc, allows its operand to be a register or a memory operand. As such, any reg variable with the mentioned types can be used. If x is a reg u64 variable then the statement "x += 1;" can be compiled into "incq %rdi", if %rdi is the register chosen by the compiler's register allocator to hold x. Array elements can also be used as operands and, if x was declared as "reg u64[4] x;" instead, then "x[0] += 1;" could be used to increment its first element. Memory operands, as discussed in subsection 4.2.1, can also be used, and the statement "[in + index] += 1;" compiles to "incq (%rdi,%rsi)", and "(u8)[in + index] += 1;" to "incb (%rdi,%rsi)", for in allocated in rdi, and index in rsi. Variable and arrays declared as stack can also be used.

Some Jasmin statements presented in table 4.2 contain operands that are not shown in the corresponding assembly. The statements "cf, a += b;" and "h, a = a * b;", which correspond to the assembly instructions "add b, a" and "mul b", respectively, are examples where this happens. The operand cf is used in this context to represent the carry flag, and it should be declared as reg bool. Even though the flags are always implicitly set according to the result of an arithmetic instruction, with the exception of lea that does not affect any flags, it is necessary to have the flag explicitly present in the statement for its value to be used by subsequent Jasmin statements. As an example, the statement "a += b;" corresponds to the add instruction, which implicitly modifies the arithmetic flags according to the computed result, but, since the carry flag is not explicitly read into a Jasmin variable in this statement, it cannot be used by later expressions.

Regarding mul, which performs an unsigned multiplication, this instruction expects a to be in register rax, if used to perform a 64-bit multiplication, and b can be any register or a memory operand. Although this instruction can be used for other types, such as u32, the following discussion will continue, for convenience, in the context of 64-bit multiplications.

The result of multiplying two 64-bit values is an 128-bit value, which this instruction allows to compute. The most significant 64-bits of the result are placed in register rdx and the lower bits in rax. This means that the only variant in this instruction is b. Hence, b is the only operand that is explicitly stated in assembly. Although the statement is presented as "h, a = a * b;", to highlight the fact that the lower bits of the multiplication are placed in the same register, it could be rewritten as "h, l = a * b;". This does not mean that the original value of a will be preserved. In fact, after this alternative statement is executed, the previous value of a cannot be used anymore. Given that mul imposes several restrictions to the register

allocation, the Jasmin programmer should be specially cautious when using this instruction repeatedly, as some loads and stores are required in between them to allow for the input operand a to be allocated in rax, and also to keep register rdx free.

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

Table 4.3: Jasmin bitwise operators for u8, u16, u32, and u64.

Table 4.3 presents more operators in the context of bitwise and shift instructions, also for the types u8, u16, u32, and u64. With the exceptions of the andn instruction, from the BMI1 extension, and the not instruction, all remaining instructions require two operands. The first statement from this table allows to compute an exclusive or, "a ^= b;". The corresponding instruction, xor, is frequently used in low-level programming languages and by C compilers to zero out variables. In the context of Jasmin programming, a variable cannot be read if it is not initialized. This means that the statement "a ^= a;" causes the compilation to abort if a is an undefined value. To initialize a variable with zero using this approach, the programmer should perform a direct call to the set0 primitive, "a = #set0();", which is compiled to "xor a, a". The remaining bitwise operands are the logical inclusive OR, logical AND, the one's complement negation, also denoted as NOT, and the Logical AND NOT, that performs a logical AND between the second source operand and the negated first source operand, with the result being written in a third operand.

Regarding the presented shifts operators, the first two, the logical shift right and left, allow to shift the bits of a given variable to the right or left, with the bits from the corresponding side being discarded and the opposite side being filled with zeros. The last discarded bit is held the in carry flag, which cannot be accessed using this syntax. The next subsection, §4.2.4, details how the carry flag can be accessed by performing a direct call to the instruction. The arithmetic shift right, sometimes referred to as a signed shift right, is distinguished by the "s" in the middle of the operator. The main difference between the shr and sar instructions is that the introduced bits on the left side will contain the sign of the original value. As such, for positive values 0's are introduced, and for negative values 1's. The arithmetic shift left performs the same operation as the logical shift left and, for that reason, only the latter is supported. The shift count, presented in the table as "i", indicates how many bits a variable

should be shifted. It can be a statically known value, from 0 to 63 for u64 variables, for instance, or a run-time value that must be allocated in register cl, which is a sub-register of rcx that corresponds to the first 8 bits of it. Hence, to perform a shift using a run-time value, the shift count can be placed in a reg u8 variable which the Jasmin compiler will try to allocate in the mentioned register.

The syntax to access some of the operands from tables 4.2 and 4.3, is not mandatory: the presented statements that correspond to two operand assembly instructions, for instance "a &= b;" or "a += b;", can be written using a three operand syntax, for instance, "c = a & b;" and "c = a + b;", respectively. These statements still compile into the same assembly instructions but, in these cases, the variable c is merged with a as those instructions only support two operands.

Operators for u128 and u256 types

The available operators for the u128 and u256 types are presented in table 4.4. All assembly instructions presented in this next table are three-operand instructions and, as such, all Jasmin statements are written accordingly. Nonetheless, it is also possible to use the presented Jasmin operands with just two operands and, in those cases, the resulting assembly instruction will have the destination operand equal to one of the source operands.

Jasmin	Assembly	Notes
c = a +*type* b;	vpadd b, a, c	Addition.
c = a -*type* b;	vpsub b, a, c	Subtraction.
c = a **type* b;	vpmull b, a, c	Multiplication.
c = a ^= b;	vpxor b, a, c	Logical exclusive OR.
c = a | b;	vpor b, a, c	Logical inclusive OR.
c = a & b;	vpand b, a, c	Logical AND.
c = !a & b;	vpandn b, a, c	Logical AND NOT.
c = a <<*type* i;	vpsll i, a, c	Shift logical left.
c = a >>*type* i;	vpsrl i, a, c	Shift logical right.
c = a >>*type* i;	vpsra i, a, c	Shift arithmetic right.

Table 4.4: Jasmin operators for u128 and u256.

When compared with the previously presented tables, 4.2 and 4.3, some operands now include the empathized word *type*. A scalar variable with type u128 or u256 can be treated as an array of words with a smaller size. For instance, if a and b are u256 variables, they can be considered by some of the instructions as arrays of four 64-bit words, eight 32-bit words, sixteen 16-bit words, or even thirty two 32-bit words. In these cases, *type* can be replaced by 4u64, 8u32, 16u16, or 32u8. For u128 variables, *type* can be replaced by 2u64, 4u32, 8u16,

or 16u8. Not all instructions support all types.

As an example, and considering that a, b, and c are reg u256 variables, the Jasmin statement "c = a +4u64 b;" corresponds to the assembly instruction vpaddq. If we also consider that a is composed by four 64-bit values, a0, a1, a2, and a3, and its internal state can be interpreted as {a0,a1,a2,a3}, with the same happening for b, then the resulting value c can also be interpreted as {a0+b0, a1+b1, a2+b2, a3+b3}, with each of the additions being performed modulo 2^{64}.

In the case being discussed, vpadd was suffixed with q to denote that the operation is performed over 64-bit words. The different suffixes for the remaining *types*, 8u32, 16u16, and 32u8, are d, w, and b. The addition and subtraction operators support all the mentioned types. The presented multiplication operator, currently supports the type 8u32 which corresponds to the instruction vpmulld.

In the context of bitwise operators, such as the logical inclusive or exclusive OR, and also the AND and AND NOT, and considering that these computations can be performed independently of its sub-types, the *type* is not mandatory. The left and right shift operators currently support statically known shift counts for the types 4u64, 8u32 and 16u16 for u256 variables and 2u64, 4u32 and 8u16 for u128 variables. The arithmetic shift right is also available, and it is only necessary to replace the u by an s in the previously mentioned types. For instance, the statement "c = a >>4s64 32;" performs an arithmetic shift right (where the sign is preserved) by 32 in each of the four 64-bit elements of a.

4.2.4 Instructions

Instructions for u8, u16, u32, and u64 types

Figure 4.4 presents all instructions that can be directly called for the types u8, u16, u32, and u64. Some of the instructions in this figure can be accessed using the corresponding operator, such as MOV or ADD which correspond to the previously discussed "=" and "+", while others, such as BSWAP or ROL can only be used by performing a direct call. At the beginning of this section, some basic intuition on how to directly call such instructions was provided: the presented instructions must be prefixed with the symbol "#" and suffixed with a size, indicating the types of the inputs.

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

Figure 4.4: Jasmin instructions for u8, u16, u32, and u64.

For instance, the assembly instruction bswap allows to reverse the order of the bytes of a 32 or 64-bit register, and it only expects one operand, which is simultaneously source and destination operand. This means that BSWAP can be called with a reg u32 or u64 variable. For two variables a and b, with a being a reg u32 and b a reg u64, the following Jasmin statements allow to reverse the byte order of these two variables: "a = #BSWAP_32(a);" and "b = #BSWAP_64(b);". This instruction, bswap, does not affect any arithmetic flags.

On the other hand, the execution of the previously mentioned rotate left instruction, ROL, can affect the overflow and carry flags, depending on the count value. The remaining flags are unchanged. In Jasmin, each direct call to an instruction returns the computed values and also the updated flags. As an example, to rotate the previously mentioned reg u64 variable b by 1, and read the corresponding flag values into two reg bool variables of and cf, the following statement can be used: "of, cf, b = #ROL_64(b, 1);".

It is worth to mention that it is not mandatory that the return values are read into a variable. For instance, if one does not want to consider the values of such flags, and it is only interested in performing the rotate operation, the previous statement can be updated to "_, _, b = #ROL_64(b, 1);", where "_" denotes a variable that can be discarded. This feature is not limited to the context of arithmetic flags and it is also possible to only consider the updated flag values: "of, cf, _ = #ROL_64(b, 1);". If all return values are ignored, "_, _, _ = #ROL_64(b, 1);", the instruction is automatically removed by the compiler. This feature can also be used in any context where one or more values are returned, including in the context of function calls, which are discussed in section 4.4. Additionally, and for convenience, any left return values except one can be completely omitted. For instance, the statement "b = #ROL_64(b, 1);" is equivalent to the one where the flags are not considered. In this case, the compiler emits a warning to let the developer know that left return values are being introduced, as "_", during compilation.

Some instructions affect all supported arithmetic flags, which is the case for the SHL and SHR, that correspond to the operators "<<" and ">>", respectively. As an example, the statement "of, cf, sf, pf, zf, b = #SHL_64(b, 1);", allows to perform a left shift to b by one bit, and also *store* all flags for subsequent usage. In this particular case, the sign, parity, and zero flag, denoted in the example as sf, pf, and zf, are set according to the resulting value that is stored in b. The overflow flag behavior is only defined for one bit shifts, and the carry flag contains the last discarded bit.

The order in which flags are presented for this instruction is common to other instructions that affect the same set of flags. It is worth to recall that, although the presented discussion always uses what can be considered intuitive names for the flags being discussed, the chosen names are not mandatory and also not related with the flag that it represents: if, by mistake, the developer swaps the position of cf and sf in the previous statement, then cf will correspond to the sign flag and sf to the carry flag.

The CMOVcc instruction allows to perform a conditional move depending on a given condition. Consider, for instance, that there are two reg u64 variables b and c, and that c should be copied to b if the overflow flag, of, is set. The statement "b = #CMOVcc_64(of, c, b);" allows to achieve this and it would be compiled to "cmovo %rcx, %rax", for c allocated in register rcx and b in rax. The first argument of CMOVcc can be any condition that corresponds to a valid instantiation of the cmov assembly instruction. As an example of this, if we also consider that the carry and zero flag are also declared and initialized, the Jasmin statement "b = #CMOVcc_64(cf || zf, c, b);" compiles to "cmovbe %rcx, %rax", which corresponds to a conditional move if below or equal instruction. This instruction can also be accessed using a more intuitive syntax which is presented during the next section, more precisely in §4.3.2.

The SETcc instruction allows to set an u8 variable to zero or one depending on an input condition. It supports the same set of conditions as the CMOVcc instruction and, since it always returns an u8 value, the size suffix is not necessary for this instruction. As an example, the Jasmin statement "d = #SETcc(cf || zf);" compiles into "setbe %al", with d declared as a reg u8 variable and being allocated into al, a sub-register of rax.

To conclude, there are three Jasmin instructions, presented in figure 4.4, that map into the same assembly instruction, imul, which allows to compute a signed multiplication: IMUL, IMULr, and IMULri. Briefly, imul can be encoded with one, two, or three operands, with each encoding having different restrictions and, for that reason, each form corresponds to a different Jasmin instruction. The first Jasmin instruction, IMUL, allows to use the one operand form of imul, where one of the inputs is expected in rax, or a sub-register of it depending on the size of the operands. The computed result is written in registers rdx and rax, or sub-registers of them, making it similar to the MUL instruction. The two operand form of imul can be accessed with IMULr, and it allows to compute the truncated multiplication of two source operands, with the computed result being written in one of the source operands, which can be any register. The three operand form can be accessed using IMULri, which is similar to the previous one given that the multiplication result is also truncated. When compared to IMULr, IMULri expects an immediate value as one of its source operands and the destination operand is used exclusively for that purpose. All multiplication instructions discussed in this paragraph return the all the available arithmetic flags, in the order that was discussed for SHL instruction, for instance.

Instructions for u128 and u256 types

Figure 4.5 presents the instructions that can be directly called with variables with types u128 and u256. Similarly to what was discussed previously, some of these instructions can also be accessed through the corresponding operator. VPADD is one example of this case, where, for any three variables a, b, and c, declared as reg u256, the Jasmin statement "a = b +4u64 c;" is equivalent to "a = #VPADD_4u64(b, c); ".

AESDEC, AESDECLAST, AESENC, AESENCLAST, AESIMC, AESKEYGENASSIST, VAESDEC, VAESDECLAST, VAESENC, VAESENCLAST, VAESIMC, VAESKEYGENASSIST, VBROADCASTI128, VEXTRACTI128, VINSERTI128, VMOVDQU, VMOVSHDUP, VMOVSLDUP, VPACKSS, VPACKUS, VPADD, VPALIGNR, VPAND, VPANDN, VPBLEND, VPBROADCAST, VPERM2I128, VPERMQ, VPEXTR, VPINSR, VPMOVSX, VPMOVZX, VPMULH, VPMULHRS, VPMULHU, VPMULL, VPMULU, VPOR, VPSHUFB, VPSHUFD, VPSHUFHW, VPSHUFLW, VPSLL, VPSLLDQ, VPSLLV, VPSRA, VPSRL, VPSRLDQ, VPSRLV, VPSUB, VPUNPCKH, VPUNPCKL, VPXOR, VSHUFPS

Figure 4.5: Jasmin instructions for u128 and u256.

The statement "a = b *8u32 c;", which allows to compute the 32-bit truncated multiplication between the eight 32-bit elements from b and c, produces the same result as the statement "c = #VPMULL_8u32(a, b);". VPMULL can also be used with the _16u16 suffix to get the lower 16-bits of the multiplication between each corresponding 16-bit element from the source operands. In this case, and before the multiplication is performed, each 16-bit element is sign-extended to 32-bits. The corresponding assembly instruction is vpmullw. To avoid the sign-extension, instruction VPMULHU is available.

Not all instructions operate on multiple sizes that correspond to vectors and, in some cases, the suffix indicating the size of the operation is not necessary. Instructions VPERM2I128 and VEXTRACTI128, which allow to shuffle two u256 variables or to extract an u128 value from an u256 variable, respectively, are only defined for the mentioned types and, as such, no suffix is required.

In some cases, the suffix allows to distinguish between u128 and u256 variables. For instance, and for the previously defined reg u256 variables a, b, and c, the suffix _256 in the statement "a = #VPMULU_256(b, c);", which allows to multiply the low 32-bit integers from each 64-bit element, indicates that VPMULU is being called in the context of u256 variables. In the event that these variables were declared as u128, the statement "a = #VPMULU_128(b, c);" could be used.

VPSRLDQ and VPSLLDQ instructions allow to perform a right and left shift, respectively, over 128-bit lanes. An u256 variable has two 128-bit lanes and an u128 has one. The suffixes that can be used in these cases are also _256 and _128. For instance, the statement "a = #VPSRLDQ_256(b, 2); " allows to right shift each lane of b by 16 bits and store the result in a. The shift count in both instructions is specified in bytes.

4.3 Control-Flow

Jasmin supports the following control-flow structures: if; for; and while statements. The difference between the for and while statements is that the former is used to unroll the loop where the latter preserves the loop structure. This section starts by detailing the conditions that can be used in if and while statements and that depend on run-time values, §4.3.1. for loops do not use mentioned conditions because, since they are unrolled, all parameters must be statically known. The remaining subsections describe the if §4.3.2, for §4.3.3, and while §4.3.4 control-flow structures.

4.3.1 Conditions

This subsection presents the conditions that can be used to dynamically control the execution path of a given Jasmin program. Conditions that depend on run-time values can be grouped in two types: boolean conditions; and word conditions.

Bool Conditions

The bool type enables the Jasmin programmer to use arithmetic flags defined in the AMD64 architecture and, in this type of conditions, all operands must have the type "reg bool". For instance, if a given flag is available from a previous operation, such as the zero flag, "_, cf, _, _, zf, a = #SUB_64(a, 1);", with a being a reg u64 variable, then the conditions "(zf)" or "(!zf)" can be used to test, correspondingly, if a is zero, or not. The corresponding assembly instructions generated by the compiler, when these are used in the context of an if structure, are je and jne. Flags can also be combined if there exists a corresponding assembly instruction. For instance, the conditions "(cf || zf)" and "(!cf && !zf)" map into the assembly instructions jbe and jnbe, respectively.

Word Conditions

This type of condition requires the two operands to have the same word type, which must be one of the following: u8, u16, u32, or u64. The first operand must be a run-time value, and the second can be a run-time or statically known value. Table 4.5 presents the available comparison operators. For instance, if two variables are declared as "reg u64 a b;", the condition "(a > b)" allows to test if a is greater than b. By default, all comparison operators consider operands as unsigned values and, to perform a signed comparison, the corresponding operator must be suffixed with an s. For instance, if the previously declared variable a is used in the condition "(a > 0)", the unsigned value of a is considered, with this test being true if any bit in a is set to one. To consider a as a signed value, and include the value 0 in the test,

which may be useful if a is being decremented inside a while loop, then the condition (a >=s 0) can be used.

	Unsigned	Signed
Greater	>	>s
Less	<	<s
Greater or Equal	>=	>=s
Less or Equal	<=	<=s
Equal	==	==
Not Equal	!=	!=

Table 4.5: Jasmin comparison operators overview.

All word conditions map into the cmp assembly instruction and, as such, operands must have types that are compatible with the ones expected by this instruction. Table 4.6 presents an overview of the expected types for each operand: the first operand should always be a run-time value, meaning that it must be a reg or a memory operand (stack or *external memory*); if the first operand is a memory operand then the second operand must be a reg variable or an immediate value (either written directly in the condition or as an inline variable) as this instruction does not support two memory operands simultaneously; if the first operand is an u64 and the second operand is a statically known value (immediate or inline), the maximum value of for an unsigned second operand is $2^{31} - 1$ given that it is sign-extended from 32 to 64 bits; if a reg variable is used as the first operand, the second operand can then be a reg, stack, *memory*, or an inline value.

First operand		Second operand	
(reg | stack | *mem*[1])	u8	(inline[2] | reg)	u8
	u16		u16
	u32		u32
	u64	inline	u32[3]
	u64	reg	u64
reg	u8	(reg | stack | *mem*)	u8
	u16		u16
	u32		u32
	u64		u64

[1] *mem* designates a memory access instruction such as "(u8)[in + 8]".
[2] the immediate value can also be written directly, "a >= 255".
[3] in this case, the u32 constant is sign-extended to 64 bits.

Table 4.6: Jasmin comparison operands types.

4.3.2 if

Jasmin supports if statements with or without the else clause. An if statement with an else clause can be written as follows:

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

condition can be a boolean, word, or any other condition that can be statically resolved. For conditions that can be statically resolved, the Jasmin compiler removes the unused branch during compilation. For run-time conditions, the then_block contains the code that is executed if the condition evaluates to true and the else_block contains the code to be executed if the condition evaluates to false. For all types of conditions, both the then_block and else_block can contain other if statements, for and while loops, or calls to inline or local functions. The else clause is not mandatory. Given the way that the cmov instruction operates, an additional if syntax to specifically use this instruction is available:

```
v1 = v2 if ( condition );
```

Figure 4.6 presents an example where an if statement is used to initialize a stack u64 variable r with the value 1 or 0 depending if a variable a (declared as reg u64 and whose initialization is not shown) is different from 0 or not. The presented assembly code shows that, after comparing a with 0 using the cmpq instruction, a jump is performed if the zero flag is 0 (jne or *jump not equal*). If the jump does not happen (zero flag is 1), the else code is executed instead. The positions of the Jasmin code corresponding to the then_block and else_block clauses have their positions swapped when compiled to assembly.

```
reg u64 a;
stack u64 r;
//...
if(a != 0)
{ r = 1; }
else
{ r = 0; }
```

```
        cmpq    $0, %rdi    # a != 0
        jne     Lt0$1
        movq    $0, (%rsp)  # r = 0
        jmp     Lt0$2
Lt0$1:
        movq    $1, (%rsp)  # r = 1
Lt0$2:
```

Figure 4.6: Jasmin if statement with an else clause.

The else clause is not mandatory and it can be omitted. Figure 4.7 presents an alternative implementation of the figure's 4.6 example: variable r is always set to 0 and, if a is different from 0, r is set to 1. In this second example, the je instruction is used (instead of jne). As a side-note, it is possible to observe that variable r corresponds to (%rsp): the displacement is not used in this example because r is the only variable declared as stack.

```
reg u64 a;
stack u64 r;
//...
r = 0;
if(a != 0)
{ r = 1; }
```

```
    movq    $0, (%rsp)
    cmpq    $0, %rdi
    je      Lt1$1
    movq    $1, (%rsp)
Lt1$1:
```

Figure 4.7: Jasmin if statement without an else clause.

The cmov instruction allows to perform a conditional copy depending on how arithmetic flags are set, and it can be invoked directly or by using the if notation. This instruction requires the destination operand to be a register and the source operand to be a register or a memory operand. Figure 4.8 presents an alternative implementation of the previous examples, from figures 4.6 and 4.7.

To abide with the cmov requirements, the storage class of the variable r was changed from stack to reg and, since this instruction does not allow for immediate values in its source operand, a global variable named one was declared and initialized with the value 1. Alternatively, this variable could be declared as "reg u64 one;". The conditional copy can then be performed with the expression "r = one if(a != 0);", which maps into the assembly instructions "cmpq $0, %rdi" and "cmovne glob_data + 0(%rip), %rax".

```
reg u64 a r;
global u64 one;
//...
r = 0;
one = 1;
r = one if(a != 0);
```

```
    movq    $0, %rax
    cmpq    $0, %rdi
    cmovne  glob_data + 0(%rip), %rax
  # ...
glob_data:
    .byte 1
    .byte 0
  # ...
```

Figure 4.8: Jasmin if statement and the cmov instruction.

The condition of the if statement remains the same as the previous examples, "(a != 0)". Since this syntax is reserved for the cmov instruction, no brackets or semicolon are allowed to enclose the assignment "r = one" given than no more that one copy can be performed in this type of if. In the event that multiple conditional copies are needed, with all copies being based on the same condition, for instance, "r0 = one if(a != 0);" and "r1 = one if(a != 0);", then it is advantageous to compute the comparison explicitly, "zf = #CMP_64(a,0);" (remaining flags are omitted), and perform the conditional copies with the following statements: "r0 = one if(!zf);" and "r1 = one if(!zf);". This way, only one cmpq instruction is generated.

The examples presented so far were focused on if statements that depend on run-time values

but, if the condition of a given if statement can be resolved during compilation, meaning that all values from these conditions are known to the compiler, then the unused branch is removed. Consider the example presented in figure 4.9, where a global parameter i is declared and initialized with the value 5.

```
// declared in the global scope
param int i = 5;
// code from a function's scope
 inline int v;
 reg u64 r;
 v = 1; r = 0;
 if((i*2) % 3 == v)
 { r += 1; }
 else
 { r += 2; }
```

```
    movq    $0, %rax
    incq    %rax
```

Figure 4.9: Jasmin if statement resolved during compiling time.

Also consider, for the sake of this example, that the goal of defining such global parameter is to write generic Jasmin code: just by changing the initialization value of the param int i the produced assembly can be different. In the presented example, the condition ((i*2) % 3 == v) can be statically resolved by the compiler given that i and v are known. In this case, since the previous expression evaluates to 1, the branch corresponding to "r += 2;" is removed.

4.3.3 for

The for statement allows to specify loops that are completely unrolled during compilation. Conditions that depend on run-time values are not allowed in this construct. It is mostly used to process register arrays, whose indexes must be statically known, or to inline the same code as many times as necessary. The discussing will follow by first presenting the two types of for loops. It then follows by discussing a more complex example, where two for loops are used together with an if statement. A for loop can be written as:

```
for it=initial_value ( to | downto ) end_value { loop_block }
```

it is an inline int variable called the iterator, initial_value specifies the first value for the iterator. The keyword to, or downto, is used to indicate if the iterator is incremented, or decremented, in each loop iteration. The end_value specifies the last value for the iterator +1, or -1, depending if it is an upwards or downwards loop, respectively.

Figure 4.10 presents an example that uses two different types of for loops that allow to initialize two register arrays, x and y declared as "reg u64[4] x y;", with the same set of values.

The initialization of both arrays happens in a different order to demonstrate the differences between the to and the downto keywords. The first array to be initialized is x. The statement "for i=0 to 4" specifies a loop that iterates four times with i assuming the following values during execution: {0,1,2,3}. To initialize y, the inverted version of the previous for was used, "for i=3 downto -1", and i it iterated through the following set of values: {3,2,1,0}.

```
reg u64[4] x y;
inline int i;

for i=0 to 4
{ x[i] = (64u) i; }

for i=3 downto -1
{ y[i] = (64u) i; }
```

```
movq    $0, %rax
movq    $1, %rcx
movq    $2, %rdx
movq    $3, %r8
movq    $3, %r9
movq    $2, %r10
movq    $1, %r11
movq    $0, %rbp
```

Figure 4.10: Jasmin for loops.

Although not strictly necessary, the initialization of the register array elements in the shown example, for instance, "x[i] = (64u) i;", includes a cast to change the type of the iterator i from the unbounded int type into a bounded type with 64 bits. If an inline int variable b is initialized with the value $2^{64} + 1$ and copied into a u64 variable then the u64 variable would be initialized with the value 1.

The assembly code of this example shows that both for's were completely unrolled. The immediate values from the source operand of the movq instructions unveil the order in which they were unrolled. As a side-note, the last register to be allocated into the declared arrays was rbp, for the fourth position of y array, which is also available to use in Jasmin as the stack frame is managed using only rsp. The compiler includes the necessary instructions (store and load from the stack) to preserve nonvolatile registers in the generated assembly code.

Any control-flow structure can be used inside for loops, including other for loops. Consider the example shown in figure 4.11 where a register array "reg u64[9] x;" is used to represent 3x3 matrix. This array could also be declared as "reg u64[3*3] x;" if more convenient. To initialize all leading diagonal elements with 0 and the remaining elements with 1, two nested for loops containing an if statement can be used. The outer loop iterates i, for the values {0,1,2}, and the inner loop iterates j, also for {0,1,2}. In the nested loop block, the expression "3*i + j" can be used as index to access the array element from the current iteration. Additionally, if an element belongs to the leading diagonal then i is equal to j. An if can be used to check for this case and, since such condition depends only on statically known values, the if is resolved by the compiler. As an additional note, it would be possible to take advantage of global parameters to make the presented code *more* generic. For instance, let's consider that a global parameter M was declared in the global scope as "param int M = 3;". The x array

```
reg u64[9] x;
inline int i j;

for i=0 to 3
{ for j=0 to 3
  { if(i==j)
    { x[3*i + j] = 0; }
    else
    { x[3*i + j] = 1; }
  }
}
```

```
movq    $0, %rax
movq    $1, %rcx
movq    $1, %rdx
movq    $1, %rsi
movq    $0, %r8
movq    $1, %r9
movq    $1, %r10
movq    $1, %r11
movq    $0, %rbp
```

Figure 4.11: Jasmin nested for loops with an if.

could then be declared as "reg u64[M*M] x;" and the for loops as "for i=0 to M" and "for j=0 to M". The indexing "3*i + j" could also be replaced with "M*i + j". Also in this context, if one would like to access the secondary diagonal instead, the if condition could be updated to "((M-i-1)==j)".

4.3.4 while

This subsection presents the while loop, which can be used to specify two different kinds of loops that are, in their essence, equivalent to the *while* and *do-while* loops that can be found in other programming languages such as C. while loops are preserved during compilation and are never unrolled or partially unrolled. They are mainly used in contexts where unrolling does not bring any specific advantages, to process inputs or outputs with arbitrary length, or to reduce the resulting assembly size. A while loop can be written as follows:

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

Similarly to the if statement, the condition section can contain any condition that depends on run-time values, which were previously discussed in subsection 4.3.1. Conditions that can be statically resolved are not useful in this context. Such conditions would either correspond to a "while(true){" or a "while(false){" statement. For the "while(true){" case, and considering that Jasmin does not support *break* statements, which would allow to terminate the execution of a given loop under certain conditions, and also that the return statement should be the last statement of any Jasmin function, such use case does not currently has any practical applications. The loop_block section can contain any Jasmin code, including nested control-flow structures.

A loop that implements the *do-while* behavior can be written as follows:

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

The loop_block is executed at least once. The condition is evaluated at the end of each loop_block execution and, if it is true, then the it_block is executed once before the next execution of the loop_block. When the condition evaluates to false, the it_block is not executed.

Figure 4.12 presents an example where a while is used to copy up to 16 bytes from external memory, referenced from a pointer in, to a "stack u64[2] s;" array. This is the first example presented within a complete Jasmin function. The length of in, in bytes, is also provided as a function argument and, for the sake of this example, we can assume that 0 <= len <= 16.

```
export fn load_bytes1(reg u64 out in len) {
  stack u64[2] s;
  reg u64 i;
  reg u8 v;

  s[0] = 0;
  s[1] = 0;
  i = #set0();

  while(i < len) {
    v = (u8)[in + i];
    s[u8 (int)i] = v;
    i += 1;
  }
  //...
}
```

```
load_bytes1:
        #stack setup is omitted
        movq    $0, (%rsp)
        movq    $0, 8(%rsp)
        xorq    %rax, %rax
        jmp     Lload_bytes1$1
Lload_bytes1$2:
        movb    (%rsi,%rax), %cl
        movb    %cl, (%rsp,%rax)
        incq    %rax
Lload_bytes1$1:
        cmpq    %rdx, %rax
        jb      Lload_bytes1$2
```

Figure 4.12: Jasmin while loop.

After both positions of the stack array s are initialized with 0, meaning that all 16 bytes corresponding to this array are zeroed out, the assignment "i = #set0();" is used to initialize i with 0 with an xor instruction. A while loop is then used to perform a copy, byte by byte, from the memory pointed by in to the stack array s. The condition of this loop is (i < len) and i is incremented at every loop iteration by the statement "i += 1;".

In the while loop_block, the copy is performed by first loading the value pointed by "in + i" into a reg u8 v variable with the expression "v = (u8)[in + i];" and then copying v into s with the expression "s[u8 (int)i] = v;". Since the mov instruction does not allow for two memory operands, it is necessary to load the value into a register first. As a side-note, the expression "s[u8 (int)i] = (u8)[in + i];" is currently accepted by the compiler, but the same

two mov instructions would be generated, one to copy from memory to a register that is automatically chosen by the compiler, and another to copy from a register to memory.

Regarding how the code from figure 4.12 is translated into assembly, right after variable i (register rax) is zeroed with the xorq instruction, an unconditional jump is performed to the end of the loop_block, where the loop condition is evaluated. If the condition is true, meaning that i < len, then a jump into the beginning of the loop is performed.

An alternative implementation for the function load_bytes1 is presented in figure 4.13. This alternative function, load_bytes2, uses a loop written using the *do-while* notation and the *loop_block* is executed at least once. This means that there should be at least one byte to copy, changing the previous prerequisite into 1 <= len <= 16.

```
export fn load_bytes2(reg u64 out in len) {
  stack u64[2] s;
  reg u64 i;
  reg u8 v;
  s[0] = 0;
  s[1] = 0;
  i = #set0();

  while {
    v = (u8)[in];
    s[u8 (int)i] = v;
    i += 1;
  } (i < len)
  { in += 1; }
  //...
}
```

```
load_bytes2:
        #stack setup is omitted
        movq    $0, (%rsp)
        movq    $0, 8(%rsp)
        xorq    %rax, %rax
        jmp     Lload_bytes2$1
Lload_bytes2$2:
        incq    %rsi
Lload_bytes2$1:
        movb    (%rsi), %cl
        movb    %cl, (%rsp,%rax)
        incq    %rax
        cmpq    %rdx, %rax
        jb      Lload_bytes2$2
```

Figure 4.13: Jasmin do-while loop.

Regarding how this second example is compiled, it_block is printed in first place and loop_block in second place. The it_block corresponds to the expression "in += 1;" in Jasmin and "incq %rsi", and it is incrementing the pointer in. Given that it_block should only be executed after the condition is evaluated to be true, an unconditional jump is introduced in the assembly to avoid this section in the first loop iteration. In this particular example not much is saved by using the optional it_block, given that one jump is introduced to avoid one execution of "incq %rsi".

4.4 Functions

One of the most interesting features in Jasmin is the support for functions. Three types of functions are available: inline; local; and export. As a brief overview, inline functions are inlined at the caller site and can be intuitively seen as an extended macro mechanism. The second type, local, allows to write functions whose code is shared. Local functions do not require any specific keyword to be declared. The third type, export, allows to write functions that (partially) implement the System V calling convention and can be called from external code. The following subsections describe the available function types in the following order: inline functions in §4.4.1; export functions in §4.4.2; and local functions in §4.4.3.

4.4.1 Inline Functions

An inline function can be written as follows:

```
inline fn function_name ( arguments ) → return_types { function_body }
```

A valid function_name starts with an underscore or a letter and, afterwards, it can contain letters, numbers, or underscores. Function inputs are declared in arguments. Inputs can have the following storage classes: stack; reg; and inline. Any type can be used, including arrays. An empty list of inputs is also allowed, which can be useful to define functions that return constant values. As an example, "(reg u64[4] a b, stack u64 s, inline int k)" is a valid argument list, with a and b being declared as register arrays of equal length and type, s as a stack variable, and k as an unbounded integer.

Output types are specified in return_types. This is an optional section and it may be omitted (including the arrow) if the function does not return anything. Multiple return types are allowed and should be separated by commas. For instance, if a given function returns two register variables with types u8 and u32, and also an integer, the following return type can be used: "→ reg u8, reg u32, inline int".

All local variables should be declared at the beginning of function_block. After that, any Jasmin code can be used, including control-flow structures and function calls to other inline or local functions. It is not possible, however, to define a function within a function. The last statement in function_block is the return statement (if return_types is defined).

Figure 4.14 presents an example of an inline function, This function, __add4, receives two register arrays, a and b, and it returns the addition (with carry propagation) of these two in a different register array named r. For this example, we can assume that a and b represent two 255-bit integers that can be reconstructed with the following expression, $2^{192} \cdot x[3] + 2^{128} \cdot x[2] + 2^{64} \cdot x[1] + 2^0 \cdot x[0]$, by replacing x by a or b. Given that each number is less or equal to $2^{255} - 1$, the last carry is guaranteed to be zero.

We also want to consider that a and b are read-only arrays in the sense that, if these arrays are used after __add4 is called, the original values are still preserved. Currently, there is no mechanism to explicitly state this, such as the keyword const which is frequently used in C programming language. To achieve this, and given that the assembly instructions that are used to perform this computation (add and adc) are two operand instructions, a needs to be copied into r first (it could also be b). To perform this copy, only one statement is required: "r = a;".

In Jasmin, it is not possible to generate assembly code solely from inline functions: these functions need to be called within the scope of an export function. Given that the code that is shown on the right side of figure 4.14 was generated from a context where a is no longer used after it was copied to r, a and r were merged by the compiler. As such, the four mov instructions are not included in the resulting assembly code. If __add4 was called from a context where a was still live, the mov instructions would be included.

```
inline fn __add4(reg u64[4] a b) → reg u64[4]
{
  inline int i;
  reg bool cf;
  reg u64[4] r;

  r = a;
  cf, r[0] += b[0];
  for i=1 to 4
  { cf, r[i] += b[i] + cf; }

  return r;
}
```

```
addq    %r10, %rax
adcq    %r11, %rcx
adcq    %rbp, %r8
adcq    %rdx, %r9
```

Figure 4.14: Jasmin inline function.

Consider, for instance, that the presented function is used in two different situations, to compute $a = a + b$ and $r = a + b$. In a hypothetical scenario where the compiler does not merge variables, and where performance is a matter of concern, two different addition functions would have to be written, one for in-place additions and another for non in-place additions. In the current state of the Jasmin compiler, only one version of the function is needed and unnecessary mov instructions are removed during compilation.

Jasmin inline functions can receive and return inline variables. These variables are no more than statically known variables that are usually initialized with an immediate value or expression. In this context, we can revisit a previous example (§4.3.3, figure 4.11) in which a register array with 9 positions that was being used to represent a 3x3 matrix was initialized. Two nested for loops, iterating i and j, were used to perform the initialization, and the index

was calculated using the expression "3*i + j". The index computation can be encapsulated by an inline function, that receives i and j and returns an inline int that corresponds to the index computation. Figure 4.16 presents an implementation for this function. The introduction of this additional function did not cause any change in the produced assembly code.

```
inline fn __m3x3() → reg u64[9] {
  reg u64[3*3] x;
  inline int i j k;

  for i=0 to 3 {
    for j=0 to 3 {
      k = index(i, j);
      if(i == j)
      { x[k] = 0; }
      else
      { x[k] = 1; }
    }
  }
  return x;
}
```

```
inline fn index(inline int i j) → inline int {
  inline int r;
  r = (3*i + j);
  return r;
}
```

Figure 4.15: Jasmin inline function with an inline return type.

Another interesting use case for inline functions that return inline variables is related to one of the permutations of KECCAK, more concretely, the ρ permutation [BDPVA09]. For convenience, the permutation's pseudocode is presented in Algorithm 1. The general intuition for the ρ permutation is that, given an initial state a, a new state A is built by rotating each element of a, with the exception of one element, which is just copied. When KECCAK-f is instantiated with a width of 1600 bits, KECCAK-f[1600], and implemented in a 64-bit architecture, then a can be seen as a 5x5 matrix of u64 values ($5 \cdot 5 \cdot 64 = 1600$).

Algorithm 1 KECCAK-f *ρ permutation*

$A[0,0] = a[0,0]$
$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
for $t = 0$ to 23 **do**
  $A[x,y] = \mathrm{ROT}(a[x,y], (t+1)(t+2)/2)$
  $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
end for

To implement Algorithm 1 in Jasmin it is possible to follow two different approaches. The first is to implement the algorithm exactly as it is shown, using a function that receives a

stack array a with 25 elements and uses a for loop to iterate t and rotate the elements of a. Assuming that $a[x, y]$, from Algorithm 1, corresponds to a[x + y*5] in Jasmin, such loop would access the array a in the following order: {1,10,7,11,17,...}. To access the elements of a sequentially, it is possible to follow a slightly different approach and separate the computation of the rotation counts from the rotation itself.

Figure 4.16 presents an implementation of Algorithm 1 using this second approach. The function presented on the left side, rotates, receives two indexes, _x and _y and returns the corresponding rotation count for that particular position. To achieve this, it is necessary to first initialize a variable r with 0 to cover the case where _x and _y are equal to 0. The for loop performs the specified computation and, if _x and _y are equal to the local x and y, the rotation count is stored in the returning variable r. Since Jasmin only supports return statements as the last statement of a function, the *execution* cannot be terminated inside the for loop body and it *continues*. However, this is not an issue: given that rotates is an inline function that returns an inline variable, and only uses statements that can be statically resolved during compilation, the compiler replaces the call to rotates by the corresponding value.

```
inline fn rotates(inline int _x _y) → inline int {
  inline int r x y z t;

  r = 0; x = 1; y = 0;
  for t=0 to 24 {
    if( _x == x && _y == y)
    { r = ((t + 1) * (t + 2) / 2) % 64; }
    z = (2*x + 3*y) % 5;
    x = y;
    y = z;
  }
  return r;
}
```

```
inline fn rho(stack u64[25] a) → stack u64[25] {
  inline int r x y;

  for y = 0 to 5 {
    for x = 0 to 5 {
      r = rotates(x, y);
      if(r != 0)
      { _, _, a[x+y*5] = #ROL_64(a[x+y*5], r); }
    }
  }
  return a;
}
```

Figure 4.16: Jasmin implementation of Keccak ρ permutation.

The second function, rho, is presented on the right side of figure 4.16. It receives an array declared as "stack u64[25] a" and returns the updated state. To perform this permutation, two for loops are used, with the outer one iterating y. This way, all accesses happen sequentially. Whenever called, this function causes 24 rol assembly instructions to be generated. rol requires two operands, the first can be a register or memory address, and the second must be an immediate value that specifies the rotation count. The instruction performs a rotation to the left and the first operand is simultaneously the source and destination operand. The rotation by 0 is avoided by using an if statement that can be statically resolved.

4.4.2 Export Functions

export functions can be called from external code if the System V calling convention is supported. The declaration of export functions is similar to the declaration of inline functions, and only the preceding keyword changes:

```
export fn function_name ( arguments ) → return_types { function_block }
```

Functions declared as export can receive up to 6 register arguments with types u8, u16, u32 and u64. Each argument map into registers rdi, rsi, rdx, rcx, r8 and r9 (or sub-parts or them depending on the types) in accordance to their order in the argument list. Stack arguments are not supported. In addition to these, export functions can also can receive up to 8 register arguments with types u128 or u256. Each of these arguments map into registers xmm0-xmm7/ymm0-ymm7, depending on the used types.

Only one reg variable can be returned and, as such, the return_types section must contain only one type. If the return type belongs to the set {u8,u16,u32,u64}, then register rax (or a sub-part of it) is used. If the function returns an u128 or u256 variable, then the register xmm0 or ymm0 is used.

Similarly to the inline functions, function_block contains all local variables declarations first, and then any Jasmin code can be used. It is not possible, however, to call other export functions or define functions within functions. The Jasmin compiler handles the stack setup and also saves caller registers (rbx, rbp, r12, r13, r14, and r15) depending on how many registers are necessary to successfully compile the program.

Figure 4.17 presents an example of an export function, add4, that creates an interface for external code to use the previously defined inline function __add4, from figure 4.14. The presented function receives two arguments, ap and bp, which are variables that contain pointers to two external arrays with 8*4 bytes each. add4 does not have a return type. Since ap is declared first, it is allocated in register rdi and bp in rsi. Although not strictly necessary for this example, given that this program would only use up to 10 registers, a stack variable aps was declared to store the value of ap in the local stack. This way, the instructions that are related with the stack setup can be observed and discussed. Two additional functions, __load4 and __store4, to load and store register arrays are also presented. The first assembly instruction that is executed in the assembly version of add4 is "movq %rsp, %r11". The old value of the stack pointer must be saved to be restored before the function returns. Since there are free registers in this example, the compiler selected register r11 to hold a copy of this value. When there are no available registers, the old stack pointer is saved in the stack. The next instruction, leaq, subtracts 8 to the stack pointer to allocate space for the aps local stack variable. The following instruction, andq, aligns the stack pointer to 8 bytes. In this particular case, it is not necessary to perform such alignment if we assume

```
inline fn __load4(reg u64 ap) → reg u64[4] {
  inline int i;
  reg u64[4] a;
  for i=0 to 4
  { a[i] = [ap + 8*i]; }
  return a;
}

inline fn __store4(reg u64 ap, reg u64[4] a) {
  inline int i;
  for i=0 to 4
  { [ap + 8*i] = a[i]; }
}

export fn add4(reg u64 ap bp) {
  reg u64[4] a b;
  stack u64 aps;
  aps = ap;
  a = __load4(ap);
  b = __load4(bp);
  a = __add4(a,b);
  ap = aps;
  __store4(ap, a);
}
```

```
add4:
        movq    %rsp, %r11
        leaq    -8(%rsp), %rsp
        andq    $-8, %rsp
        movq    %rdi, (%rsp)     # aps = ap
        movq    (%rdi), %rax     # a = __load4(ap);
        movq    8(%rdi), %rcx
        movq    16(%rdi), %rdx
        movq    24(%rdi), %rdi
        movq    (%rsi), %r8      # b = __load4(bp);
        movq    8(%rsi), %r9
        movq    16(%rsi), %r10
        movq    24(%rsi), %rsi
        addq    %r8, %rax        # a = __add4(a,b);
        adcq    %r9, %rcx
        adcq    %r10, %rdx
        adcq    %rsi, %rdi
        movq    (%rsp), %rsi     # ap = aps;
        movq    %rax, (%rsi)     # __store4(ap, a);
        movq    %rcx, 8(%rsi)
        movq    %rdx, 16(%rsi)
        movq    %rdi, 24(%rsi)
        movq    %r11, %rsp
        ret
```

Figure 4.17: Jasmin export function.

that the initial value of the stack pointer was already aligned at 8 or 16 bytes. However, if there was a local variable declared as stack u8, then leaq would be used to subtract 9 to rsp and, in this case, the alignment would be necessary. If stack u256 variables were used in the implementation, then the stack would be aligned at 32.

After the local stack frame setup is done, the ap pointer is stored in the stack, and two register arrays with four positions each are loaded from memory, using the auxiliary function __load4. If, for instance, function __load4 was frequently used two times in row, an additional function __load4x2 could be implemented: it would call __load4 two times and would return both arrays. Such hypothetical call to __load4x2, "a, b = __load4x2(ap,bp)", would yield exactly the same assembly code that is shown by figure 4.17. Jasmin programming language is designed to promote reusability.

After both arrays are loaded into registers, the addition is performed, the pointer ap is restored, and the result of the addition is stored in ap. Before returning, the function restores rsp. No caller registers were necessary in this example but, if they were, they would be saved into the stack at the beginning of the function's execution and restored at the end of the

function, more precisely, right before rsp is restored. The Jasmin compiler generates the epilogue and prologue of external functions automatically.

4.4.3 Local Functions

The declaration of local functions is similar to other functions' declarations, but a specific keyword, similar to inline or export, is not necessary:

```
#[returnaddress="stack"]
fn function_name ( arguments ) → return_types { function_block }
```

There is, however, an optional annotation that allows to specify if the return address of the function is to be placed in the stack. If the annotation is not present, the function uses one register to hold the return address and the compilation fails if there are no registers available. In contexts where registers are scarce and where freeing up one register translates in a performance penalty, this annotation should be used.

arguments should only contain reg variables. These can have any word type, such as u8 or u256, but arrays of those are not supported yet. It is also possible for the arguments to have the bool or ptr types. A ptr variable can be used, for instance, to provide a memory reference to a stack array defined in the caller's function scope. stack and inline variables are not allowed to be declared in the argument list.

The return_types section can contain any type that is also allowed in the arguments list. As restrictions, it is possible to return a reg ptr reference as long as the returned variable is also in the argument list. This means that it is not possible to return a reference (reg ptr) of a locally defined stack array. Regarding the function_block, any Jasmin code is allowed, including calls to other inline or local functions.

Local functions do not follow any calling convention and the compiler does not introduce any unexpected memory spill. This means that it is the programmer's responsibility to store any live reg variables in the stack frame (if there are not enough registers to hold all live variables) before the call to a local function happens. A call to a local function consists in an execution of a jmp instruction to a code section where some inputs are expected in certain registers, and the return address is placed in the stack frame or in a register. If the local function uses locally defined stack variables, including arrays, the caller decrements the stack pointer, rsp, before the call, to allocate the necessary amount of space. After the local function returns, it is also the caller that resets the stack pointer. Since local functions do not follow any calling convention, its compilation depends on how they are used.

Consider a scenario where there is an export function e1 that expects two reg u64 arguments, a and b, which contain memory pointers. Given that export functions follow the System V

calling convention, a and b are expected to be in registers rdi and rsi, respectively. Also consider that there is an additional local function f1 which expects one reg u64 as argument, containing a memory pointer, and that it performs some computation with the pointed values. If function e1 performs the following two calls to f1, "f1(a); f1(b);", the compilation fails. The first call to f1 imposes the restriction that f1 should be compiled expecting its argument in register rdi and the second in rsi. Both restrictions cannot be satisfied simultaneously. There are different solutions for this problem depending on how the function e1 is expected to perform: if we assume that the complete function_block of e1 is composed by just those two calls to f1, then it can be changed to "f1(a); a = b; f1(a);". This way, function f1 expects its argument to be in register rdi and, since b is copied into a, the second call is compatible with the first. Another possible solution would be to declare a local reg variable and use it in both calls, "r=a; f1(r); r=b; f1(r);".

Figure 4.18 presents an example of a local function, _add4, that receives two memory pointers ap and bp, loads them into register arrays with __load4x2, performs the addition of those using the previously presented inline function __add4, from figure 4.14, and stores the result using __store4, from figure 4.17. The implementation of __load4x2 is not shown, but it was discussed in the final paragraph of subsection 4.4.2, and it consists in two subsequent calls to __load4, also from figure 4.17.

The presented example also includes an export function, add4, that receives four memory pointers, ap, bp, cp and dp, and performs two calls to _add4, the first with ap and bp, and the second with cp and dp. According to what was previously discussed regarding the restrictions when using local functions, the approach of introducing additional variables to hold copies of the local's function inputs is used. A short version of the assembly code is also presented. Lines that contain the comment "#..." are omitted code sections, which are not included for brevity.

At first sight, add4 seems to require 15 registers to be compiled, 4 for the arguments, 2 for the rp and sp variables, which are used to hold copies of _add4's arguments, 1 for the return address of _add4, and 8 for the register arrays a and b. However, if we consider the fact that ap and bp are no longer live after being copied to rp and sp, the compiler is able to merge them, and only 13 registers are effectively used.

Since reserved registers are necessary, the compiler first decrements the stack pointer rsp and saves registers rax, rbx, rbp and r14 in the stack. These instructions are not shown. Registers r12 and r15 are not used. The return address is loaded in register r14 and the jmp to the label that indicates the starting point of _add4 is performed. Since rp and sp were merged into ap and bp, _add4 expects its arguments to be in registers rdi and rsi. After _add4 returns, cp and dp are copied into registers rdi and rsi. The second call is performed and, after returning, the top level function add4 restores all caller registers and resets the stack pointer.

```
fn _add4(reg u64 ap bp)
{
  reg u64[4] a b;

  a, b = __load4x2(ap, bp);
  a = __add4(a,b);
  __store4(ap, a);
}

export fn add4(reg u64 ap bp cp dp)
{
  reg u64 rp sp;

  rp = ap; sp = bp;
  _add4(rp, sp);

  rp = cp; sp = dp;
  _add4(rp, sp);
}
```

```
add4:
  # ...                       # stack setup and...
  # ...                       # ...save caller registers
  leaq Ladd4$2(%rip), %r14    # return address in r14
  jmp L_add4$1                # first call to _add4
Ladd4$2:
  movq %rdx, %rdi             # rp = cp
  movq %rcx, %rsi             # sp = dp
  leaq Ladd4$1(%rip), %r14    # return address in r14
  jmp L_add4$1                # second call to _add4
Ladd4$1:
  # ...                       # restore caller registers
  ret
L_add4$1:
  movq  (%rdi), %rax          # a, b = __load4x2(ap, bp);
  # ...
  addq  %r11, %rax            # a = __add4(a,b);
  # ...
  movq  %rax, (%rdi)          # __store4(ap, a);
  # ...
  jmp   *%r14                 # return to caller
```

Figure 4.18: Jasmin local function.

For this particular case, 11 instructions related with the stack setup and saving/restoring caller registers were necessary. A different approach to remove those instructions could be: instead of loading the values pointed by bp into array b, rewrite function __add4 to directly use pointer bp to access its elements to perform the addition. For instance, "cf, r[i] += [bp + 8*i] + cf;". This approach only requires 9 registers and no caller registers are necessary. This alternative version would require less 15 instructions when compared to the one presented: 11 related with stack operations and an additional 4 moves from the loading of bp into b.

4.5 Jasmin and Easycrypt

This section describes the embedding of Jasmin in EasyCrypt. The embedding is used to prove the correctness of reference implementations, equivalence between reference and optimized (often vectorized) implementations, and check the constant-time property.

4.5.1 Overview of EasyCrypt

EasyCrypt [BDG+13] is a general-purpose proof assistant for proving properties of probabilistic computations with adversarial code. It has been used for proving security of several primitives and protocols [BGLB11, BGHB11, BCLS15, ABB+17b].

EasyCrypt implements program logics for proving properties of imperative programs. In contrast to common practices (which use shallow or deep embeddings), the language and program logics are hard-coded in EasyCrypt — and thus belong to the Trusted Computing Base. The main program logics of EasyCrypt are Hoare logic, and relational Hoare logics — both operate on probabilistic programs but we only used their deterministic fragments. The relational Hoare logic allows to relate two programs, possibly with very different control flow. In particular, the rule for loops allows to relate loops that do not do the same number of iterations. This is essential for proving correctness of optimizations based on vectorization, or when the optimization depends on the input message length.

The program logics are embedded in a higher-order logic which can be used to formalize and reason about mathematical objects used in cryptographic schemes and also to carry meta-reasoning about statements of the program logic. Automation of the ambient logic is achieved using multiple tools, including custom tactics (e.g. to reason about polynomial equalities) and back-end to SMT solvers. We have found it convenient to add support for proof by computation. This tool allows users to perform proofs simply by (automatically) rewriting expressions into canonical forms.

4.5.2 Design Choices and Issues

Rather than building a verified verification infrastructure on top of the Coq formalization of the language (a la VST [App14]), we opt for embedding Jasmin into EasyCrypt. We choose this route for pragmatic reasons: EasyCrypt already provides infrastructure for functional correctness and relational proofs, and achieves reasonable levels of automation. On the other hand, embedding Jasmin in EasyCrypt leads to duplicate work, since we must define an embedding of the Jasmin language into EasyCrypt.

Although we already have an encoding of Jasmin into Coq, we cannot reuse this encoding for two reasons: first, we intend to exploit maximally the verification infrastructure of EasyCrypt, so the encoding should be fine-tuned to achieve this goal. Second, the Coq encoding uses dependent types, which are not available in EasyCrypt. However, these are relatively simple issues to resolve, and the amount of duplicate work is largely compensated by the gains of using EasyCrypt for program verification.

Hacspec is a specification language for cryptographic primitives [BKS18, MKB21]. It is designed to be easily integrated with other formal specification languages such as EasyCrypt,

Coq, and F*. In this context, there is ongoing work to build a formally verified embedding of Hacspec in Coq for optimized implementations [HLMS22]. This new approach can be used to extend the formal verification infrastructure of Jasmin.

4.5.3 Embedding Jasmin in EasyCrypt

The native language of EasyCrypt provides control-flow structures that perfectly match those in Jasmin, including if, while and call commands. This leaves us with two issues: 1) to encode the semantics of all AMD64 instructions (including SIMD) in EasyCrypt; and 2) to encode the memory model of Jasmin in EasyCrypt.

Instruction Semantics

Our formalization of AMD64 instructions aims at being both readable and amenable to building a library of reusable properties over the defined operations, in particular over SIMD instructions. The first step is to define a generic theory for words of size k, with the usual arithmetic and bit-wise operations. The semantics of arithmetic operations are based on two injections (signed and unsigned) into integers and arithmetic modulo 2^k. For bit-wise operations, we rely on an injection to Boolean arrays of size k. Naturally a link between both representations (int and Boolean array) is also created, which allows proving for example that shifting a word $n \ll i$ is the same as multiplying it by $\mathsf{to\_uint}\ 2^i$.

Scalar AMD64 operations are formalized using the theory for words, and useful lemmas about the semantics of these instructions are also proved as auxiliary lemmas. For example, the formalizations of shl and shr permit proving lemmas like $\mathsf{shl}\ x\ i \oplus \mathsf{shr}\ x\ (k-i) = \mathsf{rol}\ x\ i$, under appropriate conditions on i.

The semantics of SIMD instructions rely on the theories for 128/256 bit words, but the semantics must be further refined to enable viewing words as arrays of sub-words, which may be nested (e.g., instruction vpshufd sees 256-bit words as two 128-bit words, each of them viewed as an array of sub-words). To ease this kind of definition, we have defined a bijection between words and arrays of (sub-)words of various sizes. Then vector instructions are defined in terms of arrays of words.

Memory Model

EasyCrypt does not provide the notion of pointer natively. We rely on the concept of a global variable in EasyCrypt, which can be modified by side effects of procedures, to emulate the global memory of Jasmin and the concept of pointer to this memory. A dedicated EasyCrypt library defines abstract type global_mem_t equipped with two basic operations for load mem[p] and store mem[p ← x] of one byte, as follows:

```
type address = int.
type global_mem_t.
op "_[_]" : global_mem_t → address → W8.t.
op "_[_←_]" : global_mem_t → address → W8.t → global_mem_t.
axiom get_setE m x y w : m[x ← w][y] = if y = x then w else m[y].
```

From this basic axiom we build the semantics of load and store instructions for various word sizes. The Jasmin memory library then defines a single global variable Glob.mem of type global_mem_t, which is accessible to other EasyCrypt modules and is used to express pre-conditions and post-conditions on memory states.

Soundness

The embedding of a Jasmin program into EasyCrypt is sound, provided the program is safe. This is because the axiomatic model of Jasmin in EasyCrypt is intended to be verification-friendly, and assuming safety yields much simpler verification conditions and considerably alleviates verification of functional and equivalence properties. This assumption is perfectly fine, since Jasmin programs are automatically checked for safety before being compiled and embedded into EasyCrypt.

As potential future work, it would be interesting to make our safety checker certifying, in the sense that it automatically produces a proof of equivalence between the Coq and EasyCrypt semantics of Jasmin programs — technically, this would be achieved by formalizing in Coq a simpler semantics for safe programs, and proving automatically that the two semantics coincide for safe programs. The coincidence between the simpler semantics in Coq and the Jasmin semantics would still need to be argued informally.

Reusable EasyCrypt libraries

In the course of writing correctness proofs for our use cases we have created a few EasyCrypt libraries that will be useful for future projects. Significant effort was put into enriching the theories of words in order to facilitate proofs of computations over multi-precision representations. Concretely, a theory was created that permits tight control over the number of used bits within a word (a form of range analysis), which is crucial for dealing with delayed carry operations and establishing algebraic correctness via the absence of overflows.

The central part of this library is generic with respect to the number of limbs, so that operations like addition and school-book multiplication can be handled in a fully generic way (here we rely heavily on the powerful ring theory in EasyCrypt). When dealing with constructions such as Poly1305, base on primes which are very close to a power of 2, this means that only the prime-specific modular reduction algorithm needs special treatment. Moreover, this theory was fine-tuned to interact well with SMT provers, enabling the automatic discharge

of otherwise tedious to prove intermediate results.

4.5.4 Verification of Timing Attack Mitigations

The EasyCrypt embedding of Jasmin programs is instrumented with leakage traces that include all branching conditions plus all accessed memory addresses (this also includes array indexes since an access in a *stack* array will generate a memory access at the assembly level). It is then possible to check that the secret inputs do not interfere with this leakage trace in the classical sense that, for all public-equivalent input states $x_1 \equiv_{\mathsf{pub}} x_2$, the program will give rise to identical leakages $\ell_1 = \ell_2$. Figure 4.19 shows an example of the generated instrumented EasyCrypt code.

```
inline fn store2(
  reg u64 p,
  reg u64[2] x)
{
  [p + 0] = x[0];
  [p + 8] = x[1];
}
```

```
proc store2 (p:u64, x:u64 array2) : unit = {
  var aux: u64;
  leakages ← LeakAddr [0] :: leakages;
  aux ← x[0];
  leakages ← LeakAddr [to_uint (p + 0)] :: leakages;
  Glob.mem ← storeW64 Glob.mem (to_uint (p + 0)) aux;
  leakages ← LeakAddr [1] :: leakages;
  aux ← x[1];
  leakages ← LeakAddr [to_uint (p + 8)] :: leakages;
  Glob.mem ← storeW64 Glob.mem (to_uint (p + 8)) aux;
}
```

Figure 4.19: EasyCrypt code (right) instrumented for constant-time verification of a Jasmin program (left).

Pleasingly, EasyCrypt tactics developed to deal with information flow-like properties handle the particular equivalence relation associated with so-called *constant-time* security extremely effectively. In particular, EasyCrypt provides the sim tactics which is specialized on proving equivalence of programs sharing the same control flow (which is the case here, as we are reasoning about two executions of the same program). The tactic is based on dependency analysis and also proved very useful in justifying simple optimizations like spilling, which do not affect the control flow. In the case of constant-time verification there is a very interesting side-effect to the dependency analysis performed by this tactic: it is able to infer sufficient conditions (equality of input variables) that guarantee equality of output variables. When applied to constant-time verification this means that, when this tactic is successful (which was the case for our use-cases) the user just needs to check if the inferred set of variables are all public. We note that performing this kind of analysis at the assembly level is usually hard. We take advantage of the fact that Jasmin provides a high-level semantics that makes

it suitable for verification; in particular, the clear separation between memory, stack variables and stack arrays at source level greatly simplifies the problem.

Chapter 5

Verified Jasmin Implementations

During the previous chapter, the Jasmin programming language was extensively discussed, mainly from a practical point of view, to fulfill two goals. The first is to share the programming experience that the author of this thesis acquired over the last few years while using and contributing to the development and improvement of the Jasmin framework: it introduces the available types and features and their impact on the produced assembly code. It is the first extended tutorial of the Jasmin programming language, primarily designed for anyone interested in becoming a Jasmin programmer. The second goal is to provide the background material for the current chapter.

Some of the cryptographic implementations developed and formally verified using the Jasmin framework are presented throughout this chapter. The implementations included in this chapter are: ChaCha20 and Poly1305 [ABB+20a], and Curve25519. Regarding Curve25519, several implementations were proposed during the first Jasmin publication [ABB+17a], back in 2017. This publication included the performance results for three different Curve25519 Jasmin implementations. Two of these were obtained by automatically translating existing qhasm code into Jasmin code. The performance of the corresponding qhasm and Jasmin implementations was comparable, ranging between 147k and 149k CPU cycles depending on how many limbs were used to represent field elements. The third Curve25519 Jasmin implementation was optimized (by taking advantage of the Jasmin features), and it took roughly 144k cycles. All measurements were taken from an Intel Skylake. This chapter focuses on new Curve25519 implementations, with better performance when compared to the previous ones.

Regarding this chapter's structure, each primitive has its section. Each section provides an overview of the corresponding primitive, then describes how the implementations were designed and optimized, and compares the developed implementations with formally-verified and non-verified alternative implementations. For Curve25519, an overview of the proof is presented.

5.1 ChaCha20

ChaCha is a family of stream ciphers based on the Salsa20 family of 256-bit stream ciphers [B+08, Ber08]. Instances of the Salsa20 family are usually prefixed with the number of performed rounds, for instance, Salsa20/20 performs 20 rounds, Salsa20/12 performs 12 rounds, and Salsa20/8 performs 8 rounds. The corresponding ChaCha instances are ChaCha20, ChaCha12, and ChaCha8. As a general intuition, the main differences between Salsa20 and ChaCha are some rearrangements in the order on how some operations are performed during the round computation, to improve security, and also a rearrangement in the internal state, to improve the performance of SIMD implementations. In this section, we discuss the 20-round instance of the ChaCha family, ChaCha20, as specified in TLS 1.3.

5.1.1 Algorithm Overview

ChaCha20 defines an algorithm that expands a 256-bit key into 2^{96} key streams (each stream is associated with a 96-bit nonce) each consisting of 2^{32} blocks (each 64-byte block is associated with a counter value). Intuitively, it defines a procedure to transform an initial state into a key stream block. The initial state is constructed using a 256-bit key k (seen as eight 32-bit words), a 96-bit nonce n (seen as three 32-bit words), a 32-bit counter b and four 32-bit constants c. The initial state can be seen as the following matrix, where on the left-hand side we show the arrangement of 32-bit words and on the right-hand side we show the matrix entry numbering.

$$\begin{array}{cccc} c & c & c & c \\ k & k & k & k \\ k & k & k & k \\ b & n & n & n \end{array} \qquad \begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array}$$

The state transformation is based on the following operation, QuarterRound, that acts upon four 32-bit words at a time, here designated as a, b, c, and d, and where $+$ means addition modulo 2^{32}, $\oplus$ means the exclusive or operation, and rol corresponds to the rotate left operation by the number of bits specified by the second operand:

QuarterRound(a, b, c, d):

$a \leftarrow a + b$; $d \leftarrow d \oplus a$; $d \leftarrow \mathsf{rol}\ d\ 16$;
$c \leftarrow c + d$; $b \leftarrow b \oplus c$; $b \leftarrow \mathsf{rol}\ b\ 12$;
$a \leftarrow a + b$; $d \leftarrow d \oplus a$; $d \leftarrow \mathsf{rol}\ d\ 8$;
$c \leftarrow c + d$; $b \leftarrow b \oplus c$; $b \leftarrow \mathsf{rol}\ b\ 7$;
Return (a, b, c, d)

ChaCha20's state transformation is frequently implemented as a loop of 10 double rounds, totaling 20 rounds. A double round is composed by a column round and a diagonal round. Each round is composed by 4 quarter rounds. Given a state s, where, for instance, $s0$ and $s15$ represent the elements 0 and 15 of the above state matrix, DoubleRound could be defined as follows:

DoubleRound(s):
  $(s0, s4, s8, s12) \leftarrow$ QuarterRound$(s0, s4, s8, s12)$;
  $(s1, s5, s9, s13) \leftarrow$ QuarterRound$(s1, s5, s9, s13)$;
  $(s2, s6, s10, s14) \leftarrow$ QuarterRound$(s2, s6, s10, s14)$;
  $(s3, s7, s11, s15) \leftarrow$ QuarterRound$(s3, s7, s11, s15)$;

  $(s0, s5, s10, s15) \leftarrow$ QuarterRound$(s0, s5, s10, s15)$;
  $(s1, s6, s11, s12) \leftarrow$ QuarterRound$(s1, s6, s11, s12)$;
  $(s2, s7, s8, s13) \leftarrow$ QuarterRound$(s2, s7, s8, s13)$;
  $(s3, s4, s9, s14) \leftarrow$ QuarterRound$(s3, s4, s9, s14)$;
Return (s)

To produce a 64-byte keystream block, each 32-bit element of the initial state is added, modulo 2^{32}, with the transformed state (after 10 iterations of DoubleRound). For subsequent blocks, the counter is incremented.

5.1.2 ChaCha20 Implementations

Three Jasmin implementations of ChaCha20 were developed, one relying only on scalar operations (no vectorization), and two others that rely on the AVX and AVX2 extensions. All three Jasmin implementations implement the same interface:

```
export fn chacha20(
  reg u64 output plain,
  reg u32 len,
  reg u64 key nonce,
  reg u32 counter)
  { // ...
  }
```

The first argument, output, provides a pointer to a memory region where the ciphertext is written; plain is a pointer to the plaintext; len is the length in bytes of both the output and plain and it is limited to 4GiB of data; key and nonce are also pointers for the 256-bit secret-key and 96-bit nonce, respectively; counter contains the initial counter. Given that the developed implementations were tested and measured using the SUPERCOP framework,

which assumes that the nonce for this primitive has 8 bytes instead of the 12 bytes that are specified by TLS 1.3, a wrapper, written in C code, was developed to circumvent this incompatibility. As such, the presented performance data in this section includes a small overhead related to one call to the memset function, to set 4 (12-8) bytes as zero, and one call to memcpy to copy 8 bytes from the nonce into a local array. In practice, this overhead is negligible and can be ignored.

ChaCha20 Scalar

The scalar implementation of ChaCha20 relies on standard AMD64 registers. The most adequate data type to represent the state is u32 given that additions and rotations are performed modulo 2^{32}. Ideally, all 16 32-bit variables from the state would be in register variables. Since this is not possible, as there are only 15 registers available in this context (rsp is reserved), some memory spills are necessary during the state transformation. Nonetheless, it is possible to represent the state as a register array with 16 positions, as long as not all of its elements are live simultaneously.

QuarterRound performs 4 executions of the same pattern of operations: an addition, an exclusive or, and a rotation. This pattern can be implemented as follows:

```
inline fn line(reg u32[16] s, inline int a b c r) → reg u32[16] {
  s[a] += s[b];
  s[c] ^= s[a];
  _, _, s[c] = #ROL_32(s[c], r);
  return s;
}
```

The presented function, line, receives a register array s and, considering that each line operation only uses 3 variables, 3 additional indexes are also provided to this function, a, b, and c, to indicate which elements should be read or updated. The last argument, an inline int variable r, holds the number of bits to rotate s[c]. As such, QuarterRound procedure can be implemented as follows:

```
inline fn quarter_round(reg u32[16] s, inline int a b c d) → reg u32[16] {
  s = line(s, a, b, d, 16);
  s = line(s, c, d, b, 12);
  s = line(s, a, b, d, 8);
  s = line(s, c, d, b, 7);
  return s;
}
```

This implementation of quarter_round receives and returns the state array s and, considering

that quarter_round implicitly uses 4 variables at a time, s[a], s[b], s[c], and s[d], it is necessary to guarantee that these variables are live whenever this function is called. Both line and quarter_round are defined as inline functions and, as such, whenever this function is called the corresponding assembly instructions (of line) are inlined.

The next function is the previously discussed DoubleRound. This function performs two rounds at a time, corresponding to 8 quarter_round calls, and it will be called in the context of a while loop that iterates a local variable, from 0 to 10, for instance. As a side note, using a for loop to call DoubleRound is not a viable solution since the resulting assembly code would not be very cache friendly given that for loops are fully unrolled. If we assume the loop counter to be in a stack variable during the execution of DoubleRound, there are 15 registers available to implement this function. Since all of the 16 variables from the state are updated during a DoubleRound, it is necessary to spill at least one variable from the state into the stack.

A good candidate to be stored in the stack is $s15$, or s[15] in the Jasmin implementation. For simplicity, the next two descriptions of DoubleRound were stripped down to be as succinct as possible for the purpose of the next discussion.

DoubleRound − *Step 0* :	DoubleRound − *Step 1* :
QuarterRound$(s0, s4, s8, s12)$;	QuarterRound$(s0, s4, s8, s12)$;
QuarterRound$(s1, s5, s9, s13)$;	QuarterRound$(s2, s6, s10, s14)$;
QuarterRound$(s2, s6, s10, s14)$;	$Swap(14, 15)$
$Swap(14, 15)$	QuarterRound$(s1, s5, s9, s13)$;
QuarterRound$(s3, s7, s11, s15)$;	QuarterRound$(s3, s7, s11, s15)$;
QuarterRound$(s0, s5, s10, s15)$;	QuarterRound$(s1, s6, s11, s12)$;
$Swap(15, 14)$	QuarterRound$(s0, s5, s10, s15)$;
QuarterRound$(s1, s6, s11, s12)$;	$Swap(15, 14)$
QuarterRound$(s2, s7, s8, s13)$;	QuarterRound$(s2, s7, s8, s13)$;
QuarterRound$(s3, s4, s9, s14)$;	QuarterRound$(s3, s4, s9, s14)$;

Step 0: If we consider that $s15$ is in stack at the beginning of DoubleRound's execution, then we can *Swap* it with $s14$, or, more precisely, copy $s14$ into the stack and load $s15$ into a register variable. Then, two quarter-rounds later to be precise, $s15$ can be copied again into the stack and $s14$ loaded into a register variable. Since $s15$ is held in the stack at the start and end of this function's execution, an invariant is established, and given that this function is going to be used inside a while loop, in a context where there are no free registers, this linearity in the function's behavior increases the chances of a successful compilation from Jasmin to assembly[1].

[1]The compilation of a semantically valid Jasmin program may fail if the register allocator is not able to find a valid register allocation, §4.1.1.

Step 1: There are, however, two aspects that can be improved in the previous approach. First, it is possible to change the order of the QuarterRound calls to increase the distance from the first load of $s15$ until the time it is first used: $s15$ is loaded from the stack; and then QuarterRound$(s3, s7, s11, s15)$ is executed; after the addition of $s3$ with $s7$, $s15$ is XOR-ed with $s3$. While performing this rearrangement it would also be nice to always have a pair number of subsequent QuarterRound calls, to introduce the HalfRound function, which can perform two calls to QuarterRound. Such rearrangement can be obtained by swapping the second call of QuarterRound with the third, and the sixth with the seventh. The *Swaps* occur after the second and sixth call.

Step 2: The next step to optimize this implementation's performance is to *merge* the executions of two QuarterRounds: the line function first performs an addition; the result of the previous addition is an input of the next exclusive or; and, finally, this last result is rotated in-place. Two of these instructions depend on the previous instruction's output and are limited by the previous instruction's latency; to reduce this effect, it is possible to rearrange the code to perform two line operations at a time, one from each QuarterRound. Such function can be called double_line:

```
inline fn double_round(reg u32[16] s, stack u32 s15) → reg u32[16], stack u32 {
  stack u32 s14;
  s = half_round(s, 0, 4, 8, 12,   2, 6, 10, 14);
  s14 = s[14]; s[15] = s15;
  s = half_round(s, 1, 5, 9, 13,   3, 7, 11, 15);
  s = half_round(s, 1, 6, 11, 12,  0, 5, 10, 15);
  s15 = s[15]; s[14] = s14;
  s = half_round(s, 2, 7, 8, 13,   3, 4, 9, 14);
  return s, s15;
}
```

double_round implements the optimizations discussed in *Step 1* and it can also be used for *Step 2*, if the implementation of half_round internally uses double_line. Given that s[15] is not live at the start of this function's execution, a separate variable that contains this *missing* element, declared as stack u32 s15, is provided as an argument. Both arguments, the state s and the auxiliary variable s15, must be returned at the end of this function's executions for the corresponding values to be updated in the context of the caller function. In practice, no assembly instructions are generated as a consequence of using the return statement. A local stack variable s14 is also declared to hold the corresponding element of the state. By using this strategy, only 4 instructions that interact with memory addresses are executed for each DoubleRound.

The remaining functions of this scalar implementation of ChaCha20, such as functions that

read the plaintext, or write the ciphertext, from, and into, the corresponding memory regions, are not discussed in this document. Regarding the performance analysis of the discussed implementations, figure 5.1 presents a comparison between the discussed approaches: *Step0*, *Step1*, and *Step2*. All data presented in this figure was acquired using the same *complete* implementation of ChaCha20, and only the double_round function was swapped in between different measurement runs. This was possible since that all three implementations of double_round share the same API and require the same set of registers. The plot includes the number of CPU cycles per encrypted byte and the implementations were measured for lengths in between 128 and 16384 bytes.

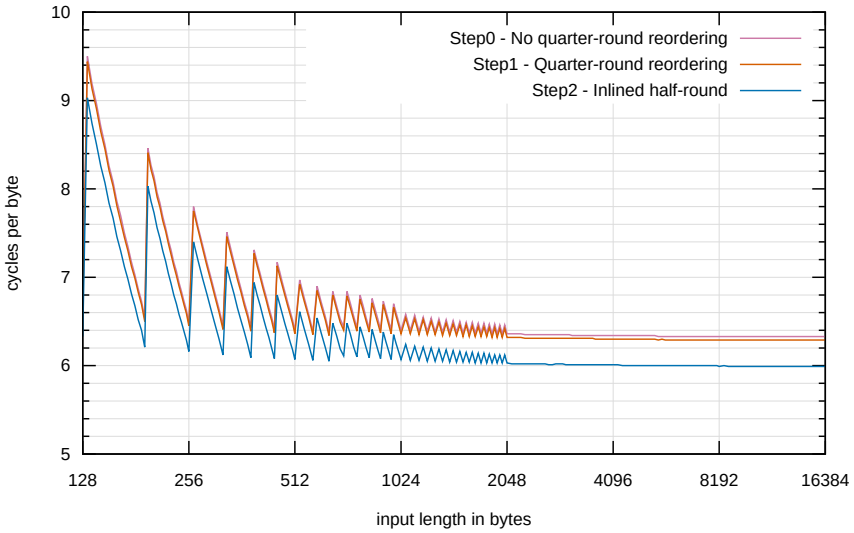

Figure 5.1: ChaCha20: performance comparison for scalar implementations.

In figure 5.1 it is possible to observe a small performance difference between *Step 0* and *Step 1*. For 16KiB, *Step 0* takes 6.33 cycles per byte (cpb) and *Step 1* 6.29 cpb. The average difference for these two steps is 0.04 cpb, with a standard deviation of 0.0055.

This difference, however, cannot be directly associated with the exact position of the *Swaps* between register and stack variables: some alternative implementations of *Step 1*, with loads/stores being performed at different locations while preserving the QuarterRound's calling order, were measured; the performance of these alternative implementations is consistent with the *Step 1* implementation's performance, which makes sense given that these implementations were benchmarked in an out-of-order CPU that internally rearranges instructions before executing them. The most probable cause for this small but consistent difference, is the reordering of the QuarterRound*s* calls.

The fastest implementation in this set is *Step 2*, taking 5.99 cpb for 16KiB plaintexts. When compared with *Step 1*, this implementation is 0.3 cpb, or roughly 5%, faster. An interesting idea in this context would be to explore how different permutations of QuarterRound calls, and

corresponding memory spills, impact the performance and, perhaps even more interestingly, to study how the Jasmin compiler could rearrange the code automatically.

ChaCha20 AVX and AVX2

The AVX and AVX2 implementations use the data types u128 and u256, respectively, which can be used to perform computations over 4 and 8 32-bits words, also respectively. To represent the state matrix in these implementations, two different internal representations are used: horizontal and vertical. The horizontal representation [GG13], is used by Jasmin implementions to compute small inputs: in AVX for inputs from 1 to 128 bytes; in AVX2 from 1 to 256 bytes. The vertical representation, implemented in OpenSSL, is used for the remaining input lengths. The discussion will follow by first describing each representation in the context of AVX and how it can be extended for AVX2.

Consider the following description of the ChaCha's state matrix. In comparison to what was presented in §5.1.1, "Algorithm Overview", in this description some commas and curly braces are used to highlight the fact that each matrix line corresponds to an array of 4 elements that can be loaded into a u128 register.

$$\begin{matrix} \{c_1, & c_2, & c_3, & c_4\} \\ \{k_1, & k_2, & k_3, & k_4\} \\ \{k_5, & k_6, & k_7, & k_8\} \\ \{b, & n_1, & n_2, & n_3\} \end{matrix} \qquad \begin{matrix} \{c_1, & c_2, & c_3, & c_4\} \\ \{k_1, & k_2, & k_3, & k_4\} \\ \{k_5, & k_6, & k_7, & k_8\} \\ \{b+1, & n_1, & n_2, & n_3\} \end{matrix}$$

In the AVX implementation, if the input length is less or equal to 64 bytes, then only 4 u128 registers are necessary (left). For inputs lengths from 65 to 128 bytes, 4 additional u128 registers are required (right). The general idea for the AVX2 implementation is exactly the same but the previous intervals are multiplied by 2: 4 u256 registers to process inputs from 1 to 128 bytes (left and right lines loaded in one u256 register); 8 u256 registers for inputs from 129 to 256 bytes, with the counters with $b+2$ and $b+3$.

Regarding how to compute a DoubleRound with the horizontal representation: the first 4 quarter rounds, corresponding to the column round, can be easily computed in parallel because all operations are performed over columns; for the diagonal round this is not true, and the state must be rearranged upon entry and exit of the diagonal round code.

Consider the case where, in the context of AVX and to produce a 64-byte block, we have the state loaded into an array "reg u128[4] s". s[1] is initially loaded with $\{k_1, k_2, k_3, k_4\}$, s[2] with $\{k_5, k_6, k_7, k_8\}$, and s[3] with $\{b+1, n_1, n_2, n_3\}$. After the first column round is performed — and if we continue to use the same set of names (k_n,n_n,...) for convenience — before the diagonal round can be executed we need to rotate s[1] to $\{k_2, k_3, k_4, k_1\}$, s[2] to

$\{k_7, k_8, k_5, k_6\}$, and s[3] to $\{n_3, b+1, n_1, n_2\}$. This can be done with the vpshufd instruction. After the diagonal round is computed, these elements should be rotated to the original position.

An implementation of the linex4 and round functions, in the context of AVX and to process inputs with up to 64 bytes, is presented next. Each linex4 function performs the equivalent to 4 line calls from the scalar implementation, one from each QuarterRound.

Since there are four different rotations that can be performed during the execution of a round (by 16, 12, 8, and 7), this behavior is encapsulated by an inline function rotate, which is called by linex4. The rotation by 16 and 8 can be performed using a single instruction, vpshufb, while the rotation for 12 and 7 is implemented using 2 shifts, one to the left (by r) and another to the right (by 32-r), and an additional instruction, which can be an exclusive-or, to combine these intermediate values. rotate is implemented using a sequence of ifs (that are resolved during compilation) to decide which approach is used depending on the value of r. The round function is the same for the column and diagonal round given that the state is rearranged in between rounds.

```
inline fn linex4(reg u128[4] s,
  inline int a b c r) → reg u128[4] {
  s[a] +4u32= s[b];
  s[c] ^= s[a];
  s = rotate(s, c, r);
  return s;
}
```

```
inline fn round(reg u128[4] s) → reg u128[4] {
  s = linex4(s, 0, 1, 3, 16);
  s = linex4(s, 2, 3, 1, 12);
  s = linex4(s, 0, 1, 3,  8);
  s = linex4(s, 2, 3, 1,  7);
  return s;
}
```

The discussion so far was focused on scenarios where just 4 (AVX) registers are used. For AVX2, the corresponding linex4 function is called linex8 and the implementation is very similar: for instance, +8u32= is used instead of +4u32=. When 8 registers are required, double_linex4 and double_linex8 functions can be implemented, following the same idea that was previously discussed for the scalar implementation. The complete implementation of the rounds function is presented next. It uses a reg u64 variable to hold the loop counter and, in between the column and diagonal rounds, the state is shuffled.

```
inline fn rounds(reg u128[4] s) → reg u128[4] {
  reg u64 c;
  c = 0;
  while(c < 10)
  { s = round(s);
    s = shuffle(s);
    s = round(s);
    s = reverse_shuffle(s);
    c += 1;
```

```
  }
  return s;
}
```

The layout of the vertical representation (larger inputs) is presented next in the context of AVX. For AVX2, 8 blocks, instead of just 4, can be computed simultaneously. Each element is replicated in each line except for the counter b, which is incremented according to the corresponding block. Given that bellow matrix has 16 lines, 16 variables are required to hold the state. Ideally, all state would be allocated in register variables during the execution of the 20 rounds and, since there are 16 registers available in this context (xmm/ymm depending on the implementation), and not just 15 as in the scalar implementation of ChaCha20, it seems reasonable to assume that it should be possible.

$$
\begin{array}{llll}
\{c_1, & c_1, & c_1, & c_1\} \\
\{c_2, & c_2, & c_2, & c_2\} \\
\{c_3, & c_3, & c_3, & c_3\} \\
\{c_4, & c_4, & c_4, & c_4\} \\
\{k_1, & k_1, & k_1, & k_1\} \\
\{k_2, & k_2, & k_2, & k_2\} \\
\{..., & ..., & ..., & ...\} \\
\{k_8, & k_8, & k_8, & k_8\} \\
\{b+3, & b+2, & b+1, & b\} \\
\{n_1, & n_1, & n_1, & n_1\} \\
\{n_2, & n_2, & n_2, & n_2\} \\
\{n_3, & n_3, & n_3, & n_3\}
\end{array}
$$

There is, however, a small detail which makes such approach unfeasible: to perform a rotation by 12 and 7, which is implemented using two shifts and an exclusive-or, a temporary register variable to hold an intermediate value of this computation is necessary. For this reason, the same approach that was taken in the scalar implementation regarding the spill of s15 is also taken for the AVX and AVX2 implementations.

An advantage of the vertical representation, when compared to the horizontal representation, is that it is not necessary to shuffle the state between the column and diagonal rounds. Regarding the disadvantages: after the blocks' processing is completed, which includes the execution of the 20 rounds and the addition with the initial state, the contents of the state should be XOR-ed with the plaintext; given that all sequences of 64 bytes corresponding to the different blocks are distributed across 16 different variables, the state must be transformed into a horizontal representation before the final XOR can happen. The transformation of the state from the vertical to the horizontal representation can be done using a sequence of the vpnpck and vperm instructions. An alternative solution, but not as efficient, would be to store

the state in memory, and then access the corresponding 32-bits to perform the XOR. Overall, the benefits of using this representation for larger messages out-weights the disadvantages.

In total, there are 3 possible approaches for each vectorized implementation: using 4 registers to hold the state in a horizontal representation to calculate 1 (AVX) or 2 (AVX2) blocks at a time; using 8 registers in a horizontal representation to calculate 2 (AVX) or 4 (AVX2) blocks at a time by following a *double line* approach; using all available registers in a vertical representation to compute 4 (AVX) or 8 (AVX2) blocks at a time.

With the goal of analyzing, individually, how each approach performs for different input lengths, figures 5.2 (AVX) and 5.3 (AVX2) are presented. Earlier in this subsection, it was already unveiled in which input lengths intervals each approach is used, and these plots present a justification for the chosen intervals. The scalar implementation is also included for comparison purposes in a light gray line given that the next discussion is mainly focused on vectorized implementations.

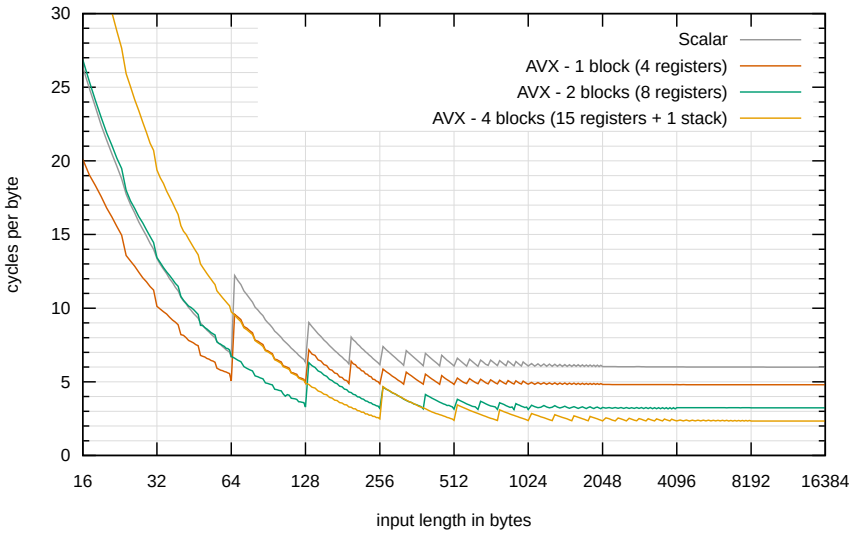

Figure 5.2: ChaCha20: comparison of different vectorization approaches (AVX).

Figure 5.2 shows that the fastest approach for input lengths up to 64 bytes is the one which uses 4 registers and the horizontal representation (4.80 cpb / 16KiB). From 65 to 128 bytes, using 8 registers and the horizontal representation is the best option (3.23 cpb / 16KiB). From 128 bytes onwards, the vertical implementation performs better, although it is very similar to the 8 register version from 256 to 384 bytes (2.34 cpb / 16KiB).

For comparison, the best scalar implementation executes at 5.99 cpb for 16KiB. Overall, vectorized implementations are faster when compared to the scalar implementation (in benchmarking/optimal conditions), which is only competitive in this scenario for inputs up to 64 bytes. As such, the final AVX implementation of ChaCha20 contains three different code

paths, for three different input lengths intervals. These intervals can be easily changed if necessary.

Regarding the performance analysis for different approaches in the context of an AVX2 implementation, the lines from figure 5.3 exhibit a very similar pattern (which is expected) when compared to the corresponding ones from figure 5.2. The overall performance is better: for inputs of 16KiB, the 4, 8, and 15 register approaches take 2.45, 1.60, and 1.20 cpb, respectively. When compared to the corresponding AVX implementations, AVX2 implementations are roughly twice as fast, which is also expected if we consider that twice as many blocks are being produced using roughly the same number of CPU instructions.

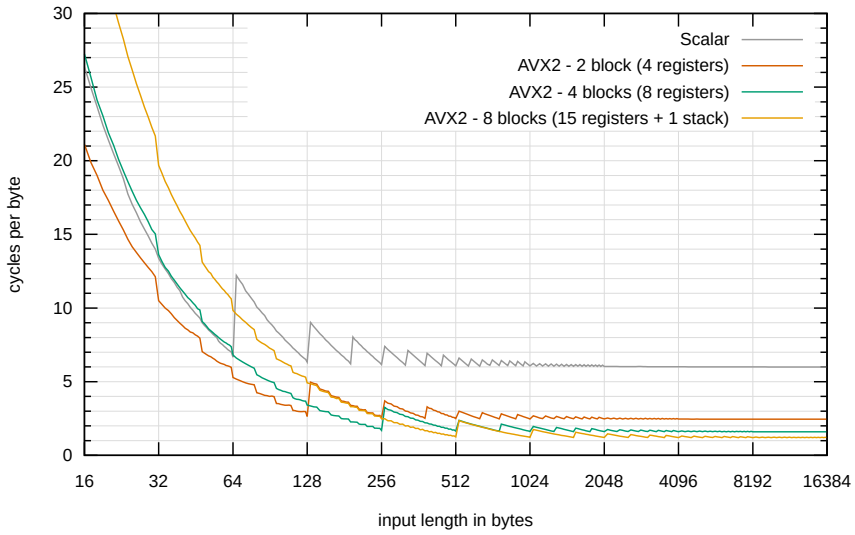

Figure 5.3: ChaCha20: comparison of different vectorization approaches (AVX2).

As a side-note, an additional experiment (related to the discussion from 4.2.2) was done: all movdqu instructions from "AVX2 - 8 blocks" implementation that do not interact with external memory regions were manually replaced in the generated assembly file by movdqa instructions (since the SUPERCOP API does not enforce that the input and output pointers should be aligned); in total, there were 4 movdqu inside the rounds function (executed 10 times for each set of 8 blocks, or 512 bytes), and 21 movdqu related to the transformation from the vertical to the horizontal representation (executed once every 8 blocks); for 16 KiB, the best reported CPU cycle count (best median of three different runs) was 19662 for movdqu code and 19660 for the movdqa version; this difference is negligible and it is most probably noise-related. Nonetheless, it would be interesting to conduct a series of experiments, including different micro-architectures, to measure the impact of different alignment setups in this context. For instance, what would be the performance penalty of having non-aligned addresses and exactly in what circumstances is movdqa advantageous in this context.

To conclude, the final AVX2 implementation of ChaCha20 also includes three different code

paths. The code path that is executed depends on the input length, which can also be easily adjusted to satisfy the requirements of different environments.

5.1.3 Performance Evaluation

In this section we compare the ChaCha20 Jasmin implementations with other open-source implementations, formally verified at source-level and non-verified. Figure 5.4 compares the performance of the previously described Jasmin implementations, Scalar, AVX, and AVX2, with three different implementations from the HACL* project [ZBPB17, PBP+20]: Hacl_ChaCha20, which is comparable to the Jasmin Scalar implementation of ChaCha20; Hacl_ChaCha20_Vec128 which corresponds to a vectorized implementation that uses AVX; and Hacl_ChaCha20_Vec256 for the AVX2 version. The evaluated HACL* code was obtained from the project's GitHub repository[2] and the considered distribution was gcc64-only, compiled using gcc version 9.3.0[3].

The comparison with HACL* compiled with CompCert is not included in this section for two main reasons: first, in the HACL* project under the dist directory, a distribution targeting CompCert cannot be found and, although the Makefile from c89-compatible distribution could most probably be adapted to make the compilation with CompCert possible, following such an approach would yield non-efficient code since CompCert is not (yet) suited for this context[4]; second, HACL* AVX and AVX2 implementations would be excluded from this analysis given that (currently) there is no version of this compiler that supports AVX/AVX2 instructions.

Overall, and for 16KiB inputs, the performance of Jasmin and HACL* implementations is similar: the non-vectorized implementations execute in 6.21 (HACL*) and 5.99 (Jasmin) cpb; vectorized implementations execute at 2.39/1.26 cpb (HACL*) and 2.34/1.20 cpb (Jasmin), for AVX/AVX2, respectively. For inputs whose lengths are less than, for instance, 512 bytes, the difference between implementations' performance is more noticeable, with Jasmin implementations being slightly faster.

If we observe the line that corresponds to the HACL* AVX2 implementation, for instance, it is possible to notice some similarities with one of the lines from figures 5.2 and 5.3 (where the performance impact of different representations was studied): the vertical representation is, most probably, the only representation that is used by HACL* AVX/AVX2 implementations for all input lengths. This, however, should not be interpreted as a problem of any kind given that such an approach also has its advantages: implementing just one code path implies that

[2] https://github.com/project-everest/hacl-star/tree/a50b659d11953dadd8d84ec5df25203cec1a746b

[3] CFLAGS set as -Ofast -march=native -mtune=native -m64 -fwrapv -fomit-frame-pointer -funroll-loops

[4] In [ABB+20a], a plot containing the performance of HACL* ChaCha20's implementation, compiled with CompCert 3.4, can be found. That implementation, at that point in time, took roughly 13 cpb, for 16KiB inputs.

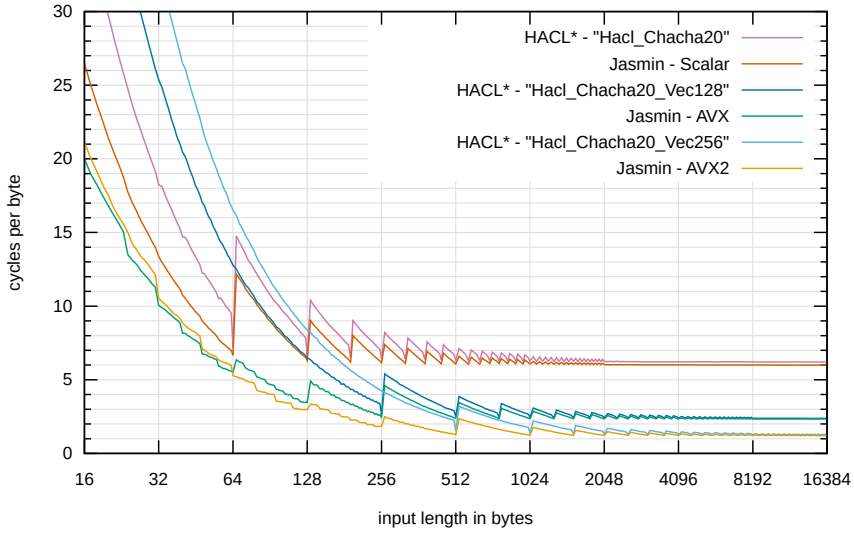

Figure 5.4: ChaCha20: comparison of Jasmin and HACL* implementations.

the resulting assembly code will be smaller and thus, more cache-friendly. For small messages, the scalar implementation can always be used. On the other hand, the advantage of using different strategies to improve the performance in specific input length intervals is that it can benefit the overall performance of systems where the length of the encrypted messages is small on average.

Figure 5.5 compares the developed Jasmin implementations with OpenSSL[5]. OpenSSL contains four different implementations of ChaCha20: an implementation in C, which can be accessed using the no-asm option during configuration; an implementation written in assembly that uses the standard x86_64 registers and that is comparable to the scalar implementation in Jasmin; an AVX implementation written in assembly; and an AVX2 implementation also written in assembly. Two static libraries were compiled, using the provided Makefiles without any modifications: one for a non-assembly library; and another that included the mentioned assembly implementations. To access the different assembly implementations, the environment variable OPENSSL_ia32cap was set according to the benchmark being run.

Regarding the performance for 16KiB inputs: 6.30/5.99/2.38/1.21 cpb for OpenSSL implementations; and 5.99/2.34/1.20 for Jasmin implementations. The plot line regarding the ChaCha20 C implementation of OpenSSL (6.30 cpb) was not included in this figure for simplicity. Regarding Jasmin and OpenSSL scalar implementations (5.99 cpb), which perform at roughly the same speeds for all (measured) input lengths, the main difference between these implementations is that OpenSSL implementation requires 4 mov instructions that read/write from/to the stack frame every round (or 8 moves for a double round) while the Jasmin implementation performs just 4 mov instructions during a double round, at the

[5] https://github.com/openssl/openssl/tree/bc8c36272067f8443f875164831ce3a5a739df3f

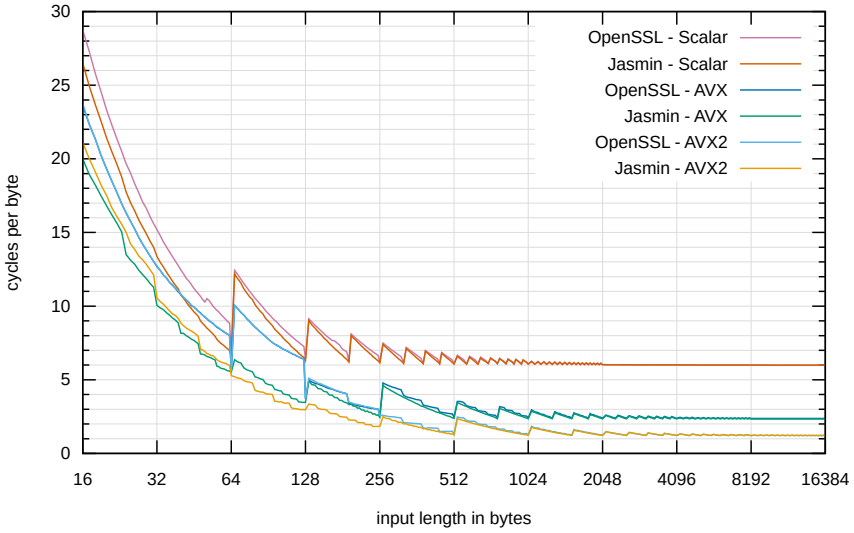

Figure 5.5: ChaCha20: comparison of Jasmin and OpenSSL implementations.

expense of having the loop counter in a stack variable. This type of difference, which in some scenarios can be observed for small inputs, easily become diluted/negligible as the inputs lengths grow due to the way that modern processors work.

In the context of vectorized implementations, AVX and AVX2, it can be observed that OpenSSL implementations were also optimized for small inputs: the difference in performance for Jasmin vs OpenSSL implementations is notoriously smaller when compared with Jasmin vs HACL*. The gap between these implementations is slightly bigger for inputs in the range of 64 to 128 bytes. The OpenSSL lines corresponding to the AVX and AVX2 only diverge for messages larger than 256 bytes which indicates that, most probably, the same code path is being shared until then. The major conclusion from this analysis is that the Jasmin programming language allows to write code that is comparable with the best performing implementations which, more frequently than not, can be found in OpenSSL, without having to write the code directly or very close to assembly.

Figure 5.6 compares the AVX2 Jasmin implementation with other open-source AVX2 implementations. This figure includes measurements from the Usuba project[6] [MD19], which has several C implementations of ChaCha20: the reported measurements are for the fastest implementation. It also includes measurements from libsodium[7], which was compiled using the provided Makefile without any modifications to it. The AVX2 implementation used by libsodium can also be found in SUPERCOP toolkit dolbeau/avx2. Another implementation from SUPERCOP included in this figure is moon/avx2/64.

Overall, and for inputs with 16KiB, the performance is very similar: 1.35/1.29/1.26/1.20 cpb

[6] https://github.com/DadaIsCrazy/usuba/tree/5ecbf056d7d8bd77462d29309c85f897f36d0f49
[7] https://github.com/jedisct1/libsodium/tree/a016aea61214668827e18c6278ac25b0bbc98ca5

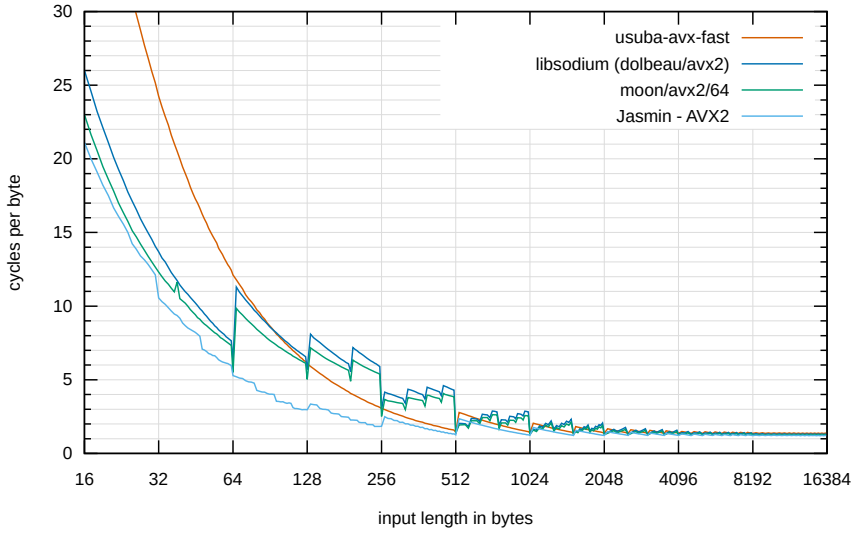

Figure 5.6: ChaCha20: comparison of AVX2 implementations.

for Usuba/libsodium/moon/avx2/64/Jasmin. The Jasmin implementation is slightly faster for inputs lengths between 64 and 256 bytes. Additionally, libsodium can be easily compiled with CompCert (version 3.9): 15.17 cpb for 16KiB inputs.

5.2 Poly1305

Poly1305 [Ber05b] is a message authentication code that is used together with ChaCha20 as one of the two ciphersuites recommended in the TLS 1.3 RFC. Poly1305 is a one-time authenticator (the key should be used once) that allows the sender to attach a cryptographic tag t to a transmitted message m. The receiver of the message should be able to derive the same session key k autonomously, and recompute the tag on the received message. If the tags match, the receiver is assured that only the sender could have transmitted it, provided k is secret and authentic.

5.2.1 Algorithm Overview

Poly1305 takes a 32-byte one-time key k and a message m and it produces a 16-byte tag t. The key k is seen as a pair (r, s), in which each component is treated as a 16-octet little-endian number, with the following format restrictions: octets $r[3]$, $r[7]$, $r[11]$ and $r[15]$ should have their top 4 bits cleared, whereas octets $r[4]$, $r[8]$ and $r[12]$ are required to have their two lower bits cleared. For the purpose of this section we assume that $k = (r, s)$ is generated as a pseudorandom 256-bit string, after which r is *clamped* to its correct format.

To authenticate a message m, it is split into 16-byte blocks m_i, for $i \in [1, 2, \dots]$. Each

block m_i is then converted into a 129-bit number b_i by reading it as a 16-byte little-endian value and then setting the 129-th bit to one. The authenticator t is computed by sequentially accumulating each such number into an initial state $a_0 = 0$ according to the following formula: $a_i = (a_{i-1} + b_i) \times r \pmod{p}$, for $i \in [1, 2, \dots]$ and where $p = 2^{130} - 5$ is prime. Finally, the secret key s is added to the accumulator (over the integers) and the tag t is simply the lowest 128 bits of the result serialized in little-endian order. The choice of p is crucial for optimization, as it is close to a power of 2: modular reduction can be performed by first reducing modulo 2^{130} and then adjusting the result using a simple computation that depends on the offset 5.

5.2.2 Poly1305 Implementations

Three Jasmin implementations of Poly1305 were developed, one relying only on scalar operations and two others for the AVX and AVX2 extensions. All three Jasmin implementations implement the same interface:

```
export fn poly1305(reg u64 out in _inlen _k)
{ // ...
}
```

Regarding the function arguments: out specifies a pointer to write the computed tag t (16 bytes); in is a pointer to the input data with length _inlen; and _k points to the 256-bit secret-key. Given that _inlen and _k are allocated in registers rdx and rcx, respectively, and these registers are required by the multiplication and some shift instructions, also respectively, in the context of the scalar implementation (which is also used by vectorized implementations for small inputs as it will be discussed next) the following assignments must be performed at the beginning of this function's execution in order for those values to be allocated in different registers:

```
export fn poly1305_ref3(reg u64 out in _inlen _k)
{ reg u64 inlen k;
  inlen = _inlen;
  k = _k;
  // ...
```

In the context of integrating the Jasmin implementations in SUPERCOP, and given that the presented Jasmin interface is compatible with the API that is specified by it, it was not necessary to write a C wrapper, in contrast to what was required for the ChaCha20 implementations.

Poly1305 Scalar

This implementation of Poly1305 uses 64-bit variables. From the algorithm's overview: we need to compute $a_i = (a_{i-1} + b_i) \times r \pmod p$ where a corresponds to the accumulator, b corresponds to the message block (16 bytes) being consumed with the upper (129th) bit set to one, and r corresponds to the first 16 bytes (*clamped*) of the secret key k.

r requires two 64-bit variables and h, which initially corresponds to the expression $(a_0 + b_1)$, requires three 64-bit variables. The third 64-bit variable of h is initially set to one. Given this, the expression $h \times r \pmod p$ is equivalent to $(h[0] + 2^{64}h[1] + 2^{128}h[2]) \cdot (r[0] + 2^{64}r[1]) \pmod p$. This multiplication can be unrolled as follows:

$$\begin{array}{lcll} 2^0 & \cdot & (h[0] \cdot r[0]) & + \\ 2^{64} & \cdot & (h[0] \cdot r[1] + h[1] \cdot r[0]) & + \\ 2^{128} & \cdot & (h[1] \cdot r[1] + h[2] \cdot r[0]) & + \\ 2^{192} & \cdot & (h[2] \cdot r[1]) \pmod p & \end{array}$$

Given that p is a prime close to a power of two, it is possible to take advantage of the fact that $a \cdot 2^{130} + b \pmod{2^{130} - 5}$ is congruent with $a \cdot 5 + b \pmod{2^{130} - 5}$. Additionally, given that the first two bits of $r[1]$ are set to zero due to *clamping*, making it a multiple of 4, for some multiplications that involve $r[1]$ we take advantage of the fact that, for these cases, $a \cdot 2^{130} \cdot 2^{-2} + b \pmod{2^{130} - 5}$ is congruent with $a \cdot 5/4 + b \pmod{2^{130} - 5}$. The previous expression, to compute $h * r$, can be rearranged as follows:

$$\begin{array}{lcllll} 2^0 & \cdot & (h[0] \cdot r[0] & + & h[1] \cdot r[1] \cdot 5/4) & + \\ 2^{64} & \cdot & (h[0] \cdot r[1] + h[1] \cdot r[0] & + & h[2] \cdot r[1] \cdot 5/4) & + \\ 2^{128} & \cdot & (h[2] \cdot r[0]) \pmod p & & & \end{array}$$

Considering that $(r[1] \cdot 5/4)$ is frequently used (twice for each 16-byte block of the input) this value can be precomputed.

The implementation of clamp, which initializes r and precomputes $(r[1] \cdot 5/4)$ is presented next. This function receives a pointer k and returns a register array r with 3 elements. It starts by loading 16 bytes from memory into the first two positions of r, and then the corresponding bits are cleared. It then initializes r[2] with r[1] >> 2 + r[1], which corresponds to $r[1] \cdot 5/4$ given that the first two bits of r[1] are zero.

```
inline fn clamp(reg u64 k) → reg u64[3] {
  reg u64[3] r;
  r[0] = [k + 0];
  r[1] = [k + 8];
  r[0] &= 0x0ffffffc0fffffff;
```

```
  r[1] &= 0x0ffffffc0ffffffc;
  r[2] = r[1];
  r[2] >>= 2;
  r[2] += r[1];
  return r;
}
```

Figure 5.7 presents the Jasmin implementation of the mulmod function which allows to efficiently compute $h * r$. It receives two register arrays as arguments: h, which contains the accumulator $(a_{i-1} + b_i)$; and r, which is initialized by the clamp function. The multiplication result is returned in h.

To perform this computation some auxiliary variables are used: t0, t1, and t2 to hold 64-bit intermediate values corresponding to the 2^0, 2^{64}, and 2^{128} positions, respectively; and rax and rdx for holding the multiplication result between two 64-bit values. As a note, although the underlying multiplication instruction uses registers with the same name (rax and rdx), such variable names do not interfere in any way with the compiler's register allocation.

The order in which multiplications are performed is: $(h[2] \cdot r[1] \cdot 5/4)$; $(h[2] \cdot r[0])$; $(h[0] \cdot r[0])$; $(h[1] \cdot r[0])$; $(h[1] \cdot r[1] \cdot 5/4)$; and $(h[0] \cdot r[1])$. The first two multiplications from this sequence (which involve $h[2]$) can be performed using the imulq instruction, given that the result fits in a 64-bit register variable without loss of precision. The remaining multiplications require the 128-bit multiplication, mulq, which reads its inputs from rax and an arbitrary register, and write the result into rax and rdx for the lower and upper part of the result, respectively. The presented multiplication order was chosen to minimize the number of additions. By using this approach, the multiplication can be performed with: 2 imulq; 4 mulq; 9 movq; 3 addq; and 3 adcq assembly instructions. In comparison, the approach taken by OpenSSL, which implements this multiplication routine using a different order $((h[0] \cdot r[1])$; $(h[0] \cdot r[0])$; $(h[1] \cdot r[0])$; $(h[1] \cdot r[1] \cdot 5/4)$; $(h[2] \cdot r[1] \cdot 5/4)$; $(h[2] \cdot r[0]))$, requires one more addq and also one more adcq instruction.

After the multiplication is completed, with the results being held by t0, t1, and h[2], it is necessary to reduce the result modulo 2^{130}. The first step is to take the upper 62 bits of h[2], multiply this value by 5 and add it to the first limb, t0. To achieve this, a mask with the first two bits as zero is placed in h[0] and then h[0] is AND-ed with h[2]. Effectively, h[0] contains the value of the upper 62 bits of h[2] multiplied by 4. After this, it is only necessary to add the value corresponding to the upper 62 bits of h[2] to h[0] to complete the multiplication by 5. Overall, the multiplication instruction is expensive and should be avoided if possible. The upper bits of h[2] are cleared and, at this point, h[2] contains at most the value 3, given that only the first two bits can be set. After the last carry-chain addition is performed to include the values from t0 and t1 in the computation, h[2] is at most 4. When compared to OpenSSL, the reducing method is exactly the same, and it is only differentiated by the position of the

mask initialization which, in OpenSSL, happens some instructions before when compared to Jasmin (which can also be replicated in Jasmin if necessary).

```
inline fn mulmod(reg u64[3] h r) → reg u64[3] {
  reg bool cf;
  reg u64 t0 t1 t2;
  reg u64 rax rdx;

  t2 = r[2];
  t2 *= h[2]; // (h[2] * r[1] * 5/4)
  h[2] *= r[0]; // (h[2] * r[0])

  rax = r[0];
  rdx, rax = rax * h[0];
  t0 = rax; t1 = rdx; // (h[0] * r[0])

  rax = r[0];
  rdx, rax = rax * h[1]; // (h[1] * r[0])

  cf, t1 += rax;
  _ , h[2] += rdx + cf;

  rax = r[2];
  rdx, rax = rax * h[1]; // (h[1] * r[1] * 5/4)

  h[1] = rdx;
  h[1] += t2;
  t2 = rax;
  rax = r[1];
  rdx, rax = rax * h[0]; // (h[0] * r[1])

  cf, t0 += t2; // 2**0
  cf, t1 += rax + cf; // 2**64
  _ , h[2] += rdx + cf; // 2**128

  // reduce
  h[0] = 0xfffffffffffffffc;
    t2 = h[2];
    t2 >>= 2;
  h[0] &= h[2];
  h[0] += t2; // h[0] = (h[2] / 4) * 5
  // <=> (h[2] / 4) * 4 + (h[2] / 4)

  h[2] &= 0x03; // 2 bits left in h[2]

  cf, h[0] += t0;
  cf, h[1] += t1 + cf;
  _ , h[2] +=  0 + cf; // 0 <= h[2] <= 4

  return h;
}
```

Figure 5.7: Poly1305: mulmod function in Jasmin.

After all the input message blocks are consumed by sequentially calling mulmod, a final reduction needs to happen to ensure that the final result is reduced modulo $2^{130} - 5$. From mulmod, it can be assumed (and proved) that $h[2]$ is at most 4 and that $h[0]$ and $h[1]$ can be any value up to $2^{64} - 1$. h needs to be reduced if it is greater of equal to $2^{130} - 5$. Given that at most one bit after the 130th position can be set in h, we can start by calculating $h + 1 \cdot 5$ and check if this value has its 131st bit set. If it has, then $h + 5 \pmod{2^{130}}$ is the reduced value, if it does not, h is already in bounds.

The presented description of the scalar implementation of Poly1305 synthesizes the most relevant aspects of it and it is mostly focused on how the polynomials are represented and how the multiplication routine of Poly1305 can be implemented efficiently. To better understand what are the gains related to the proposed mulmod implementation, in which the

multiplications are performed using a different order when compared to the best alternative implementations, figure 5.8 shows the performance of two (complete) Jasmin implementations of Poly1305 that only differ on mulmod: the top line corresponds to an Jasmin implementation that uses a version of mulmod that exactly matches the code run by OpenSSL; the bottom line corresponds to an Jasmin implementation that includes mulmod from figure 5.7. For 16KiB inputs, these implementations take 1.17 and 1.04 cpb. The difference between the discussed approaches can also be observed during subsection 5.2.3, which includes a broader performance comparison.

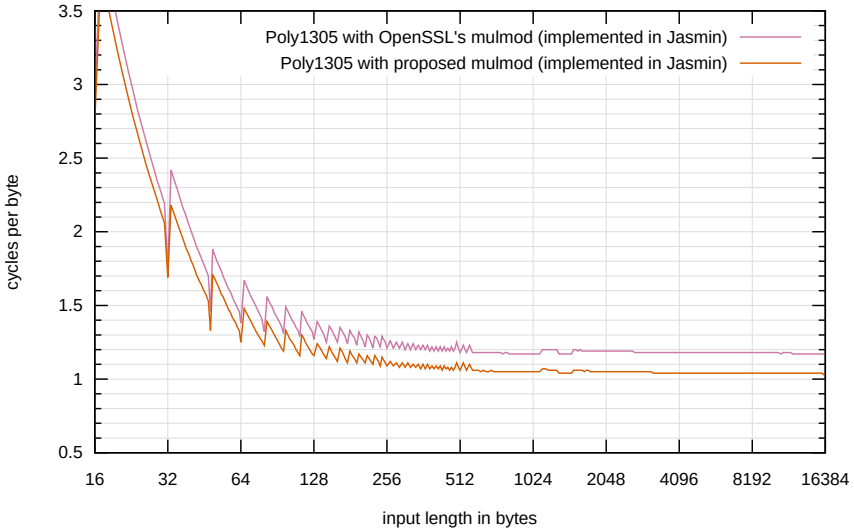

Figure 5.8: Poly1305: comparison of different mulmod implementations.

Poly1305 AVX and AVX2

The main idea for vectorizing Poly1305 is somehow similar to the one that was described for ChaCha20: multiple blocks can be computed at once. For this scenario, however, it is necessary to take into consideration that the computation of the *next* accumulator depends on the previous one: the accumulator a_i can be computed as $(a_{i-1} + b_i) \times r \pmod{p}$, where b_i corresponds to the message block being processed with the upper bit set to one. As an example, consider a case where there are exactly 8 message blocks (128 bytes) to process. The previous formula can be unrolled as:

$$\begin{aligned} &b_1 \cdot r^8 + b_2 \cdot r^7 + b_3 \cdot r^6 + b_4 \cdot r^5 + \\ &b_5 \cdot r^4 + b_6 \cdot r^3 + b_7 \cdot r^2 + b_8 \cdot r \pmod{p} \end{aligned}$$

Depending on the chosen strategy to compute multiple blocks at once, this formula can be written differently. For instance, consider the case where we would like to rearrange this

formula to process 2 message blocks simultaneously, which is useful for an AVX implementation:

$$\begin{aligned}&((((b_1 \cdot r^2 + b_3) \cdot r^2 + b_5) \cdot r^2) + b_7) \cdot r^2 \quad + \\ &((((b_2 \cdot r^2 + b_4) \cdot r^2 + b_6) \cdot r^2) + b_8) \cdot r \quad (\text{mod } p)\end{aligned}$$

Intuitively, if H_1 represents an array with two polynomials and it is initialized with the first two blocks of the message $\{b_1, b_2\}$ (to avoid an initial multiplication by zero), then, to process inputs whose lengths are multiple of $16 \cdot 2$, it is possible to define a (vectorized) procedure that computes $H_n = H_{n-1} \cdot R^2 + B_n$, where R^2 is defined as $\{r^2, r^2\}$ and B_n contains the next set of two blocks $\{b_{2n-1}, b_{2n}\}$. Generically, such procedure can be iteratively called to consume the input message. To finish the computation, H_n should be multiplied by $\{r^2, r\}$. The previous formula can be slightly adjusted to optimize the performance of the AVX implementation:

$$\begin{aligned}&((0 \cdot r^4 + b_1 \cdot r^2 + b_3) \cdot r^4 + b_5 \cdot r^2 + b_7) \cdot r^2 \quad + \\ &((0 \cdot r^4 + b_2 \cdot r^2 + b_4) \cdot r^4 + b_6 \cdot r^2 + b_8) \cdot r \quad (\text{mod } p)\end{aligned}$$

This formula allows to process four blocks at once. The multiplication of r^4 by zero is included to make the pattern more clear and it can be easily avoided in the implementation. Using the previous notation, the computation can be described as $H_n = H_{n-1} \cdot R^4 + \{b_{4n-3}, b_{4n-2}\} \cdot R^2 + \{b_{4n-1}, b_{4n}\}$, where R^4 is defined as $\{r^4, r^4\}$.

Similarly to what was first discussed in the context of processing two blocks simultaneously, the same pattern can be used for an AVX2 context, to enable the computation of four blocks without unrolling:

$$\begin{aligned}&(b_1 \cdot r^4 + b_5) \cdot r^4 \quad + \\ &(b_2 \cdot r^4 + b_6) \cdot r^3 \quad + \\ &(b_3 \cdot r^4 + b_7) \cdot r^2 \quad + \\ &(b_4 \cdot r^4 + b_8) \cdot r \quad (\text{mod } p)\end{aligned}$$

In this case, the computation is $H_n = H_{n-1} \cdot R^4 + B_n$, where R^4 is defined as $\{r^4, r^4, r^4, r^4\}$ and with B_n corresponding to $\{b_{4n-3}, b_{4n-2}, b_{4n-1}, b_{4n}\}$. H_1 can also be initialized with the first set of four blocks and a final multiplication by $\{r^4, r^3, r^2, r\}$ must occur.

So far the discussion was focused on how to process a number of blocks that is multiple of two, for AVX, and four, for AVX2. For messages with different lengths, the remaining blocks, or partial blocks, can be consumed by the previously discussed scalar implementation. For this to be possible, after the multiplication by $\{r^2, r\}$ or $\{r^4, r^3, r^2, r\}$ is done, the polynomials contained in H are added and reduced to a representation that is compatible

with the previously discussed scalar implementation. The execution continues using the scalar implementation.

Regarding the multiplication of polynomials, the Jasmin instructions available for this context are #VPMULU and #VPMULU_256, for AVX and AVX2, respectively, which correspond to vpmuludq in assembly. This instruction allows to multiply the low 32-bit integers from each 64-bit element. For instance, given two reg u128 variables a and b initialized with (4u32){0,a1,0,a2} and (4u32){0,b1,0,b2}, the statement "c = #VPMULU(a, b)", where c is also a reg u128 variable, allows to compute (2u64){a1*b1,a2*b2}. As such, it is not possible to use 64-bit limbs as in the scalar implementation. Both vectorized implementations of Poly1305 use 26-bit limbs and five limbs are required to represent 130-bit words. Using four limbs is not an option given that the maximum limb size for this context would be 32-bits if there was an efficient way of performing carry-chain additions using AVX/AVX2 instructions. Using six limbs would also not provide any significant advantage considering that the result of multiplying two 26-bit words can be represented within 52-bits, which already provides some leeway (64-52 bits) to perform a significant number of additions before reaching a point where an overflow can occur. Additionally, using five 26-bit limbs has the advantage that 130 is a multiple of the limb size and, as such, the reduction is simplified. For any two polynomials a and b, represented with five 26-bit limbs, the multiplication $a \times b \pmod p$ can be unrolled as follows:

$$
\begin{aligned}
2^{0} &\cdot (a[0]\cdot b[0] + a[1]\cdot b[4]\cdot 5 + a[2]\cdot b[3]\cdot 5 + a[3]\cdot b[2]\cdot 5 + a[4]\cdot b[1]\cdot 5) &+\\
2^{26} &\cdot (a[0]\cdot b[1] + a[1]\cdot b[0] + a[2]\cdot b[4]\cdot 5 + a[3]\cdot b[3]\cdot 5 + a[4]\cdot b[2]\cdot 5) &+\\
2^{52} &\cdot (a[0]\cdot b[2] + a[1]\cdot b[1] + a[2]\cdot b[0] + a[3]\cdot b[4]\cdot 5 + a[4]\cdot b[3]\cdot 5) &+\\
2^{78} &\cdot (a[0]\cdot b[3] + a[1]\cdot b[2] + a[2]\cdot b[1] + a[3]\cdot b[0] + a[4]\cdot b[4]\cdot 5) &+\\
2^{104} &\cdot (a[0]\cdot b[4] + a[1]\cdot b[3] + a[2]\cdot b[2] + a[3]\cdot b[1] + a[4]\cdot b[0]) \pmod p
\end{aligned}
$$

In the context of AVX two multiplications ($a \times b \pmod p$) can be computed simultaneously and, in AVX2, four. For instance, the previously discussed H is declared as a reg u128[5] in the AVX implementation and as reg u256[5] in AVX2. Hence, each set of five 26-limbs is *vertically* disposed. In the previous unrolled multiplication, it is possible to observe that $b[0]$ is never multiplied by 5. Intuitively b can be seen as a placeholder for r^4, r^3, r^2, and r^1. As such, it is not necessary to precompute the multiplication between the first limb of these polynomials and 5: the corresponding arrays can be declared, for instance, as reg u128[4]. The declaration of mulmod for the AVX2 implementation is shown next. It expects a register array h which represents the accumulator, a register array b which is already initialized with the four message blocks (64 bytes). It also receives r which, in this case, corresponds to four copies of r^4 or $\{r^4, r^3, r^2, r\}$ and rx5, which corresponds to four copies of $r^4 \cdot 5$ or $\{r^4 \cdot 5, r^3 \cdot 5, r^2 \cdot 5, r \cdot 5\}$, minus the first element which is not necessary. This function computes (h + b) * r, which is a small reordering in comparison to the generic

formula previously presented, to avoid the first multiplication by zero.

```
inline fn add_mulmod_avx2(
  reg u256[5] h b,
  stack u256[5] r,
  stack u256[4] rx5) → reg u256[5]
{ //...
}
```

Figure 5.9 compares the vectorized approaches being discussed so far. The optimized version of the scalar implementation is also included for completeness. The y-axis of the plot was purposely truncated at 5 cpb to improve the plot analysis. The scalar implementation is the one that performs the best for small inputs, up until 256 bytes, and it takes 1.04 cpb for 16KiB inputs. For messages with more than 256 bytes, the AVX2 implementation, which internally uses the scalar implementation for the remaining blocks, is the fastest, taking 0.53 cpb for 16KiB.

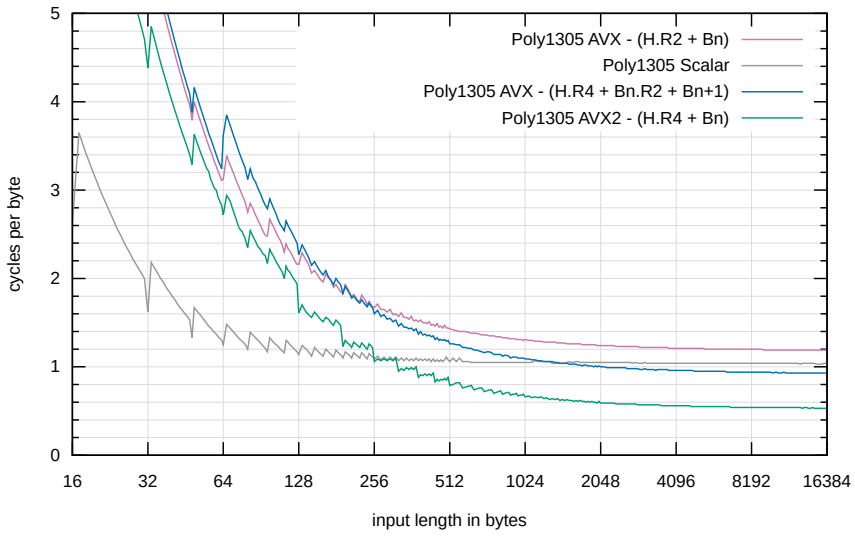

Figure 5.9: Poly1305: comparison of Jasmin implementations.

To avoid the initial multiplication by zero, the AVX2 implementation requires at least 8 blocks (or 128 bytes) of input and, after that, it consumes four blocks in each iteration. Because of this, the presented performance data for this AVX2 experimental implementation corresponding from roughly 30 bytes up until 126 bytes are essentially the overhead of precomputing and initializing the required variables for the AVX2 implementation, which are not used because there are not enough blocks. It is also possible to observe a change in the line pattern due to this. The unrolled approach for the AVX implementation performs better than the first discussed approach: 1.19 vs 0.93 cpb / 16KiB. The final AVX implementation only uses vectorization for inputs that are greater than 1024 bytes (for short messages scalar

is used) and the threshold for the AVX2 implementation was set at 256 bytes. Similarly to the presented ChaCha20 implementations, these values can be easily updated if required.

5.2.3 Performance Evaluation

In this section we compare the Poly1305 implementations with other open-source implementations, formally verified and non-verified. Figure 5.10 compares the performance of the previously described Jasmin implementations, Scalar, AVX, and AVX2, with one implementation from ValeCrypt [BHK+17] project and two others from HACL* [ZBPB17, PBP+20]: Poly1305_Vale, which is comparable to the Jasmin Scalar implementation of Poly1305; Hacl_Poly1305_Vec128 which corresponds to a vectorized implementation that uses AVX; and Hacl_Poly1305_Vec256 for the AVX2 version. The code was downloaded from the project's GitHub repository[8] and the considered distribution was gcc64-only, compiled using gcc version 9.3.0[9]. Regarding the HACL* and CompCert combination, the same remark from 5.1.3 applies in this context.

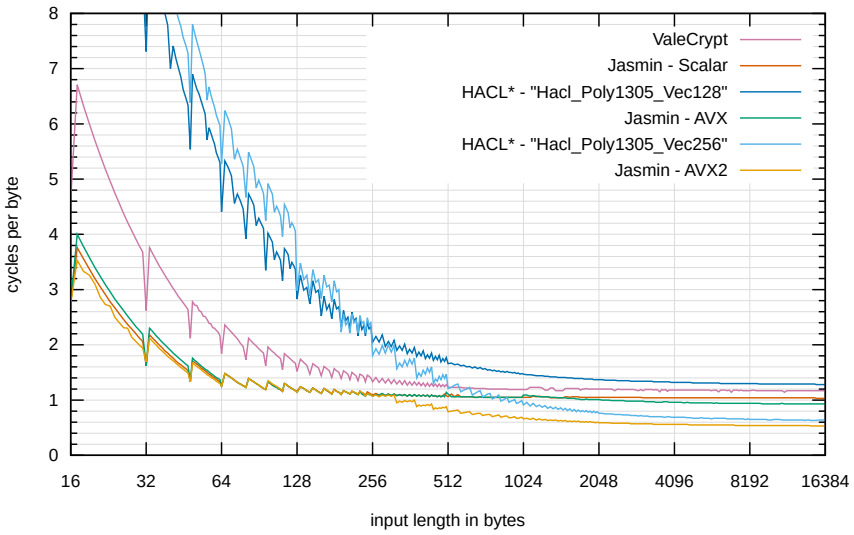

Figure 5.10: Poly1305: comparison of Jasmin and HACL* implementations.

For 16KiB inputs, the performance is: 1.04/1.17 cpb for Jasmin Scalar and ValeCrypt; 0.93/1.28 cpb for AVX implementations; and 0.53/0.64 cpb for AVX2 implementations. For short inputs, all three Jasmin implementations exhibit good cycles per byte performance, mainly because of the usage of a mixed approach, where the AVX and AVX2 implementations avoid the corresponding internal state initialization by checking the input length beforehand. Also worth highlighting that the ValeCrypt implementation is proven correct at the assembly level, similarly to the proposed Jasmin implementations.

[8] https://github.com/project-everest/hacl-star/tree/a50b659d11953dadd8d84ec5df25203cec1a746b
[9] CFLAGS set as -Ofast -march=native -mtune=native -m64 -fwrapv -fomit-frame-pointer -funroll-loops

Figure 5.11 compares the developed Jasmin implementations with OpenSSL[10]. This figure contains the data collected for four Poly1305 OpenSSL implementations, one written in C and the others written in assembly. The same observations that were made for ChaCha20, regarding the compilation of multiple libraries and testing procedure, also apply in this context. The line corresponding to OpenSSL's C implementation is also not included in this figure.

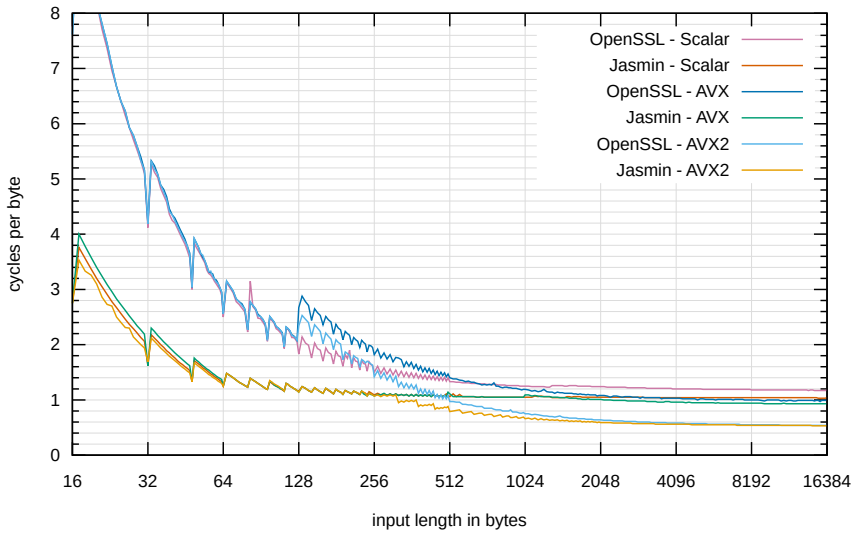

Figure 5.11: Poly1305: comparison of Jasmin and OpenSSL implementations.

Regarding the performance for 16KiB inputs: 1.97/1.18/0.99/0.54 cpb for OpenSSL implementations; and 1.04/0.93/0.53 for Jasmin implementations. For the sake of a fair comparison, and regarding the small difference between 0.54 cpb and 0.53 cpb, the best reported median values for these implementations and for the aforementioned input length were 8800 and 8744 CPU cycles for the OpenSSL and Jasmin AVX2 implementations, respectively. A more approximate cycle per byte count for these two would be 0.5371 and 0.5336 cpb, which is effectively the same. All Poly1305 implementations were benchmarked five times and the five reported medians for these were (in the order that they were collected): 8800, 8808, 8808, 8812, and 8810 CPU cycles for OpenSSL AVX2; and 8746, 8746, 8748, 8744, and 8744 for Jasmin AVX2. It can be considered that the benchmarking setup is stable. This difference of roughly 50 CPU cycles is (most probably) related to the OpenSSL API for calling Poly1305, which requires three functions calls: Init; Update; and Finish. It is also possible to observe in figure 5.11 that OpenSSL is also sharing the same code for all implementations represented in the plot up until 128 bytes. For inputs slightly greater than 128 bytes the performance slightly deteriorates when compared to the scalar implementation. Overall, ValeCrypt and OpenSSL share the same code, and the small difference between these two is related to the used API. As for future work, the Jasmin implementations of Poly1305 will also be updated

[10] `https://github.com/openssl/openssl/tree/bc8c36272067f8443f875164831ce3a5a739df3f`

to also support non one-shot calls: the inputs can be iteratively consumed with calls to *update* functions.

Figure 5.12 compares the AVX2 Jasmin implementation with other open-source implementations. It includes measurements from libsodium[11], which was compiled using the provided Makefile without any modifications to it. The implementation included in libsodium is not an AVX2 implementation and it is written in C. For reference, two other implementations from SUPERCOP are included, amd64 and moon/avx2/64. The cycles per byte for the implementations included in this figure and for 16KiB inputs are: 2.71 cpb for amd64; 1.17 cpb for libsodium; 0.61 cpb for moon/avx2/64; and 0.53 cpb for Jasmin AVX2.

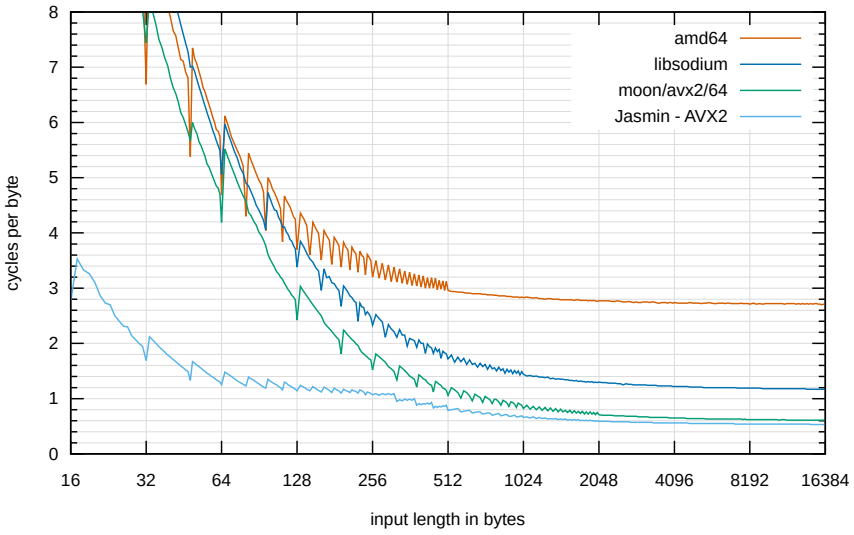

Figure 5.12: Poly1305: comparison between open-source implementations.

5.3 Curve25519

Curve25519 is an elliptic-curve Diffie-Hellman key exchange protocol proposed by Bernstein [Ber06b]. It is based on the custom-designed curve Curve25519 defined as $E : y^2 = x^3 + 486662x^2 + x$ over the field $\mathbb{F}_{2^{255}-19}$. This curve was chosen to provide cryptographic security, but design choices also took into consideration the need for speed. As a result of these choices, Curve25519 has been adopted for widespread use in various contexts, including the TLS and the Signal protocols.

[11] `https://github.com/jedisct1/libsodium/tree/a016aea61214668827e18c6278ac25b0bbc98ca5`

5.3.1 Algorithm Overview

Elliptic curve cryptography [HMV04] relies on hardness assumptions on algebraic groups formed by the points of carefully chosen elliptic curves over finite fields. Let $\mathbb{F}_q$ be the finite field of prime order q. An elliptic curve is defined by the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy an equation of the form $E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$, for a_1, a_2, a_3, a_4, $a_6 \in \mathbb{F}_q$ (with certain restrictions on these parameters). This set of points, together with a "point at infinity", form a group of size $l \approx q$. The group law has a geometric interpretation, which is not relevant for the purpose of this discussion; what *is* important is that the group law can be computed very efficiently — particularly when compared to the computations underlying other algebraic structures used in public-key cryptography — using only a few operations in $\mathbb{F}_q$. Similarly, scalar multiplication,[12] which is the core operation for elliptic curve cryptography, can also be computed very efficiently.

Scalar multiplication in Curve25519 is usually implemented using Montgomery's differential-addition chain — also known as Montgomery ladder — which permits performing the computation directly over the x-coordinate of elliptic curve points. This algorithm is shown in Algorithm 2. It is ideal for high-security and high-speed implementation for two reasons. First, it is much simpler than the generic algorithm for elliptic curves, so its overall efficiency essentially only depends on the cost of the underlying field operations, which can be computed very fast in modern architectures. Second, it is highly regular and can be implemented in constant-time by executing exactly the same code for each scalar bit (called a *ladder step*), making sure that the appropriate inputs are fed to this code via (constant-time) swapping of (X_2, Z_2) with (X_3, Z_3).

Algorithm 2 Montgomery Ladder

Input: A scalar k and the x-coordinate x_P of a point P on E.
Output: (X_{kP}, Z_{kP}) fulfilling $x_{kP} = X_{kP}/Z_{kP}$
$t \leftarrow \lceil \log_2 k + 1 \rceil$
$X_1 \leftarrow xP$; $X_2 \leftarrow 1$; $Z_2 \leftarrow 0$; $X_3 \leftarrow x_P$; $Z_3 \leftarrow 1$
for $i \leftarrow t - 1$ **downto** 0 **do**
  if bit i of k is 1 **then**
    $(X_3, Z_3, X_2, Z_2) \leftarrow \mathsf{ladderstep}(X_1, X_3, Z_3, X_2, Z_2)$
  else
    $(X_2, Z_2, X_3, Z_3) \leftarrow \mathsf{ladderstep}(X_1, X_2, Z_2, X_3, Z_3)$
  end if
end for
return (X_2, Z_2)

[12] Given a curve point P and a scalar $k \in \mathbb{Z}$, scalar multiplication computes the point $Q = k \cdot P = \underbrace{P + \ldots + P}_{k \text{ times}}$.

The computations of each step in the ladder, all over $\mathbb{F}_q$, are shown in Algorithm 3. Typical implementations of the scalar multiplication operation implement the Montgomery ladder step in fully inlined hand-optimized assembly, and also include field multiplication and inversion as hand-optimized assembly routines (these are needed to recover the final x-coordinate of the result once the ladder is computed). The main difference between various implementations lies in the representation of $\mathbb{F}_{2^{255}-19}$ field elements and their handling in the hand-crafted assembly code, as the optimal choice varies from one architecture to another due to word size and available machine operations, and their relative efficiency. The higher-level functions that call the assembly routines for the various ladder steps and finalize the results are usually implemented in C. This is inconvenient when formal verification is the goal, since the relevant routines are now split between two programming languages with very different characteristics.

Algorithm 3 One step of the Curve25519 Montgomery Ladder

function ladderstep$(X_1, X_2, Z_2, X_3, Z_3)$

$A \leftarrow X_2 + Z_2$
$AA \leftarrow A^2$
$B \leftarrow X_2 - Z_2$
$BB \leftarrow B^2$
$E \leftarrow AA - BB$
$C \leftarrow X_3 + Z_3$
$D \leftarrow X_3 - Z_3$
$DA \leftarrow D \cdot A$
$CB \leftarrow C \cdot B$
$X_3 \leftarrow (DA + CB)^2$
$Z_3 \leftarrow X_1 \cdot (DA - CB)^2$
$X_2 \leftarrow AA \cdot BB$
$Z_2 \leftarrow E \cdot (AA + 121665 \cdot E)$ *or* $Z_2 \leftarrow E \cdot (BB + 121666 \cdot E)$
return (X_2, Z_2, X_3, Z_3)

end function

5.3.2 Curve25519 Implementations

Three implementations of the Curve25519 scalar multiplication were developed in Jasmin: ref4, ref5, and mulx. ref4 and mulx implementations use four 64-bit limbs to represent field elements while ref5 uses five 51-bit limbs. ref4 and ref5 use the mulq assembly instruction to perform the multiplications, which modifies the arithmetic flags, while mulx uses the mulx assembly instruction, which does not affect the arithmetic flags. mulx is defined in BMI2 (Bit Manipulation Instruction set), and it is useful in contexts where is it advantageous to perform

multiple carry-chain additions: since the arithmetic flags are not changed by the execution of this instruction, it can be used together with the addition instructions from the Intel ADX extension, adcx and adox, which only affect the carry and overflow flag, respectively. All implementations share the same interface:

```
export fn curve25519(reg u64 rp kp up)
{ // ...
}
```

All arguments in this function are memory pointers. When compared with the SUPERCOP API, rp, kp, and up, from the Jasmin interface, correspond to u8 *q, cu8 *n, and cu8 *p, respectively. The chosen argument names are aligned with RFC7748 [LHT16].

In this section, the scalar multiplication with the base point 9 is not discussed. When the point is known, the computation can be optimized [OLH+17]. There is a preliminary Jasmin implementation available[13] that implements the optimizations described in [OLH+17], but it is not formally verified yet. This constitutes future work. Also future work is the formal verification of a vectorized Jasmin implementation (AVX2) that implements the optimizations described in [FHLD19a]. In an ideal scenario, the proofs of these two additional implementations can leverage the verification effort taken to verify the implementations presented in this section. Regarding this section's structure, first, we discuss the field element's basic arithmetic operations for each considered number of limbs, four and five. The discussion follows by presenting the optimization steps that were taken to optimize one step of the Montgomery Ladder algorithm. To conclude, an overview of the complete Jasmin implementation of the Curve25519 is presented.

Field Arithmetic

There are four basic arithmetic operations that must be implemented for this primitive: addition; subtraction; multiplication; and squaring. Multiplication and squaring are essentially the same, given that squaring an element is the same as multiplying the element by itself, but the actual implementation can take advantage of that to improve performance. Given that the size of these field elements can fit in 255 bits (but 256 bits may be required for performance reasons), these can be represented by arrays of four or five u64 variables. The value contained in a four-limb array a can be reconstructed with the following expression: $2^0 \cdot a[0] + 2^{64} \cdot a[1] + 2^{128} \cdot a[2] + 2^{192} \cdot a[3]$. If the reconstructed value is greater or equal than $2^{255} - 19$, an additional $\pmod{2^{255} - 19}$ is required. This step must be performed in constant-time. The previous expression can be easily extended for a 5-limb representation.

[13] https://github.com/tfaoliveira/libjc/

Addition and Subtraction

The addition and subtraction algorithms for the ref4 and mulx implementations are the same given that, in both implementations, the same 4-limbs representation is used. Figure 5.13 presents two inline functions, to perform an addition (left) and a subtraction (right). The presented functions expect as first argument a register array f and a stack array gs, and return, in a register array, the result of computing f + gs and f - gs, respectively.

```
inline fn add4_rrs(reg u64[4] f, stack u64[4] g)
  → reg u64[4]
{
  inline int i; reg bool cf;
  reg u64[4] h;
  reg u64 z;

  _,_,_,_,_,z = #set0();
  h = f;

  cf, h[0] += gs[0];
  for i=1 to 4
  { cf, h[i] += gs[i] + cf; }

  _, z -= z - cf;
  z &= 38;

  cf, h[0] += z;
  for i=1 to 4
  { cf, h[i] += 0 + cf; }

  _, z -= z - cf;
  z &= 38;
  h[0] += z;

  return h;
}
```

```
inline fn sub4_rrs(reg u64[4] f, stack u64[4] gs)
  → reg u64[4]
{
  inline int i; reg bool cf;
  reg u64[4] h;
  reg u64 z;

  _,_,_,_,_,z = #set0();
  h = f;

  cf, h[0] -= gs[0];
  for i=1 to 4
  { cf, h[i] -= gs[i] - cf; }

  _, z -= z - cf;
  z &= 38;

  cf, h[0] -= z;
  for i=1 to 4
  { cf, h[i] -= 0 - cf; }

  _, z -= z - cf;
  z &= 38;
  h[0] -= z;

  return h;
}
```

Figure 5.13: Curve25519: add and sub functions for 4-limbs in Jasmin.

Both function names are suffixed with the return and arguments types: rrs. Depending on the requirements of the top-level code (the code that calls these leaf functions), it may be practical to have multiple functions definitions that perform the same arithmetic operation but, in some cases, are optimized for their argument type. For instance, a function add_rsr, which would receive as first argument a stack array and as second argument a register array,

may be easily implemented as a call to the presented function add_rsr where the arguments are swapped given that addition is commutative.

As another example, the definition of functions such as add_sss, that operate only with stack arrays, can be implemented as a load for the first argument to a register array, followed by a call to the presented function add_rrs, followed by a store of the result for subsequent return. This programming pattern improves the readability of the complete implementation and eases the optimization process of code such as the one presented in Algorithm 3.

Regarding the computation that is being performed by add4_rrs and sub4_rrs, after setting a variable z with the value 0 and copying all elements from the first argument into another register array[14], a carry chain addition, or subtraction with borrow, is performed. If the carry flag is set after the first loop, then the value needs to be reduced. To avoid any side-channel leakage, the reduction is always performed. Variable z is then set to 0 or 38 depending if the carry flag is set or not. This is equivalent to multiply the last carry, or borrow, bit by 2×19. After adding, or subtracting, z to the array to perform the reduction, the same procedure is used again since the carry flag can be set after the addition, or subtraction. This second time, however, the first limb can accommodate an addition, or subtraction, by 38 without the carry flag being set[15]. The resulting value is then less than 2^{256}, but it can be greater than $2^{255} - 19$. The implementations described in this paragraph are standard, in the sense that other cryptographic libraries implement the same algorithm using the same sequence of instructions, for instance, in qhasm or directly in assembly.

The code that implements the addition and subtraction in the ref5 implementation, which uses five 51-bit limbs, is not shown but it is described next. The main motivation for using a 51-bit limb representation is to reduce the usage of the adc instruction in micro-architectures where this instruction has a low throughput and big latency. For instance, in Intel Nehalem CPUs, adc and sbb instructions, which allow to perform an addition with carry and a subtraction with borrow, respectively, have a latency and a throughput of 2 cycles, when both operands are registers. On the other hand, add and sub have a latency and throughput of 1 and 0.33, respectively. For comparison, the Skylake micro-architecture has a latency and a throughput of 1 for adc and sbb, and a latency of 1 and a throughput of 0.25 for add and sub. Given this non-negligible difference in latency and throughput in Nehalem, implementations that use a limb size that is less than the total amount of available bits, in this case 64 bits, perform faster than the alternative approach that uses four 64-bit limbs [BDL+12b]. The main reason for this is that, by using 51-bit limbs, one can perform a field element addition, or subtraction, without fully propagating the carry (use add instead of adc) and without performing a reduction (which can be done when multiplications are

[14]The addition and subtraction is not performed inplace for these functions, although the compiler could actually remove this copy if the return value is attributed to the first argument, given that the variables would, most probably, be merged.

[15]As an intuition, this property needs to be explicitly proven in EasyCrypt.

done, for instance). An additional step (when compared to 64-bit limbs implementations) that is performed when doing the subtraction, is to first add $2 \times p$, with p being $2^{255} - 19$, to the element being subtracted, before the subtraction happens. The result is congruent modulo $2^{255} - 19$. This implies extra reasoning when doing the proving the correctness of the algorithms.

Multiplication and Squaring

The multiplication between two field elements f and g, each represented with four 64-bit limbs, corresponds to the following expression:

$$\begin{aligned}(2^0 \cdot f[0] &+ 2^{64} \cdot f[1] + 2^{128} \cdot f[2] + 2^{192} \cdot f[3]) \times \\ (2^0 \cdot g[0] &+ 2^{64} \cdot g[1] + 2^{128} \cdot g[2] + 2^{192} \cdot g[3])\end{aligned}$$

The previous multiplication can be expanded as follows:

$$\begin{aligned}
&2^0 \cdot (f[0] \cdot g[0]) + \\
&2^{64} \cdot (f[0] \cdot g[1] + f[1] \cdot g[0]) + \\
&2^{128} \cdot (f[0] \cdot g[2] + f[1] \cdot g[1] + f[2] \cdot g[0]) + \\
&2^{192} \cdot (f[0] \cdot g[3] + f[1] \cdot g[2] + f[2] \cdot g[1] + f[3] \cdot g[0]) + \\
&2^{256} \cdot (f[1] \cdot g[3] + f[2] \cdot g[2] + f[3] \cdot g[1]) + \\
&2^{320} \cdot (f[2] \cdot g[3] + f[3] \cdot g[2]) + \\
&2^{384} \cdot (f[3] \cdot g[3])
\end{aligned}$$

A register array h, with eight u64 elements, can hold the result of the previous expression, which can be later reduced modulo the prime being used. It is also possible to observe that if f and g are the same, which means that a square is being done, then several multiplications can be replaced by a couple of additions, which need to be performed in any case. For instance, $f[0] \cdot g[1]$ is effectively the same value as $f[1] \cdot g[0]$ and, as such, the second multiplication does not need to happen, and this value can be added to itself.

The mulx assembly instruction allows to multiply rdx by another 64-bit register (or memory operand) and the result, a 128-bit number, is written in two registers: one contains the lower 64 bits; and the other contains the higher 64 bits. The output of mulx can be written in any two registers. In comparison, mulq always writes the results in rdx and rax. In this context, where one of the inputs must be loaded to a specific register, one of the field elements can be a stack array and the other a register array. For instance, if f is a stack array, then it would favor performance if every sequence of multiplications that uses a given limb is performed sequentially: when $f[0]$ is loaded into a register, then $f[0] \cdot g[0]$, $f[0] \cdot g[1]$, $f[0] \cdot g[2]$, and $f[0] \cdot g[3]$ can occur. This pattern (operand scanning) can be seen by shifting the previously presented multiplication:

$$
\begin{array}{lcllllllll}
2^{0} & \cdot & (f[0]\cdot g[0] & & & & & &) & + \\
2^{64} & \cdot & (f[0]\cdot g[1] & + & f[1]\cdot g[0] & & & &) & + \\
2^{128} & \cdot & (f[0]\cdot g[2] & + & f[1]\cdot g[1] & + & f[2]\cdot g[0] & &) & + \\
2^{192} & \cdot & (f[0]\cdot g[3] & + & f[1]\cdot g[2] & + & f[2]\cdot g[1] & + \; f[3]\cdot g[0]) & + \\
2^{256} & \cdot & (& & f[1]\cdot g[3] & + & f[2]\cdot g[2] & + \; f[3]\cdot g[1]) & + \\
2^{320} & \cdot & (& & & & f[2]\cdot g[3] & + \; f[3]\cdot g[2]) & + \\
2^{384} & \cdot & (& & & & & f[3]\cdot g[3]) &
\end{array}
$$

The result of the first multiplication of the first column, $f[0] \cdot g[0]$, is written in $h[0]$ and $h[1]$. The lower part of the second multiplication, $f[0] \cdot g[1]$, needs to be added to $h[1]$, and the higher part is written to $h[2]$. The carry resulting from adding the lower part of $f[0] \cdot g[1]$ to $h[1]$ is added to $h[2]$ after the third multiplication, $f[0] \cdot g[2]$. The same applies to the remaining limbs. At the end of this column, the carry is added to $h[4]$. Only the carry flag and the corresponding adcx instruction are used until this point.

For the remaining columns, the overflow and carry flags (and corresponding adox and adcx instructions) are used to perform the additions between each multiplication result (low and high) and the corresponding limb. For instance, after $f[1]$ is loaded into a register and multiplied by $g[0]$, the lower part must be added to $h[1]$. From this addition results a carry, which is placed in the overflow flag (adox). The upper part of this multiplication must be added to $h[2]$ which will also result in a carry, but this time it is placed in the carry flag (adcx). The flags are propagated while adding the results of subsequent additions. Similarly to the finalization procedure from the first column, the carry and overflow flags are added to $h[5]$. While it is intuitive that a given carry can be added to the upper part of a given multiplication without unexpectedly overflowing, it is not as intuitive when two 1-bit values are added to the same element. This is addressed by the correctness proof.

After the multiplication is performed, the result needs to be reduced. To perform the reduction, $h[4]$, $h[5]$, $h[6]$, and $h[7]$ are multiplied by 38, and added to the corresponding limb. The reduction is similar to the one described in the context of addition. The reduction implementation also takes advantage of mulx features. The discussed algorithm is equivalent to the corresponding OpenSSL implementation (the one that does use the mulx instruction), minus one copy from register to stack that OpenSSL uses to free one register.

The discussion was focused on the multiplication routine for the case where four 64-bit limbs are used in a context where mulx, adox and adcx are available. The intuition for the remaining multiplication implementations is the same but, for implementations that use mulq to perform the multiplication of field elements, both arguments need to be in stack.

Optimizing one step of Montgomery Ladder

All three implementations share, in its essence, the same top-level code, with only the leaf functions (for instance, the ones that are used to multiply and perform additions and subtractions between field elements) being different. It is stated as 'in its essence' because, since there are 4 and 5-limb implementations, which require arrays of different sizes, the top-level functions' declarations and corresponding local variables must be differently declared: in ref4 and mulx implementations these elements can be declared as 'reg u64[4]'; and in ref5 as 'reg u64[5]'. However, it would be interesting to explore how the support in Jasmin for the definition of new data types, which could be used to encapsulate the instantiated type until fixing it during compilation time, could benefit code modularity. In practice, it is possible to replicate such a feature by using a preprocessor and corresponding #define directives, for instance. The problem with following this approach is that some of the proofs would need to be replicated as there would be different extractions for the same program.

During a scalar multiplication, roughly 90% of the CPU time is spent on Algorithm 2. The remaining 10% corresponds to a modular inversion computed after the Montgomery ladder. These are round percentages based on the Jasmin implementations of this primitive. Additionally, the optimized modular inversion proposed in [BY19] is not yet implemented in Jasmin. It is estimated that using this faster modular inversion algorithm (and based on the data provided by the authors of the work) the overall performance, and for mulx implementation, should be improved by 2% or 3%. This is future work.

The key for achieving a highly efficient implementation of Curve25519 scalar multiplication relies, first, on a careful implementation of the ladderstep function (shown in Algorithm 3), given that it is called 255 times and also that it uses most of the CPU time in this primitive, and, second, the functions that perform the addition, subtraction, multiplication, and square, with the latter being a specialization of the multiplication algorithm. We start by discussing alternative ways of *unrolling* ladderstep into a sequence of single calls to the available leaf functions $(+, -, \cdot, ^2)$ and study the performance impact of the resulting assembly code.

Figure 5.14 presents a low-level implementation of Algorithm 3. The presented ladderstep0 implementation can be found in a SUPERCOP's implementation, amd64-64[16], a qhasm implementation where the ladderstep function is fully inlined. In this case, fully inlined means that there are no function calls to the leaf operations, for instance, multiplication or addition. As such, this sequence of operations is not directly observable from the source file referred on the previous footnote. As mentioned during Algorithm Overview, § 5.3.1, some implementations use two different programming languages: in this case, amd64-64 has the higher-level functions written in C and the ladderstep, for instance, written in qhasm, which can be independently compiled to assembly using the qhasm compiler. amd64-64, which

[16]The complete path is: crypto_scalarmult/curve25519/amd64-64/ladderstep.S; Related paper [BDL+12b].

ladderstep0$(X_{1s}, X_{2s}, Z_{2s}, X_{3s}, Z_{3s})$

$T_1 \leftarrow X_{2s}$
$T_2 \leftarrow T_1$
$T_1 \leftarrow T_1 + Z_{2s}$
$T_2 \leftarrow T_2 - Z_{2s}$
$T_{1s} \leftarrow T_1$
$T_{2s} \leftarrow T_2$
$T_7 \leftarrow T_{2s}^2$
$T_{7s} \leftarrow T_7$
$T_6 \leftarrow T_{1s}^2$
$T_{6s} \leftarrow T_6$
$T_5 \leftarrow T_6$
$T_5 \leftarrow T_5 - T_{7s}$
$T_{5s} \leftarrow T_5$
$T_3 \leftarrow X_{3s}$
$T_4 \leftarrow T_3$
$T_3 \leftarrow T_3 + Z_{3s}$
$T_4 \leftarrow T_4 - Z_{3s}$
$T_{3s} \leftarrow T_3$
$T_{4s} \leftarrow T_4$
$T_9 \leftarrow T_{2s} \cdot T_{3s}$
$T_{9s} \leftarrow T_9$
$T_8 \leftarrow T_{1s} \cdot T_{4s}$
$Z_Q \leftarrow T_8$
$Z_Q \leftarrow Z_Q - T_{9s}$
$T_8 \leftarrow T_8 + T_{9s}$
$X_{3s} \leftarrow T_8$
$Z_{3s} \leftarrow Z_Q$
$X_Q \leftarrow X_{3s}^2$
$X_{3s} \leftarrow X_Q$
$Z_Q \leftarrow Z_{3s}^2$
$Z_{3s} \leftarrow Z_Q$
$Z_Q \leftarrow X_{1s} \cdot Z_{3s}$
$Z_{3s} \leftarrow Z_Q$
$X_P \leftarrow T_{7s} \cdot T_{6s}$
$X_{2s} \leftarrow X_P$
$Z_P \leftarrow T_{5s} \cdot 121666$
$Z_P \leftarrow Z_P + T_{7s}$
$Z_{2s} \leftarrow Z_P$
$Z_P \leftarrow T_{5s} \cdot Z_{2s}$
$Z_{2s} \leftarrow Z_P$
return $(X_{2s}, Z_{2s}, X_{3s}, Z_{3s})$

Figure 5.14: Curve25519: ladderstep version 0.

implements the ladderstep0 as shown, takes roughly 147.4K CPU cycles to execute on an Intel Skylake (i7-6500U).

The algorithm presented in figure 5.14 considers register allocation restrictions: each variable is an array that represents a field element and variables suffixed with an $_s$, for instance, X_{1s} or T_{1s}, correspond to stack arrays while the remaining variables are register arrays. As additional notes, all additions and subtractions are performed inplace (where the output is written to the first argument, which is a register array) and multiplication and squaring operations expect all arguments in stack arrays and return the computed result in registers.

The optimized Curve25519 implementation presented in the first Jasmin paper [ABB+17a], implements ladderstep0 as shown in figure 5.14 but with some changes to improve performance: instead of receiving and returning X_{2s} as a stack array, X_2 is placed in registers upon entry and exiting this function. To achieve this, the first line of the previous algorithm ($T_1 \leftarrow X_{2s}$) is removed and every subsequent usage of T_1 is replaced by X_2. Next, to guarantee (with minimal cost) that the value corresponding to X_2 is loaded into registers before the return happens, the multiplication of T_{7s} with T_{6s} (which is copied into X_{2s}) can

be moved to the end of ladderstep0. Also inside the loop of Montgomery Ladder, shown in Algorithm 2, is the constant-time swap operation between (X_2, Z_2) with (X_3, Z_3). This function can also be implemented to take advantage of the fact that X_2 is a register array.

The Jasmin implementation of Curve25519 included in [ABB+17a], which includes the improved ladderstep, takes 143.4K CPU cycles to execute on the same Intel Skylake (i7-6500U) and under the same conditions of the previous benchmark for amd64-64. It is then 4K CPU cycles faster than amd64-64. To better understand the performance impact of the optimization described in the last paragraph, a different Jasmin implementation containing ladderstep0 without the described optimizations (and also a constant-time swap that operates on stack arrays) was developed. This intermediate implementation takes 146K CPU cycles to execute. Hence, writing the implementation using Jasmin and, consequently, avoiding function calls from C to assembly, causes a performance improvement of 1.4K CPU cycles. The remaining 2.6K CPU cycles can be attributed to the described optimization. Since the loop body of Montgomery Ladder is executed 255 times for Curve25519, the loop body takes roughly less 10 cycles to execute.

It is important to highlight at this point that a similar performance improvement can be achieved using qhasm: the described optimizations can be integrated in amd64-64 by implementing the while loop in qhasm, instead of C, which would avoid the corresponding calls to ladderstep and the constant-time swapping functions, and, additionally, would also allow for X_2 to be in registers. In fact, this statement is true for all implementations: any efficient Jasmin function can be encoded in qhasm, and also the opposite. Given the features of the Jasmin programming language and its compiler, it may be, however, more convenient to implement and test different optimization strategies using Jasmin.

Considering that the computation of each step of the Montgomery Ladder is the most speed-critical part in the complete Curve25519 computation, further research was done on how to take advantage of the characteristics of the Jasmin programming language while using different arrangements for the ladderstep function. Figure 5.15 shows three different implementations to compute one step of the Montgomery Ladder. This set of implementations are higher-level when compared to the one presented in figure 5.14: for these, one can assume that the basic operations arithmetic operations for the field elements are defined for both register and stack inputs and outputs, inplace or non-inplace in the case of additions and subtractions. In contrast to the algorithm shown in figure 5.14, all elements are stack arrays, except those suffixed with an $_r$ which are placed in registers.

The leftmost implementation, ladderstep1, can be found in some Curve25519 implementations, for instance, in OpenSSL[17]. Depending on how Algorithm 3 is unrolled, the multiplication of a field element by the constant 121666, or 121665, occurs. All the presented alternatives are equivalent in the sense that they perform the same computation. RFC7748 [LHT16],

[17] https://github.com/openssl/openssl/

ladderstep1$(X_1, X_2, Z_2, X_3, Z_3)$	ladderstep2$(X_1, X_2, Z_2, X_3, Z_3)$	ladderstep3$(X_1, X_2, Z_{2r}, X_3, Z_3)$
$T_0 \leftarrow X_3 - Z_3$	$T_0 \leftarrow X_3 - Z_3$	$T_0 \leftarrow X_2 - Z_{2r}$
$T_1 \leftarrow X_2 - Z_2$	$T_1 \leftarrow X_2 - Z_2$	$X_2 \leftarrow X_2 + Z_{2r}$
$X_2 \leftarrow X_2 + Z_2$	$X_2 \leftarrow X_2 + Z_2$	$T_1 \leftarrow X_3 - Z_3$
$Z_2 \leftarrow X_3 + Z_3$	$Z_2 \leftarrow X_3 + Z_3$	$Z_2 \leftarrow X_3 + Z_3$
$Z_3 \leftarrow X_2 \cdot T_0$	$Z_3 \leftarrow X_2 \cdot T_0$	$Z_3 \leftarrow X_2 \cdot T_1$
$Z_2 \leftarrow Z_2 \cdot T_1$	$Z_2 \leftarrow Z_2 \cdot T_1$	$Z_2 \leftarrow Z_2 \cdot T_0$
$T_0 \leftarrow T_1^2$	$T_0 \leftarrow T_1^2$	$T_2 \leftarrow X_2^2$
$T_1 \leftarrow X_2^2$	$T_1 \leftarrow X_2^2$	$T_1 \leftarrow T_0^2$
$X_3 \leftarrow Z_3 + Z_2$	$X_3 \leftarrow Z_3 + Z_2$	$X_3 \leftarrow Z_3 + Z_2$
$Z_2 \leftarrow Z_3 - Z_2$	$Z_2 \leftarrow Z_3 - Z_2$	$Z_2 \leftarrow Z_3 - Z_2$
$X_2 \leftarrow T_1 \cdot T_0$	$X_2 \leftarrow T_1 \cdot T_0$	$X_2 \leftarrow T_2 \cdot T_1$
$T_1 \leftarrow T_1 - T_0$	$T_2 \leftarrow T_1 - T_0$	$T_0 \leftarrow T_2 - T_1$
$Z_2 \leftarrow Z_2^2$	$Z_2 \leftarrow Z_2^2$	$Z_2 \leftarrow Z_2^2$
$Z_3 \leftarrow T_1 \cdot 121666$	$Z_3 \leftarrow T_2 \cdot 121665$	$Z_3 \leftarrow T_0 \cdot 121665$
$X_3 \leftarrow X_3^2$	$X_3 \leftarrow X_3^2$	$X_3 \leftarrow X_3^2$
$T_0 \leftarrow T_0 + Z_3$	$T_0 \leftarrow T_1 + Z_3$	$T_2 \leftarrow T_2 + Z_3$
$Z_3 \leftarrow X_1 \cdot Z_2$	$Z_3 \leftarrow X_1 \cdot Z_2$	$Z_3 \leftarrow X_1 \cdot Z_2$
$Z_2 \leftarrow T_1 \cdot T_0$	$Z_2 \leftarrow T_2 \cdot T_0$	$Z_{2r} \leftarrow T_2 \cdot T_0$

Figure 5.15: Curve25519: ladderstep version 1.

specifies 121665 as the constant to be used. The implementation shown in the middle of figure 5.15, ladderstep2, is included as an intermediate implementation between ladderstep1 and ladderstep3, to make clear what are the necessary changes for the implementation to use the multiplication with 121665: an additional temporary array T_2 is necessary to hold the result of subtracting T_0 from T_1, instead of overwriting T_1, to preserve this value for a later addition with Z_3. Hence, ladderstep2 requires more stack space than ladderstep1. Depending on deployment restrictions, this may be a concern. In the considered scenario, where the code is targeted to the AMD64 architecture and very likely to be run in very capable environment, this additional need for memory can be ignored.

Similarly to the ladderstep0, ladderstep3 implements some rearrangements in the way that computations are performed when compared to ladderstep2: to minimize the amount of memory loads and stores, Z_2 is chosen to be placed in registers instead of stack. The implementation of ladderstep3 moves the operations that are dependent of Z_{2r} to the top of the function to take advantage of this fact. The last multiplication, $Z_{2r} \leftarrow T_2 \cdot T_0$, also leaves the result in registers. Similarly to the optimization previously described, one field element is always placed in registers during the Montgomery Ladder loop and the implementation of the constant-time swap algorithm also takes advantage of this. The previous Jasmin implementation that previously took 143.4K CPU cycles to execute, when equipped with the ladderstep3 implementation, now takes 129.4K CPU cycles on an Intel Skylake (i7-6500U). This is a performance improvement of 14K CPU cycles.

The discussion so far was focused on how different implementation strategies for the ladderstep function can be used to improve the performance of Curve25519. The discussion took as a base example the amd64-64, an implementation available in SUPERCOP, and iterated from there. The presented performance data corresponds to the ref4 Jasmin implementation, which uses four 64-bit limbs and the mulq instruction. As previously mentioned, the multiplication function of this implementation reads both arguments from stack.

The intuition for this is that, given that two arrays with four limbs each are being multiplied, eight registers are required to hold the intermediate multiplication result. In addition to this, two specific registers are used by the mulq instruction (rdx and rax), one more to load the second input of the multiplication instruction if one wants to avoid the usage of a memory operand (the first input is already considered: rax), and another register as a temporary variable. For implementations that use a 4-limbs representation, this corresponds to 12 registers. As such, there are only 3 left registers. It is possible however, to implement a version of the multiplication algorithm that receives the first argument as a register array, as long as this first argument is not live after the multiplication function returns.

With such multiplication function, it is possible to change the squaring of T_0 to return a register array instead: $T_{1r} \leftarrow T_0^2$. Given that the following addition and subtraction do not require many registers, T_{1r} can be live during the execution of these. The next step is to swap the following two operations, $X_2 \leftarrow T_2 \cdot T_1$ and $T_0 \leftarrow T_2 - T_1$, in order for the subtraction to happen before the multiplication. The multiplication can then be updated to $X_2 \leftarrow T_{1r} \cdot T_2$ considering that it is commutative. With this optimization, the complete implementation takes 127.7K CPU cycles to execute, making it 1.8K CPU cycles faster than the Jasmin version that implements ladderstep3 as presented in figure 5.15.

It is important to state that the performance gains obtained by using the presented optimizations are not exclusively related with the avoidance of memory loads and stores. For instance, if we change the ladderstep3 to perform $X_3 \leftarrow X_3^2$ right after X_3 is set by $X_3 \leftarrow Z_3 + Z_2$, the CPU cycle count increases from 129.4K to 132.6K CPU cycles. As such, data dependencies must also be considered. It would be interesting to further explore how to automatically improve the performance (by extending the Jasmin compiler) of such algorithms. The mulx implementation also benefits from the described optimizations, with the fastest version taking 102.8K CPU cycles to execute on an Intel Skylake (i7-6500U).

Implementation Overview

Figure 5.16 shows the top-level code of the Curve25519 implementation in Jasmin: the left-side of the figure shows the curve25519_scalarmult function, which is used by the two existent four limbs implementations, ref4 and mulx; the right-side of the figure shows an implementation of the montgomery_ladder. The implementations for the five limb version of

the code is similar: only the length of the arrays that represent field elements change.

As an overview of the presented algorithms, curve25519_scalarmult starts by storing the output pointer in a stack variable, to free the associated register for later usage, given that this pointer is only needed at the end of the function's execution. The scalar in k is loaded into a byte array using function decode_scalar_25519 (not shown) according to the algorithm's specification, which includes setting some bits to zero and one bit to one. A load is also performed for the u-coordinate using the function decode_u_coordinate. curve25519_scalarmult then calls montgomery_ladder to perform the scalar multiplication. The result is given to function encode_point, which computes $X_2 * Z_{2r}^{-1} \pmod{2^{255} - 19}$. The naming of variables and functions presented here follow the corresponding RFC as close as possible.

The montgomery_ladder function, shown on the right-side of figure 5.16, performs the computation described by Algorithm 2. Given the register array u, it initializes the corresponding points and loops over the bits of the scalar k. The main difference between the shown implementation of montgomery_ladder and Algorithm 2 is that the access of the corresponding scalar bit and constant-time swapping are placed inside function montgomery_ladderstep. Stack variable s indicates if the state, (x2, z2r, x3, z3), is swapped or not.

```
export fn curve25519_scalarmult(
 reg u64 rp kp up)
{
  inline int i;
  stack u8[32] k;
  stack u64[4] x2;
  reg u64[4] u z2r r;
  stack u64 rps;

  rps = rp;
  k = decode_scalar_25519(kp);
  u = decode_u_coordinate(up);
  (x2,z2r) = montgomery_ladder(u, k);
  r = encode_point(x2,z2r);

  rp = rps;
  for i=0 to 4
  { [rp + 8*i] = r[i]; }
}
```

```
inline fn montgomery_ladder(
 reg u64[4] u, stack u8[32] k) →
 stack u64[4], reg u64[4]
{
  stack u64[4] us x2 x3 z3;
  reg u64[4] z2r;
  stack u64 ctrs s;
  reg u64 ctr;

  (x2,z2r,x3,z3) = init_points(u); us = u;
  ctr = 255; s = 0;
  while {
    ctr -= 1; ctrs = ctr;
    (x2, z2r, x3, z3, s) =
      montgomery_ladderstep(
        k, us, x2, z2r, x3, z3, s, ctr);
    ctr = ctrs;
  } (ctr > 0)
  return x2, z2r;
}
```

Figure 5.16: Curve25519: scalarmult implementation in Jasmin.

5.3.3 Performance Evaluation

In this section we compare the performance of different Curve25519 implementations. Differently from what was done for other primitives, in this section measurements are presented using tables. Although the multiplication with the base point was not discussed (crypto_scalarmult_base), preliminary results are included for completeness: the corresponding Jasmin implementations still need to be optimized and formally verified.

		scalarmult	scalarmult_base
Jasmin	ref4	128932	87520
	ref5	142760	-
	mulx	102784	67886
Everest	Vale	119904	-
	HACL*	141874	-
BoringSSL	Fiat-Crypto	163972	60066
fld-ecc-vec	amd64	117210	35284
	avx2	102212	
OpenSSL	fe64	123102	116360
	fe51	148538	
	no-asm	160426	
SUPERCOP	amd64-51	147276	-
	amd64-64	147370	-
	donna	267984	-
	donna-c64	151384	-
	sandy2x	141092	-

Table 5.1: Curve25519: performance comparison on an Intel Skylake (i7-6500U).

Table 5.1 presents the benchmark results for several Curve25519 implementations on an Intel Skylake CPU. On the top of the table, the measurements for the Jasmin implementations can be found. mulx is the fastest Jasmin implementation, taking 102.8K CPU cycles to executed, followed by ref4 and ref5, which take 129K and 142.8K CPU cycles, respectively. The small discrepancy between the reported value of 129K CPU cycles in table 5.1 and the aforementioned 129.4K CPU cycles (during the discussion on how to optimize one step of the Montgomery Ladder), is due to different implementations being used: the previous discussion took as a base example the Jasmin implementation that was developed for [ABB+17a] and iterated from there, while the value reported in this table relates to a slightly improved implementation, which is closer to RFC7748, that is partially shown in figure 5.16.

In the context of project Everest [BBDL+17], two implementations from Evercrypt [PPF+20][18]

[18] https://github.com/project-everest/hacl-star/tree/a50b659d11953dadd8d84ec5df25203cec1a746b

cryptographic library are included in table 5.1: the first implementation, developed in Vale, takes 119.9K CPU cycles to compute, is platform dependent, and it is comparable (given that it uses mulx, adox, and adcx instructions) to the Jasmin mulx implementation; the second implementation, from HACL*, which is portable, takes 141.9K CPU cycles to execute. The cycle count for scalarmult_base is roughly the same for these implementations which indicates that the scalarmult_base is implemented as a call to scalarmult and, as such, the corresponding values are omitted (-). For these measurements, the library was compiled using the same compiler and flags[19] that were previously used for ChaCha20 and Poly1305 performance evaluations, during § 5.1.3 and § 5.2.3, respectively.

BoringSSL[20], which includes code from Fiat-Crypto [EPG+19a] for the Curve25519 computation, takes 164K CPU cycles to execute for the scalarmult computation and 60K CPU cycles for the scalarmult_base, which is the second best value from in table 5.1.

The implementations proposed by [FHLD19b], fld-ecc-vec[21] in the table, are fastest non-Jasmin implementations available in this context, taking 117.2K and 102.2K CPU cycles for the amd64 implementation (uses mulx instruction) and avx2 versions, respectively. The scalarmult_base, which takes 35.3K CPU cycles, is the same for both versions. The code was compiled with Clang, version 10.0.0. A preliminary Jasmin vectorized implementation, comparable to the fld-ecc-vec/avx2, already exists, but it takes roughly 115K CPU cycles to execute and, as such, it still needs to be optimized.

OpenSSL[22] provides three implementations that can be executed on a Skylake, here designed by: fe64, an assembly implementation which uses four 64-bit limbs and the mulx instruction; fe51, another assembly implementation that uses five 51-bit limbs; and no-asm, which corresponds to the C implementation that can be used when OpenSSL is compiled with the -no-asm flag. These implementations take 123.1K, 148.5K, and 160.4K CPU cycles, respectively, to execute. The implementation (and performance) of scalarmult_base is the same for all considered setups (it is written in C), and it takes 116.4K CPU cycles to execute.

In addition to the previously discussed implementations, some implementations from the SUPERCOP toolkit are included in the same table, to establish a common point for comparison, given that its performance can be easily, and independently, checked by anyone who is familiar with this toolkit. The fastest implementation in this set is sandy2x, which takes 141.1K CPU cycles.

Overall, and in the evaluated micro-architecture, the proposed Jasmin mulx is competitive with all alternatives, with the avx2 version, from fld-ecc-vec, being the only one that performs slightly faster than this one.

[19] gcc version 9.3.0; CFLAGS set as -Ofast -march=native -mtune=native -m64 -fwrapv -fomit-frame-pointer -funroll-loops

[20] https://github.com/google/boringssl/tree/d7936c23cb9f3c4058d9cf6e3f503285d8024156

[21] https://github.com/armfazh/fld-ecc-vec/tree/7d8984d01b6c4e81a7b9680e981442f31fdc26e2

[22] https://github.com/openssl/openssl/tree/bc8c36272067f8443f875164831ce3a5a739df3f

Table 5.2 presents four evaluations for a limited set of Curve25519 implementations, namely Jasmin's ref4 and ref5, and SUPERCOP's amd64-64 and amd64-51. This performance analysis was carried on an Intel Westmere CPU, where five limbs implementations are faster due to the low latency and throughput of adc instructions. It is possible to observe that both Jasmin implementations perform slightly faster than the corresponding qhasm ones.

	Notes	scalarmult
JASMIN	ref4	256232
	ref5	222532
SUPERCOP	amd64-64	267028
	amd64-51	229360

Table 5.2: Curve25519: performance comparison on an Intel Westmere (i5-650).

5.3.4 Formally Verifying Curve25519

Figure 5.17 presents an EasyCrypt specification of Curve25519. This specification is close to RFC7748 [LHT16]. The entry point for this cryptographic operation is the operator scalarmult, which expects as inputs two 256-bit arrays, with k and u representing the scalar and an u-coordinate, respectively.

scalarmult starts by invoking the operator decodeScalar25519 to clear the specified bits on k. The operator decodeUCoordinate is then used to load the contents of u into a zp variable. zp consists of an element of the finite field $\mathbb{F}_q$ (for q prime). Although it is not shown in the corresponding figure, zp is defined as an instantiation of ZModP.ZModField, with the prime set to $2^{255} - 19$. ZModP theories are included in the standard distribution of EasyCrypt. The scalarmult operation continues by invoking montgomery_ladder and the final result is given to encodePoint, which encodes the result of "x_2 * (z_2^(p - 2))" (as in RFC7748) in a 256-bit array.

montgomery_ladder is defined as a foldl operation. In this case, foldl is a high-order operator that successively applies montgomery_ladder_step to the initial state, ((Zp.one, Zp.zero), (init, Zp.one)), as many times as the length of the list given by the expression (rev (iota_ 0 255)), where iota_ 0 255 represents a list containing the values from 0 to 254 and rev reverses this list. This corresponds to the order in which the bits of k are accessed. The described computation corresponds to one do-while (or while) loop in an imperative programming context.

The montgomery_ladder_step operator explicitly branches over the corresponding bit of k, corresponding to k.[ctr], where ctr corresponds to the current index, and, depending on whether the corresponding bit is set, the state, a pair of points, is swapped. Given that k is secret, the actual implementation is constant-time and does not perform any branch to

```
op decodeScalar25519 (k : W256.t) =
  let k = k[0 ← false] in
  let k = k[1 ← false] in
  let k = k[2 ← false] in
  let k = k[255 ← false] in
  let k = k[254 ← true ] in
      k.

op decodeUCoordinate (u : W256.t)
  = inzp (to_uint u).

op add_and_double
 (qx : zp)
 (nqs : (zp * zp) * (zp * zp)) =
  let x1 = qx in
  let (x2, z2) = nqs.`1 in
  let (x3, z3) = nqs.`2 in
  let a  = x2 + z2 in
  let aa = a * a in
  let b  = x2 + (- z2) in
  let bb = b*b in
  let e = aa + (- bb) in
  let c = x3 + z3 in
  let d = x3 + (- z3) in
  let da = d * a in
  let cb = c * b in
  let x3 = (da + cb)*(da + cb) in
  let z3 = x1 * ((da + (- cb))*(da + (- cb))) in
  let x2 = aa * bb in
  let z2 = e * (aa + (inzp 121665 * e)) in
      ((x2, z2), (x3, z3)).
```

```
op swap_pair (t : ('a * 'a) * ('a * 'a))
  = (t.`2, t.`1).

op ith_bit (k : W256.t) (i : int) = k[i].

op montgomery_ladder_step
 (k : W256.t)
 (init : zp)
 (nqs : (zp * zp) * (zp * zp))
 (ctr : int) =
  if ith_bit k ctr
  then swap_pair (add_and_double init
         (swap_pair (nqs)))
  else add_and_double init nqs.

op montgomery_ladder(init : zp, k : W256.t) =
  foldl (montgomery_ladder_step k init)
        ((Zp.one, Zp.zero), (init, Zp.one))
        (rev (iota_ 0 255)).

op encodePoint (q : zp * zp) =
  let q = q.`1 * (ZModpRing.exp q.`2 (P - 2)) in
     W256.of_int (asint q).

op scalarmult (k : W256.t) (u : W256.t) =
  let k = decodeScalar25519 k in
  let u = decodeUCoordinate u in
  let r = montgomery_ladder u k in
     encodePoint (r.`1).
```

Figure 5.17: Curve25519: Specification in EasyCrypt.

perform the swap. The specification was written as presented to keep it as human-readable as possible (but it increases the proof complexity). The state, swapped or not, is given to the add_and_double operator, also written as close as possible to the corresponding RFC.

The correctness proof for Curve25519 is organized in four hops. Each hop is used to approximate the specification to the concrete implementation. The first hop aims to prove the equivalence between the discussed specification, which abstracts many implementation details, and a different representation of the primitive closer to the corresponding implementation. This second encoding of the algorithm is still written using operators (i.e., functionally and not yet imperatively by using procedures). Several transformation steps are considered in this hop. The inversion, in the specification represented by the expression

```
op invert0(z1 : zp) : zp =
  let z2 = exp z1 2 in
  let z8 = exp z2 (2*2) in
  let z9 = z1 * z8 in
  let z11 = z2 * z9 in
  let z22 = exp z11 2 in
  let z_5_0 = z9 * z22 in
  let z_10_5 = exp z_5_0 (2^5) in
  let z_10_0 = z_10_5 * z_5_0 in
  let z_20_10 = exp z_10_0 (2^10) in
  let z_20_0 = z_20_10 * z_10_0 in
  let z_40_20 = exp z_20_0 (2^20) in
  let z_40_0 = z_40_20 * z_20_0 in
  let z_50_10 = exp z_40_0 (2^10) in
  let z_50_0 = z_50_10 * z_10_0 in
  let z_100_50 = exp z_50_0 (2^50) in
  let z_100_0 = z_100_50 * z_50_0 in
  let z_200_100 = exp z_100_0 (2^100) in
  let z_200_0 = z_200_100 * z_100_0 in
  let z_250_50 = exp z_200_0 (2^50) in
  let z_250_0 = z_250_50 * z_50_0 in
  let z_255_5 = exp z_250_0 (2^5) in
  let z_255_21 = z_255_5 * z11 in
  z_255_21 axiomatized by invert0E.
```

```
lemma eq_invert0 (z1 : zp) :
  invert0 z1 = invert_p z1.
proof.
  rewrite invert0E invert_pE /invert_p_p1
        /invert_p_p2 /invert_p_p3 //.
qed.

lemma eq_invert0p (z1 : zp) :
  invert0 z1 = exp z1 (P − 2).
proof.
  rewrite eq_invert0 eq_invert_p //.
qed.
```

Figure 5.18: Curve25519: Hop 1: Intermediate step to prove the inversion in EasyCrypt.

(ZModpRing.exp q.'2 (P - 2)), is implemented as a series of squares and multiplications using several temporary values. The first step to address such transformation is to decompose the previous expression in a series of exponentiation and multiplications and prove them equivalent. Given that the intermediate proof goals rapidly grew to unsustainable sizes (where the CPU time required to verify the proof was not compatible with an interactive proof environment), the unrolled inversion operation was first split into three operators, and intermediate correctness properties were proven on these. These intermediate properties were then used to prove the equivalence between ZModpRing.exp q.'2 (P - 2) and invert0, as shown in figure 5.18. The next step (not shown) was to replace the exponentiations from invert0 by sequences of calls to squares, to approximate, even more, the functional implementation from the low-level implementation.

Another interesting transformation performed in this first hop is how the state is swapped during the execution of the montgomery_ladder_step. In the specification, it is possible to observe that if k.[ctr] is 1 (true) the input pair is swapped before and after add_and_double. As such, the pair is kept in its original format at each entry and exit of montgomery_ladder_step. The low-level implementation only performs one swap for each montgomery_ladder_step execution (to save CPU time). If the pair is kept as it is after add_and_double is called,

and in the case that the pair is already in a swapped state in the next iteration, then the swap only occurs if the bit is 0 (false). The low-level implementation can achieve this by computing an XOR between the current bit and the previous bit, and the constant-time swap can be implemented using a series of XORs and masks based on this bit.

To replicate this behavior in this first hop, the state was extended with an additional boolean value to contain the pair's current status (becoming a triple), which states if it the first two elements are swapped. Some intermediate steps were necessary to prove the equivalence between the operator presented in figure 5.19 and the one previously discussed, from figure 5.17. As a small note, cswap is still encoded using an explicit if statement. The transformation from a non-constant time implementation into a constant-time swap is addressed in the fourth hop, which imports the EasyCrypt representation of the Jasmin implementation. The presented cswap operator is the specification for the concrete implementation. The operator select_triple_12 returns the first two elements of the state.

```
op select_triple_12 (t : ('a * 'a) * ('a * 'a) * γ) = (t.`1, t.`2).

op cswap( t : ('a * 'a) * ('a * 'a), b : bool ) =
  if b
  then swap_pair t
  else t.

op montgomery_ladder3_step(k : W256.t, init : zp, nqs : (zp * zp) * (zp * zp) * bool, ctr : int) =
  let nqs = cswap (select_triple_12 nqs) (nqs.`3 ^^ (ith_bit k ctr)) in
  let nqs = add_and_double1 init nqs in
  (nqs.`1, nqs.`2, (ith_bit k ctr)).
```

Figure 5.19: Curve25519: Hop 1: Montgomery ladder step.

One of the aspects that were necessary to consider was how the state is initialized by montgomery_ladder (not swapped) and how it is returned when the computation is completed (also not swapped). This is true (the state is in a non-swapped state after foldl terminates) when the first bit of k is set to 0 (false), which can be ensured in this context given that decodeScalar25519 is used.

The operand montgomery_ladder3_step has an additional input compared to the original definition, a boolean indicating if the points are swapped. Because of this, it was necessary to create a new lemma to reason about the properties of the foldl (for the montgomery_ladder proof), by establishing a relational invariant between the two versions of the state. The first step to tackle this is to prove that the extended state (triple) can always be reconstructed into the original state by swapping the points if necessary. The second step is to argue that, given that k.[0] is false, it is not necessary to perform any swap at the end. By approaching small problems at each step, the proof complexity is reduced.

The second hop in this proof implements a series of procedures to be proven equivalent to the previously discussed operators. Essentially, this hop transforms the functional specification into an imperative one. Many details are still abstracted in this EasyCrypt implementation of Curve25519. For instance, the swap operation is still not implemented following the constant-time discipline. Additionally, all arithmetic operations over zp elements are encapsulated. For instance, statements such as a ← b * c, where a, b, and c have type zp, are replaced by a ← mul (b, c). At this stage, these arithmetic procedures are simply defined as calls to the native operation—in the shown example, mul just contains a local variable declaration, for instance a, and the statement a ← b * c, followed by a return of a—but it prepares for the subsequent stages of the proof, where these simple procedure calls will eventually be replaced by a call to the *real* code. Figure 5.20 shows the montgomery_ladder procedure that is proven equivalent to the corresponding operator from the specification (implemented using a foldl).

```
proc montgomery_ladder (init' : zp, k' : W256.t) : zp * zp * zp * zp = {
  var x2 : zp;
  var z2 : zp;
  var x3 : zp;
  var z3 : zp;
  var ctr : int;
  var swapped : bool;
  x2 ← witness;
  x3 ← witness;
  z2 ← witness;
  z3 ← witness;
  (x2, z2, x3, z3) ← init_points (init');
  ctr ← 254;
  swapped ← false;
  while (0 ≤ ctr)
  { (x2, z2, x3, z3, swapped) ← montgomery_ladder_step (k', init', x2, z2, x3, z3, swapped, ctr);
    ctr ← ctr − 1;
  }
  return (x2, z2, x3, z3);
}
```

Figure 5.20: Curve25519: Hop 2: Montgomery ladder.

To relate the behavior of procedures with the corresponding operators we use Hoare logic. Figure 5.21 shows the lemma that states that proc montgomery_ladder performs the same computation as the op montgomery_ladder3. In this lemma, the statement k.[0] = false is included in the preconditions to make it clear that the final swap is unnecessary. This second hop concludes with the lemma eq_h2_scalarmult, also defined using Hoare logic, where it is proved that the procedure version of the scalarmult operation is the same as the operator scalarmult from the specification.

```
lemma eq_h2_montgomery_ladder (init : zp) (k : W256.t) :
  hoare [MHop2.montgomery_ladder : init' = init ∧ k[0] = false ∧ k' = k
       ⇒ ((res.`1, res.`2),(res.`3,res.`4)) = select_tuple_12 (montgomery_ladder3 init k)].
proof.
```

Figure 5.21: Curve25519: Hop 2: Lemma Montgomery ladder.

The third and fourth hops of this proof are intrinsically related but exist as separate units to isolate the contents of each one. All procedures implemented in the second hop are verified for termination during the third hop. Figure 5.22 shows the lemma ill_montgomery_ladder, and corresponding proof, which states that MHop2.montgomery_ladder is lossless (terminates with probability one).

For most procedures, this can be trivially proven given the absence of control-flow instructions, and, in those cases, the proof consists of just one call to one EasyCrypt tactic. Nonetheless, a lemma for this property exists for all existing procedures (which should be used in favor of simply proving the property in place whenever needed). By following this approach, if the underlying implementation changes and the proof for the termination is no longer trivial, only one lemma needs to be updated, and the remaining parts of the proof which rely on it continue to work. Also, in the context of the third hop, it is proved (using probabilistic Hoare logic) that with probability one, the procedures compute the same results as the corresponding operators.

```
lemma ill_decode_scalar_25519 : islossless MHop2.decode_scalar_25519.
proof. islossless. qed.

lemma eq_h3_decode_scalar_25519 k:
  phoare [ MHop2.decode_scalar_25519 : k' = k ⇒ res = decodeScalar25519 k] = 1%r.
proof. by conseq ill_decode_scalar_25519 (eq_h2_decode_scalar_25519 k). qed.

lemma ill_montgomery_ladder : islossless MHop2.montgomery_ladder.
proof.
  islossless. while true (ctr+1). move ⇒ ?. wp. simplify.
  call(_:true ⇒ true). islossless. skip; smt().
  skip; smt().
qed.

lemma eq_h3_montgomery_ladder (init : zp) (k : W256.t) :
  phoare [MHop2.montgomery_ladder : init' = init ∧ k[0] = false ∧ k' = k
       ⇒ ((res.`1, res.`2),(res.`3,res.`4)) =
          select_tuple_12 (montgomery_ladder3 init k)] = 1%r.
proof. by conseq ill_montgomery_ladder (eq_h2_montgomery_ladder init k). qed.
```

Figure 5.22: Curve25519: Hop 2: Lemma Montgomery ladder.

The fourth hop aims at proving the equivalence between the EasyCrypt modules extracted from the Jasmin implementation and the implementation from hop 2. Once this is done, the extracted code can be proven correct against the specification. Some of the equivalence proofs in this hop are work-in-progress. The proofs related to the arithmetic operations, namely, the multiplication, squaring, addition, and subtraction, are included in this set. The general intuition for proving the correctness of the arithmetic operations is that for some operations, such as the addition and subtraction, they produce results that are not equal to the corresponding zp operation, but the value is congruent modulo the prime being used. Given that the addition and subtraction outputs are used as inputs to the multiplication and squaring, this must also be accounted for. At the end of the scalarmult execution, a full reduction is performed to ensure that the returned value is actually the same and, as such, during the proof, this also needs to be accounted for. Other parts of the proof are already done, some of them by assuming that these operations behave as expected, such as it is the case of the add_and_double. The constant-time version of the swapping algorithm is also proven correct.

One of the reasons that justify the presented design for the proof, where the proof between a specification and an extracted implementation is organized in several hops that address different types of transformations, is maintainability. Although it is technically possible to prove the equivalence between a high-level specification (such as the one that was shown) and concrete implementations in an (almost) single step, any minor update in the Jasmin implementation could significantly increase the difficulty of updating the proof, given that the different stages would not be logically isolated. As an intuition, the constant time-swap can be implemented differently depending on the available instructions. If we consider a scenario where a second Curve25519 implementation is developed, and this second implementation only differs in the way that the swap is performed, it should be trivial to extend the previously existing proof to the new implementation. The same observation can be made for how internal variables are represented, as different implementations may require a different number of limbs.

Chapter 6

Conclusions and Future Work

There are several approaches to the implementation of high-speed cryptographic software. The best-performing implementations are developed using programming languages or tools that provide almost negligible abstraction. A carefully designed implementation developed using such tools or techniques can usually outperform the machine code automatically generated from high-level programming languages. The available C compilers that produce the best performing machine code are not formally verified, and this may represent a liability given that there are no guarantees about the produced machine code. The task of formally verifying such C compilers for functional correctness and preservation of the constant-time property is (more certainly than not) not happening any time soon. At the lowest level of abstraction and highest control is pure assembly programming. It is challenging to write, audit, and maintain the code at this level. As the level of abstraction lowers, the potential for producing implementations that outperform code generated by the best C compilers increases, but the probability of a developer, no matter the experience, inadvertently inserting subtle bugs also increases.

The sweet spot for producing high-speed implementations while keeping a very reasonable level of abstraction is implemented by the Jasmin programming language and its compiler. It allows having almost complete control of the generated assembly code while using a syntax similar to C. In some cases, due to the increased level of abstraction that is provided when compared to assembly programming, it enables performance improvements that are much more difficult to achieve in other contexts, as shown during chapter 5, for instance, in § 5.3.2. The correctness proof of the compiler and the sound connection to a verification framework, via an embedding of the Jasmin in EasyCrypt which is, essentially, a one-to-one map, addresses the problem of connecting the machine code that is executed with high-level specifications and security proofs. Other relevant properties addressed by Jasmin are memory safety, constant-time policy, and, soon, supporting counter-measures against misspeculation-based attacks with minimal cost. The work performed in the context of this

thesis significantly contributed to defining the current shape of the Jasmin programming language and its compiler and thus, completely fulfilled the original goal of this thesis:

"*bridging the gap between high-assurance and high-speed cryptographic implementations*"

In an ideal world, high-level specifications of any algorithm could be given to a formally verified compiler to generate architecture-dependent code that runs as fast as physically possible for any given CPU. This is not the case, nor will it be over the following decades. Perhaps during the subsequent decades, there will not even be a practical justification for all the spent resources on optimizing code, and all of this work will be replaced by low-cost machines that print microchips. However, while those times do not come, we can continue with the development of an entirely formally verified cryptographic library. The implementations and corresponding proofs developed in the context of this thesis are publicly available in libjc[1], a GitHub repository.

During the last couple of years, this project grew, and, as a result, the Formosa project was created (roughly) at the beginning of 2022. Formosa's[2] primary goal is to bring together the expertise required to improve the existing tools and methodologies and implement a high-speed and high-assurance cryptographic library. libjc will no longer be actively maintained. libjade[3] is the successor of libjc, and it is a high-speed high-assurance cryptographic library designed to fulfill the requirements of its potential users: support for multiple architectures, formally verified high-speed cryptographic implementations including state-of-the-art countermeasures against timing attacks.

One of the aspects that significantly contribute to the success of an open-source project is to have a community that benefits if the project continues to succeed. Because of this, we need to promote Jasmin. First, we need to increase the number of Jasmin developers by extending the available documentation and organizing Jasmin-related events. Second, we need to understand the requirements we must fulfill for our solutions to get adopted. Both parts are already work in progress, with promising results.

From a technical point of view, and to improve state of the art in this area, several features can be added to the Jasmin programming language. Supporting register arrays as arguments for local functions could be useful, for instance, to reduce the resulting code size of Curve25519 implementations with almost negligible overhead. Passing the arguments through pointers to the stack (reg ptr) or transforming register arrays into individual registers are the current possibilities to reduce the code size. However, by following this approach, some overhead and complexity are introduced, which can be avoided by the proposed feature. Nonetheless,

[1] `https://github.com/tfaoliveira/libjc`
[2] `https://formosa-crypto.org/`
[3] `https://github.com/formosa-crypto/libjade/`

code size will be reduced when migrating Curve25519 implementations from libjc to libjade. As an intuition, the current inversion algorithm currently represents 70% of the code size and just 10% of the time taken by the complete computation (which can also be improved, independently, using the algorithm as described in [BY19]), and the remaining code has roughly the same size qhasm implementations of Curve25519.

Currently, Jasmin exported functions are only allowed to receive inputs variables declared as registers. To pass a memory pointer to a Jasmin exported function, we use the type "reg u64" (AMD64). It would be interesting to extend the language for the exported functions to receive as inputs "reg ptr" variables, that point to arrays whose length could be statically known, or defined by another input, for instance, "(reg ptr u8[inlen] in, reg u64 inlen)". The need for this feature is particularly evident while implementing hash functions and cryptographic constructions that use hash functions. For instance, in an HMAC-SHA256 implementation from libjc[4], the function must receive an additional pointer to an external memory region (variable hkpadded) to avoid code duplication. If such an argument is not provided, two versions of the _sha256_blocks would need to be implemented, one for handling external memory and another for internal stack arrays (to compute the last blocks or to hash data from stack memory). Another solution would be to always copy the input data from external memory to internal memory and perform the computation on internal memory (at the expense of non-negligible CPU time). With such a feature, code duplication could be avoided and, for instance, function _sha256_blocks could take as input a given reg ptr that could point to external or internal memory. This feature is not easy to implement, as some assumptions on input pointers need to be considered, and many changes are required on the compiler and corresponding proofs, but it would reduce the complexity of Jasmin implementations and correctness proofs. Also related to this context, extending the language to support stack arguments could be useful, especially for the context of ABIs that do not specify as many register arguments as the System V AMD64.

In the author's opinion, the last two features are the ones that can benefit the most (short term) the development and maintainability of a formally-verified Jasmin library. Many other features and improvements are currently being discussed, and several of these have one aspect in common: they aim to increase the level of abstraction of the Jasmin programming language. Jasmin is, and always will be, a framework where the developer has control over the produced assembly code. However, in non-performance-critical contexts, some of this control may not be necessary, and Jasmin can be improved in many different ways (via new language constructions, additional compilation passes, new EasyCrypt theories) to reduce development time.

[4] https://github.com/tfaoliveira/libjc/blob/glob_array3/src/crypto_hash/sha256/common/sha256.jazz
https://github.com/tfaoliveira/libjc/blob/glob_array3/src/crypto_auth/hmacsha256/common/hmacsha256.jazz

References

[ABB+17a] 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 Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017.

[ABB+17b] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1989–2006. ACM, 2017.

[ABB+19] José Bacelar Almeida, Cecile 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 Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1607–1622. ACM Press, November 11–15, 2019.

[ABB+20a] 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*, pages 965–982, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press.

[ABB+20b] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, and Tiago Oliveira. Certified compilation for cryptography: Extended x86 instructions and constant-time verification. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *Progress in Cryptology - INDOCRYPT 2020: 21st International Conference in Cryptology in India*, volume 12578 of *Lecture Notes in Computer Science*, pages 107–127, Bangalore, India, December 13–16, 2020. Springer, Heidelberg, Germany.

[AL00] Kazumaro Aoki and Helger Lipmaa. Fast implementations of aes candidates. In *AES Candidate Conference*, pages 106–120. National Institute of Standards and Technology, 2000.

[AOK99] Kazumaro AOKI. Comments on NIST's Efficiency Testing for Round1 AES Candidates, 1999. URL: `https://csrc.nist.rip/encryption/aes/round1/comments/990415-kaoki1.pdf`.

[App14] Andrew W. Appel. *Program Logics - for Certified Compilers.* Cambridge University Press, 2014.

[B+08] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[Bar04] William C. Barker. Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. NIST Special Publication 800-67 Version 1, 2004. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-67ver1.pdf`.

[BB94] Eli Biham and Alex Biryukov. An improvement of Davies' attack on DES. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 461–467. Springer, 1994.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[BB12] William C. Barker and Elaine Barker. Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. NIST Special Publication 800-67 Revision 1, 2012. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-67r1.pdf`.

[BBDL+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of

https. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BCC+12] Daniel J Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. Usable assembly language for gpus: a success story. *Cryptology ePrint Archive*, 2012.

[BCLS15] Gilles Barthe, Juan Manuel Crespo, Yassine Lakhnech, and Benedikt Schmidt. Mind the gap: Modular machine-checked proofs of one-round key exchange protocols. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 689–718. Springer, 2015.

[BCS13] Daniel J Bernstein, Tung Chou, and Peter Schwabe. Mcbits: fast constant-time code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 250–272. Springer, 2013.

[BD97] Dileep Bhandarkar and Jason Ding. Performance characterization of the Pentium Pro processor. In *Proceedings Third International Symposium on High-Performance Computer Architecture*, pages 288–297. IEEE, 1997.

[BDG+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier López, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[BDL+12a] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012.

[BDL+12b] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30):320–337, 2009.

[Ber05a] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[Ber05b] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.

[Ber06a] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[Ber06b] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany.

[Ber07] Daniel J Bernstein. qhasm: tools to help write high-speed software, 2007. URL: `http://cr.yp.to/qhasm.html`.

[Ber08] Daniel J Bernstein. The salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[BGK76] Dennis Branstad, Jason Gait, and Stuart Katzke. Report of the Workshop on Cryptography in Support of Computer Security. Held at the National Bureau of Standards, September 21-22, 1976. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nbsir77-1291.pdf`.

[BGLB11] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security verifiable IND-CCA security of OAEP. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2011.

[BHK+17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017: 26th USENIX Security Symposium*, pages 917–934, Vancouver, BC, Canada, August 16–18, 2017. USENIX Association.

[BKS18] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. *hacspec: Towards Verifiable Crypto Standards: 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings*, pages 1–20. 01 2018.

[BL] DJ Bernstein and Tanja Lange. ebacs: Ecrypt benchmarking of cryptographic systems. `http://bench.cr.yp.to`.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467, 2016.

[bor21] Boringssl. `https://github.com/google/boringssl`, 2021.

[BR19] Elaine Barker and Allen Roginsky. Transitioning the Use of Cryptographic Algorithms and Key Lengths. NIST Special Publication 800-131A Revision 2, 2019. URL: `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf`.

[Bra75] Dennis Branstad. Encryption protection in computer data communications. In *4th Data Communications Symposium; October 7-9, 1975; Quebec City, Quebec, Canada*, pages 8–1, 1975.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72, 1991.

[BS92] Eli Biham and Adi Shamir. Differential cryptanalysis of the full 16-round DES. In *Annual International Cryptology Conference*, pages 487–496. Springer, 1992.

[BS08] Daniel J Bernstein and Peter Schwabe. New aes software speed records. In *International Conference on Cryptology in India*, pages 322–336. Springer, 2008.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/8298`.

[CHL+14] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying curve25519 software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–309, 2014.

[Cho15] Tung Chou. Sandy2x: New curve25519 speed records. In *International Conference on Selected Areas in Cryptography*, pages 145–160. Springer, 2015.

[Cop94] Don Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM journal of research and development*, 38(3):243–250, 1994.

[CPB+12] Shu-jen Chang, Ray Perlner, William E. Burr, Meltem Sönmez Turan, John M. Kelsey, Souradyuti Paul, and Lawrence E. Bassham. Third-round report of the sha-3 cryptographic hash algorithm competition. *NIST Interagency Report*, 7896:121, 2012.

[CS09] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the cell broadband engine. In *International Conference on Cryptology in Africa*, pages 368–385. Springer, 2009.

[Cur05] *Organizing DESCHALL*, pages 63–73. Springer New York, New York, NY, 2005.

[Dav78] Ruth M. Davis. The Data Encryption Standard in Perspective. In *Proceedings of the Conference on Computer Security and the Data Encryption Standard*, pages 4–13. National Bureau of Standards Special Publication SP 500-27, 1978.

[DH77] Whitfield Diffie and Martin E Hellman. Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer*, 10(6):74–84, 1977.

[DM95] Donald Davies and Sean Murphy. Pairs and triplets of DES S-boxes. *Journal of Cryptology*, 8(1):1–25, 1995.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[Dwo99] Morris Dworkin. SECOND ADVANCED ENCRYPTION STANDARD CANDIDATE CONFERENCE. *Journal of Research of the National Institute of Standards and Technology*, 104(4), 1999.

[Dwo00] Morris Dworkin. THIRD ADVANCED ENCRYPTION STANDARD CANDIDATE CONFERENCE, 2000. URL: `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/aes-development/aes3report.pdf`.

[EPG+19a] 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*, pages 1202–1219, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[EPG+19b] 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. IEEE, 2019.

[EYCP01] Adam J Elbirt, Wei Yip, Brendon Chetwynd, and Christof Paar. An fpga-based performance evaluation of the aes block cipher candidate algorithm finalists. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):545–557, 2001.

[FGH+19] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. In *Principles of Programming Languages (POPL 2019)*. ACM, January 2019.

[FHLD19a] Armando Faz-Hernández, Julio López, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–35, 2019.

[FHLD19b] Armando Faz-Hernández, Julio López, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)*, 45(3), 2019.

[GG13] Martin Goll and Shay Gueron. Vectorization of ChaCha stream cipher. Cryptology ePrint Archive, Report 2013/759, 2013. `https://eprint.iacr.org/2013/759`.

[HLMS22] Rasmus Holdsbjerg-Larsen, Mikkel Milo, and Bas Spitters. A verified pipeline from a specification language to optimized, safe rust. 2022.

[HMS+76] M Hellman, R. Merkle, R. Schroeppel, L. Washington, W. Diffie, S. Pohlig, and P Schweitzer. Results of an Initial Attempt to Cryptanalysis the NBS Data Encryption Standard. In *Report SEL-76-012*. Stanford Electronics Laboratory Menlo Park, CA, 1976.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. Guide to elliptic curve cryptography, 2004.

[IBM21] IBM. Cryptography for a Connected World. `https://www.ibm.com/ibm/history/ibm100/us/en/icons/cryptography/`, 2021. Accessed: 2021-03-04.

[IKM00] Tetsuya Ichikawa, Tomomi Kasuya, and Mitsuru Matsui. Hardware Evaluation of the AES Finalists. In *AES Candidate Conference*, volume 2000, pages 279–285. Citeseer, 2000.

[KBK12] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012.

[Kra17] Vlad Krasnov. How" expensive" is crypto anyway, 2017. URL: `https://blog.cloudflare.com/how-expensive-is-crypto-anyway/`.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security (rfc7748), 2016.

[Mat93] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 386–397. Springer, 1993.

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[MD19] Darius Mercadier and Pierre-Évariste Dagand. Usuba: high-throughput and constant-time ciphers, by construction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–173, 2019.

[Mei76] Paul Meissner. Report of the Workshop on Estimation of Significant Advances in Computer Technology. Held at the National Bureau of Standards, August 30-31, 1976. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nbsir76-1189.pdf`.

[MKB21] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust*. PhD thesis, Inria, 2021.

[MSL+20] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[MSW77] Robert Morris, Neil JA Sloane, and Aaron D Wyner. Assessment of the National Bureau of Standards proposed federal data encryption standard. *Cryptologia*, 1(3):281–291, 1977.

[NBB+01] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the Advanced Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology*, 106(3):511, 2001.

[NBD+99] James Nechvatal, Elaine Barker, Donna Dodson, Morris Dworkin, James Foti, and Edward Roback. Status report on the first round of the development of the Advanced Encryption Standard. *Journal of Research of the National Institute of Standards and Technology*, 104(5):435, 1999.

[NIS99] Security Technology Group Information Technology Laboratory NIST. NIST's Efficiency Testing for Round1 AES Candidates, 1999. URL: `https://csrc.nist.rip/encryption/aes/round1/conf2/NIST-efficiency-testing.pdf`.

[NNS10] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *International Conference on Cryptology and Information Security in Latin America*, pages 109–123. Springer, 2010.

[OLH+17] Thomaz Oliveira, Julio López, Hüseyin Hışıl, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-) compute a ladder. In *International Conference on Selected Areas in Cryptography*, pages 172–191. Springer, 2017.

[Oli12] Tiago Oliveira. *Verificação de software criptográfico de elevado desempenho*. PhD thesis, 2012. `https://github.com/tfaoliveira/qhasm-translator/`.

[oS73] National Bureau of Standards. Cryptographic Algorithms for Protection of Computer Data during Transmission and Dormant Storage - Solicitation of Proposals. Federal Register, Vol. 38 No. 93 page 12763, Tuesday, May 15, 1973.

[oS74] National Bureau of Standards. Encryption Algorithms for Computer Data Protection - Reopening of Solicitation. Federal Register, Vol. 39 No. 167 page 30961, Tuesday, August 27, 1974.

[oS75a] National Bureau of Standards. Encryption Algorithm for Computer Data Protection - Request for Comments. Federal Register, Vol. 40 No. 52 pages 12134–12139, Monday, March 17, 1975.

[oS75b] National Bureau of Standards. Federal Information Processing Data Encryption - Proposed Standard. Federal Register, Vol. 40 No. 149 pages 32395, Friday, August 1, 1975.

[oS81] National Bureau of Standards. Guidelines for Implementing and Using the NBS Data Encryption Standard. Federal Information Processing Standards, Pub. 74, 1981. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub74.pdf.

[oS88] National Bureau of Standards. Data Encryption Standard. Federal Information Processing Standards, Pub. 46-1, 1988. URL: https://csrc.nist.gov/CSRC/media/Publications/fips/46/1/archive/1988-01-22/documents/NBS.FIPS.46-1.pdf.

[oST93] National Institute of Standards and Technology. Data Encryption Standard. Federal Information Processing Standards, Pub. 46-2, 1993. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub46-2.pdf.

[oST97a] National Institute of Standards and Technology. Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard. Federal Register, Vol. 62 No. 1 pages 93-94, Thursday, January 2, 1997. URL: https://www.federalregister.gov/d/96-32494.

[oST97b] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard. Federal Register, Vol. 62 No. 177 pages 48051-48058 Friday, September 12, 1997. URL: https://www.federalregister.gov/d/97-24214.

[oST97c] National Institute of Standards and Technology. Comments on Proposed AES Minimum Acceptability Requirements And Evaluation Criteria, 1997.

[oST98] National Institute of Standards and Technology. Request for Comments on Candidate Algorithms for the Advanced Encryption Standard (AES). Federal Register, Vol. 63 No. 177 pages 49091-49093 Monday, September 14, 1998. URL: https://www.federalregister.gov/d/98-24560.

[oST99a] National Institute of Standards and Technology. Data Encryption Standard. Federal Information Processing Standards, Pub. 46-3, 1999. URL: https://csrc.nist.gov/CSRC/media/Publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf.

[oST99b] National Institute of Standards and Technology. Public Comments Regarding The Advanced Encryption Standard (AES) Development Effort - Round 1 Comments, 1999. URL: https://csrc.nist.gov/CSRC/media/Projects/

Cryptographic-Standards-and-Guidelines/documents/aes-development/R1comments.pdf.

[oST99c] National Institute of Standards and Technology. Request for Comments on the Finalist (Round 2) Candidate Algorithms for the Advanced Encryption Standard (AES). Federal Register, Vol. 64 No. 178 pages 50058-50061 Wednesday, September 15, 1999. URL: https://www.federalregister.gov/d/99-24014.pdf.

[oST00a] National Institute of Standards and Technology. AES3 Conference Feedback Form - Summary, 2000. URL: https://csrc.nist.rip/encryption/aes/round2/conf3/AES3FeedbackForm-summary.pdf.

[oST00b] National Institute of Standards and Technology. Public Comments Regarding The Advanced Encryption Standard (AES) Development Effort - Round 2 Comments, 2000. URL: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/aes-development/R2comments.pdf.

[oST01] National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards, Pub. 197, 2001. URL: https://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[oST05] National Institute of Standards and Technology. Announcing Approval of the Withdrawal of Federal Information Processing Standard 46–3. Federal Register, Vol. 70 No. 96 pages 28907-28908, Thursday, May 19, 2005. URL: https://www.federalregister.gov/d/05-9945.

[oST07] National Institute of Standards and Technology. Announcing the Development of New Hash Algorithm(s) for the Revision of Federal Information Processing Standard (FIPS) 180–2, Secure Hash Standard. Federal Register, Vol. 72 No. 14 pages 2861-2863, Tuesday, January 23, 2007. URL: https://www.federalregister.gov/d/E7-927.

[PBP+20] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. Haclxn: Verified generic simd crypto (for all your favourite platforms). In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 899–918, 2020.

[PPF+20] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon

Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy*, pages 983–1002, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press.

[PZR+17] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in f. *Proc. ACM program. lang.*, 1(ICFP):17–1, 2017.

[RD99] Edward Roback and Morris Dworkin. First advanced encryption standard (aes) candidate conference–ventura, ca, august 20-22, 1998. *Journal of Research of the National Institute of Standards and Technology*, 104(1):97, 1999.

[SKW+99] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Performance comparison of the AES submissions. In *Second AES Candidate Conference*, pages 15–34, 1999.

[Smi71] Smith, J. L. The Design of Lucifer, A Cryptographic Device for Data Communications. Technical report, IBM Thomas J. Watson Research Center, 1971.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[ZBPB17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

Appendix A

The Role of Standards in Cryptography

A.1 An overview of the Data Encryption Standard

The origin of the Data Encryption Standard (DES) can be traced back to 1968 when the National Bureau of Standards (NBS), currently known as the National Institute of Standards and Technology (NIST), initiated a study to understand computer security needs in the US Government, its companies and citizens [IBM21]. Thereafter, in May of 1973, NBS published the first solicitation of proposals for an algorithm that could be standardized and used to protect computer data in transmission or storage, mainly motivated by the increasing amount of digital communication [oS73] — later in 1977, Ruth Davis, director of the Institute for Computer Sciences and Technology from NBS, confirmed that this first solicitation did not had the desired outcome: several proposals to develop new cryptographic algorithms were submitted, but none of them would constitute a viable solution that could go through the process of standardization [Dav78]. It is also mentioned in the cited report that the best solution, from a theoretical point of view, required the usage of two infinite tapes filled with random characters that could be added and subtracted to a given message in order to encrypt and decrypt it. Unfortunately, suppliers of infinite tapes were not available at the time, making this solution unviable.

One year later, in August of 1974, the solicitation was reopened "*in order to ensure that a full opportunity to submit algorithms for consideration is accorded to all parties*" [oS74]. NBS received many proposals, but most of them were considered too specific or ineffective for standardization[1]. The candidate that stood out was Lucifer [Dav78], developed by IBM Research Center in the late '60s and early '70s and, at the time of this second solicitation, already being used by the Lloyds Bank in its cash-dispensing machines. A description of

[1] To the best of my knowledge, the complete list of submissions is not available.

Lucifer can be found in a technical report signed by J. L. Smith from IBM that dates to April 1971 [Smi71]. This document was declassified in February of 1972. Lucifer, as presented in the report, was a cryptographic device that allowed the encryption and decryption of messages, which were internally split into 16-byte blocks, using a 128-bit secret key. The cited report contains the device's physical description and Lucifer's encryption and decryption algorithms' specification. As a curiosity, this machine would take 200 microseconds to encrypt a 16-byte message.

The National Security Agency (NSA) conducted a technical analysis of the candidate at the request of NBS to conclude that "*no shortcuts or secret solutions were found*", confirming its suitability to be used in the upcoming standard [oS81]. In March of 1975, the proposed DES algorithm was published in the Federal Register [oS75a] along with a request for comments that could be submitted until mid-May 1975. In August of that same year, the previously published algorithm was proposed to be a standard [oS75b]. Comments were also requested in this second announcement and should be sent within 90 days. Many cryptographers exposed their concerns regarding the security of the algorithm. One of these concerns was related to the insufficient key length and the corresponding security implications: the original design of the proposal, Lucifer, used (at least in one of its versions) 128-bit keys and the algorithm proposed to become a standard used 56-bit keys.

Interestingly, the proposal explicitly mentioned a 64-bit key: "*Introduction. The algorithm is designed to encipher and decipher blocks of data consisting of 64 bits under control of a 64-bit key.*"; and also, "*(...) a different block K of key bits is chosen from the 64-bit key designated by KEY*" [oS75a]. The main issue is that, since 8 of the 64 bits may be used for error control, the number of possible keys was actually 2^{56}. Although there is a mention of this fact in the proposal's appendix, "*One bit in each eight-bit byte of the KEY may be utilized for error detection in key generation, distribution, and storage. Bits 8, 16,..., 64 are for use in assuring that each byte is of odd parity.*", Robert Morris et al. argued that this was misleading, since an inattentive reading of the technical documentation could leave the impression that there were 2^{64} possible keys [MSW77]. It was also reported by [MSW77] that Dennis Branstad from NBS,wrote a paper describing DES where it also states that the key had 64 bits [Bra75].

Whitfield Diffie and Martin Hellman also had strong objections regarding the chosen key length, stating that its size was carefully chosen such that only the NSA, and no one else, could break it. They claimed that, at the time of the standard's publication, it would be possible to build a specially purposed machine for $20 million[2] to break the proposed algorithm in just 12 hours (known-plaintext attack). They predicted that a specially-purposed chip, to test if the encryption of a known-plaintext produces the expected ciphertext, could be built at a reasonable cost. They argued that such a chip could be designed to only take 1 microsecond

[2]Equivalent to roughly $87 millions in 2021.

per tested key, given that there were almost no input-output operations involved — which were very expensive at the time and, as a matter of fact, still are.

Such chip could be loaded, for instance, at the beginning of the day, with the plaintext, the corresponding ciphertext, and an initial secret key that would be iterated. If a matching ciphertext is found, it would stop the search and write the candidate key to some output mechanism. If 1 million of those chips were put in parallel and each one of them was initially loaded with different keys to cover different key spaces, then this machine would be able to test 10^9 keys per second or roughly $2^{56.26}$ over 24 hours and, thus, being able to break a key every 12 hours, on average. They estimate that each chip would cost $10 to produce on such scale, and a factor of 2 was considered for design and deployment costs. IBM studied this possibility and concluded that it could build such machine and have it delivered by 1981, for $200 million [MSW77, DH77].

The other main concern was related to the unavailable documentation regarding the mathematical foundations of the substitution boxes. In a preliminary study, conducted by M. Hellman et al. [HMS+76] and developed in just one month, it was stated that a suspicious structure was found in the S-boxes and it could be due to: accidental weakness; intentionally inserted trap-door; or there could be no weakness at all. Later in that report, more precisely in section V, "*DES's Structure*", the following statement can be found:

> ''Despite an initial division of opinion, our group is now convinced that the DES S-Boxes were carefully chosen with certain structures in mind.''

In order to address these concerns, NBS organized two workshops in late August and September of 1976. The reports of these workshops are available in [Mei76] and [BGK76]. The first workshop, "*Estimation of Significant Advances in Computer Technology*", was mainly focused on discussing future advances in technology that could impact the security of DES. Part of the discussion that happened in the workshop, and transcribed in [Mei76], can be defined by the following question: Would, when, at what cost, and with what probability, will it be possible to build a specially purposed machine designed to perform an exhaustive key search that allows to compromise DES? Several scenarios were studied, but only two would allow for a key exhaustion time close to what W. Diffie and M. Hellman had predicted: 24 hours for key exhaustion time, an estimated cost between $50 and 72 million, success probability between 10% and 20%, depending on technological advances, and could only be delivered roughly fourteen years later, in 1990. The $200 million machine from IBM was not included for discussion as this information was shared outside of the workshop [MSW77, DH77]. It is also stated in [DH77] that IBM had withdrawn the study and that the workshop's conclusions should be accepted. W. Diffie and M. Hellman maintained their original position, which is thoroughly discussed in [DH77].

The second workshop, "*Cryptography in Support of Computer Security*", was attended by

42 people from the Federal Government, industry, and universities, and it was organized to collect expert opinions about the mathematical and statistical characteristics of DES algorithm. The problems related to the insufficient length of the 56-bit key were discussed. During the introductory session, [BGK76], M. Hellman claimed that he had information that, while IBM was responsible for the algorithm's design, the key length was set by NSA. W. Tuchman, from IBM, replied that the algorithm was published as submitted. There was an evident concern regarding the secret key length and its implications: from the industry reports of a client that would only update the secret key one time each month or some others that would be introducing the key via a terminal, using visible characters, which would result in 48 bits of entropy, unanimously considered insecure. In response to these cases, one attendee suggested that users should be educated, and if, after they get their education, they still use their wife's name as a secret key, it is their fault.

Another topic that also dominated discussions in this second workshop was the non-random substitution boxes, with W. Tuchman confirming that the S-Boxes were indeed not random to improve security. However, the details were classified and could not be revealed: he stated that while getting approval to export the algorithm, IBM discovered that classified design principles were used when requested by NSA not to make these details public. Other operational aspects, such as key distribution, were discussed during the workshop.

Despite all the controversy, the Data Encryption Standard was published in 1977, reaffirmed in 1983 with no changes, reaffirmed again in 1988 with minor changes [oS88, pg. 4, item 18], updated in 1993 to allow for software implementations [oST93, pg. 3, item 12] with the last revision happening in 1999 to only allow Single DES for legacy purposes and to introduce Triple-DES (TDEA) as a FIPS-approved algorithm [oST99a, pg.4 item 12]. FIPS 46-3 was withdrawn in 2005 [oST05] but TDEA was still approved by NIST and planned to co-exist with AES until 2030 [Bar04, BB12]. After the Sweet32 attack was published in 2016 [BL16] efforts were made to restrict the keying options and the maximum number of blocks that could be encrypted with the same key bundle. Three-key TDEA encryption is currently disallowed after 2023 [BR19].

As final remarks for this first part of the introduction, in 1991 E. Biham and A. Shamir published a paper on differential cryptanalysis [BS91] claiming that the DES algorithm combined with random S-boxes is easier to break, and even small changes in one of the S-Boxes can significantly reduce the attack complexity. This indicates that both IBM and NSA were aware of such attacks, which was later confirmed by D. Coopersmith in 1994 [Cop94]. Also according to [Cop94], the public releasing of information about differential cryptanalysis, at the time of the standardization process, could be used to attack many ciphers from that period and, in general, to weaken the US advantage in the field of cryptography. Nonetheless, I would like to highlight the following quote from [HMS+76]:

‘‘(...) it is poor security practice to trust a system whose design and

```
certification will not be described.''
```

The downfall of the Data Encryption Standard

In the early 90's, several papers on cryptanalysis were published, with many of them targeting DES. As previously mentioned, differential cryptanalysis was made public [BS91, BS92] and also a linear cryptanalysis method was published in 1993 by M. Matsui [Mat93] which enabled to break the 16-round DES with 2^{47} known plaintexts. Other works were also published during this period and, although these attacks could not be considered very practical due to the non-negligible amount of plaintexts or ciphertexts required, they presented significant improvements when compared to brute-force attacks [DM95, BB94]. Nevertheless, exhaustive key searches were becoming more feasible. In late January of 1997, at the RSA Conference held in San Francisco, California, RSA Security launched a series of challenges to draw attention to the lack of security of the Data Encryption Standard: a ciphertext was published and the first person or group to discover the original plaintext would win the $10000 prize. The first challenge was solved in 96 days. The second and third challenges, which happened in 1998, took 39 days and 56 hours, respectively, to be solved. The fourth and last one, launched in January of 1999, was concluded in roughly 23 hours. The recovered plaintext was:

```
``See you in Rome (second AES Conference, March 22-23, 1999)''
```

A.2 An overview of the Advanced Encryption Standard

Right before the start of the previously mentioned series of challenges, on the 2nd of January of 1997, an announcement from NIST was published in the Federal Register to announce the beginning of a process to develop the Advanced Encryption Standard (AES) [oST97a]. This new standard would replace the 20-year-old DES by providing a new algorithm to protect sensitive government data into the next century. A multi-year transition period was expected since, in NIST's view, the security level provided by DES was still considered adequate for many applications. This process was initiated with a request for comments from the public and organizations to understand their needs. The announcement also included an initial set of requirements, mainly to guide future candidates as proposals were not accepted at that time: 1) AES should be a publicly defined symmetric block cipher whose design allows for increasing the key length as necessary; 2) hardware and software implementations would be considered; 3) candidates would be evaluated under several criteria, being security, computational efficiency, and design simplicity some of them.

After receiving comments until early April 1997, with some exposing concerns about the International Traffic in Arms Regulations and non-US experts' participation on the AES

contest [oST97c], the "*Submission Requirements Workshop*" was held on the 15th of the same month to discuss the evaluation requirements and procedure. The call for proposals was later published on the 12th of September of 1997 [oST97b], soliciting submissions until the 15th of April of the following year, 1998. These would be subject to a preliminary review, performed by NIST, to check for the submissions' completeness: the feedback would be given until the 15th of May, with a final deadline on the 15th of June. The evaluation criteria were now detailed and organized in three sections: *Security*, *Cost*, and *Algorithm and Implementations Characteristics*.

Regarding security, the most important property for any cryptographic component, several factors would be taken into consideration: level of security of the algorithm when compared to other proposals; resistance against known attacks; the mathematical soundness of the construction; and also any factor submitted by experts during the review process that could impact the security claims of the submitters. In the cost section, computational efficiency or, in other words, the speed at which the algorithm would be able to perform, usually measured in CPU cycles or, in some cases, throughput per second, would be evaluated by NIST. Memory requirements would also be evaluated, and, as a general rule, the less required memory, the better. The contest would have two different evaluation rounds, with the first one being mainly focused on software implementations and characteristics such as code size and random access memory (RAM) requirements. The second round would focus on hardware implementations, with the evaluation being guided by the number of gates necessary to implement the proposal. As for algorithms and implementations characteristics: a flexible algorithm that could accommodate different key and block lengths, yield efficient implementations across different platforms, and, in general, with a simple design. It is also noted in the call that more than one candidate algorithm can be standardized if some proposals can complement each other in different scenarios.

The candidates' complete list was announced at the first AES candidate conference in mid-August of 1998 [RD99]. From the 21 submitted packages, six were considered incomplete, and the remaining 15 were successfully validated. The list corresponding to the six rejected candidates was announced at the conference and published on NIST's website. In total, submitters from 12 different countries were involved: USA had five submissions; Canada had two; and the remaining countries were involved with at least one. Some of the submitters' claims were considered very promising: some candidates claimed to be more secure than Triple-DES while performing faster than Single DES. A request for comments was made at the conference and later officiated by [oST98]. NIST was seeking the assistance of the public, researchers, and several others to select the best five candidates or, depending on the point of view, to exclude at least ten submissions for the second evaluation round. Comments could be submitted until the 15th of April of 1999. In between these dates, the second AES conference was held, as previously noted, in Rome on March 22-23, 1999.

The second AES conference report [Dwo99] presents a general overview of the conference's events, from the surveys' presentations to the cryptanalysis efforts that were made to study the candidates. In total, 28 papers were submitted and 21 were presented. The performance evaluation was also discussed during the conference. The first survey, presented by James Foti from NIST, concluded that the fastest algorithms for a 32-bit processor, C implementations, and 128-bit keys were the following: CRYPTON; MARS; RC6; Rijndael and Twofish. Further evaluation from NIST for 8 and 64-bit platforms was reported to be required and it would be performed during Round 2. In the following survey presentation, Bruce Schneier, one of the submitters of Twofish, alerted to the impact that larger keys could have on performance and also suggested that comparisons should be using optimized assembly implementations since that, in environments where speed is a critical factor, low-level implementations would be used. B. Schneier also reported the following algorithms to perform better in a DEC Alpha, a 64-bit RISC processor: DFC, Rijndael, Twofish, and HPC.

In another survey, Eli Biham, one of the designers of Serpent, spoke in favor of a "fair speed/security" evaluation method: Serpent was designed to have a considerable security margin with the authors claiming that, although the submitted candidate performed 32 rounds, 16 rounds would be enough to provide the desired security level. By using its proposed evaluation method, the top candidates regarding speed would be, in order, the following: Twofish, Serpent, MARS, Rijndael, and CRYPTON. When compared with the conclusions from the study performed by James Foti from NIST, RC6 would be removed from the top five candidates, and Serpent would be included. In general, performance evaluation of cryptographic algorithms has always been a topic of concern and debate, as it can be observed next.

After the second AES conference, comments were still being received and were later published for public consultation [oST99b]. During this period, many noteworthy comments were made, being one of them the recommendation for the Round 1 finalists sent by the IBM's AES Team: MARS, RC6, Twofish, Serpent, and Rijndael. They also recommended that only one final winner should be chosen, alerting that the increased complexity of a multi-winner scenario would not necessarily compensate the benefits. Later that year, the AES Round 1 report was published [NBD+99], to summarize Round 1 results and announce the five finalists: MARS, RC6, Rijndael, Serpent, and Twofish. Regarding the ten excluded candidates and the security analysis conclusions: major attacks were found in five candidates, and lesser attacks in the other five. Coincidentally, the candidates that suffered major attacks were also the slowest ones [NBD+99].

Regarding the performance analysis discussed in Round 1 report [NBD+99]: in the call for proposals for the AES contest [oST97b], NIST announced that the benchmarking platform would be an IBM-compatible PC equipped with an Intel Pentium Pro Processor clocked at 200MHz, 64MB of RAM, Borland C++ 5.0 compiler and running Windows 95. As a

curiosity, it had an L1 8KB data cache, 8KB instruction cache, and, according to the Intel ARK website, it was available with three different L2 cache sizes throughout its life cycle: 256KB, 512KB, and 1MB. The announced specifications did not include the L2 cache size in the description. According to [BD97], the mentioned CPU model has a 14 stage pipeline, implements an out-of-order execution model and speculative execution.

In the AES Round 1 report, in the appendix section, several tables summarizing the performance of the 15 candidates can be found. These tables contain the data collected by several authors across different platforms and environments. The data collected by NIST using the aforementioned platform, and presented during the second AES conference [NIS99], was not included in these tables and cannot be found in the report. We are now going to compare and discuss some measurements from NIST's presentation, which were not included in the final report, with measurements collected by B. Schneier et al. [SKW+99], included in Table 1 of AES Round 1 report. The full specifications of the benchmarking environment for the Schneier's et al. results cannot be found in the original paper, but the CPU is reported to be the same. NIST's measurements are presented first, followed by a slash, followed by Schneier's et al. measurements. To encrypt one block of data, 16 bytes, with a 128-bit key, using 32-bit C implementations, the reported speed in clock cycles for the finalists were: 807/390 cycles for MARS; 636/260 cycles for RC6; 809/440 cycles for Rijndael; 1629/1030 cycles for Serpent and 973/400 cycles for Twofish.

There are significant differences between the measurements: for instance, the ratio between NIST's and Schneier's et al. measurements for Rijndael is 1.84 and the maximum and minimum ratios that can be found for this set are 2.45 (RC6) and 1.58 (Serpent), with a total average of 2.07. A small part of these differences could be related to the NIST's API: in a comment sent to NIST on the 15th of April 1999 [AOK99], Kazumaro Aoki, one of the authors of an AES candidate, exposed its concerns about the performance evaluation methodology, noting that there were some discrepancies between the presented measurements.

To continue the discussion on this subject, we will use Rijndael as our example: in slide 16 of [NIS99], NIST reported that it takes 809 cycles to encrypt one block of data, roughly corresponding to 50 cycles per byte (cpb). In the same slide, it is mentioned that the function that corresponds to the NIST's API, when called with a NULL cipher, takes 41 cycles which, if we assume this to be the overhead, it corresponds to approximately 2.7 cpb for this context. Slide 21 of this same presentation reports that Rijndael has a throughput of 22942 Kb/s when 1MB of data is encrypted, which, in a CPU clocked at 200MHz, corresponds to something in the range of 68-70 cpb depending if we consider one Kb to be 1000 or 1024 bits.

The discrepancy between reported CPU cycles for one block and throughput also exists for other candidates: from the total of 15, 11 performed worse when comparing single block encryption with 1MB encryption, with all differences ranging from -67% (penalty) until 35% (improvement), with a total average of -18.5%. For Rijndael, the performance difference is

-38%. This effect could be related to some cache effects, as suggested by Aoki. An increasing number of cache misses, as plaintext data increases, can be considered the most likely cause to justify these differences but, since penalties were not similar across all implementations or seem to show any pattern, it is difficult to withdraw any conclusions on the subject.

Regarding the 809/440 cycles, or 50/27 cpb reported by NIST and Schneier et al. for Rijndael: the estimated 2.7 cpb overhead from the NIST's API does not seem to be the only plausible cause to justify a 23 cpb difference; possible causes for this discrepancy may include differences in compilers or compiler versions, CPU model or caches sizes, slightly different implementations or benchmarking methodologies. Overall, measuring and comparing software is an intricate task, sometimes seen as a reasonably simple one, whose results are often affected by minor or overlooked details. The comparison complexity increases significantly when evaluation must be done on multiple platforms and using different environments.

The second round of the AES contest officially began on the 15th of September 1999 [oST99c]. Comments were to be submitted until the 15th of May 2000. The third AES conference was also announced, and it would be held in New York in mid-April 2000. The third conference report [Dwo00] provides an overview of the topics that were discussed. As previously stated, hardware implementations were the central focus of this round. There were two sessions dedicated to this topic during the conference: "*Field-Programmable Gate Array (FPGA) Evaluations*" and "*Application-specific Integrated Circuit (ASIC) Evaluations*".

In the context of the former, Adam Elbirt presented the results of implementing the encryption operation of four finalists in an FPGA device (MARS was not considered in this study due to timing constraints) [EYCP01]: when considering implementations optimized for throughput, not area, Serpent exhibited the best performance results in the feedback and non-feedback modes, achieving, for instance, a throughput of 444.2 Mbit/s in feedback mode. The second-best candidate, Rijndael, performed at 300.1 Mbit/s, Twofish at 127.7 Mbit/s, and RC6 at 126.5 Mbit/s. Rijndael needed less area to be implemented and achieved the best ratio between throughput and used space for feedback mode of operation, but in non-feedback mode, Serpent was also the best candidate according to this metric of evaluation. The device used to produce the reported values was a Xilinx Virtex XCV1000BG560-4.

Regarding ASIC evaluations and feedback mode of operation, Tetsuya Ichikawa reported that Rijndael could achieve 1.95 Gbit/s and was followed by Serpent, which performed at 931 Mbit/s. Twofish had a throughput of 394 Mbit/s, and MARS and RC6 had roughly the same performance, 226 Mbit/s and 204 Mbit/s, respectively. Under the same conditions, DES and Triple-DES had a throughput of 1.16 Gbit/s and 407 Mbit/s [IKM00].

In a different session, "*Platform-specific evaluations*", Aoki reported that Rijndael was the candidate that benefited the most with the MMX instruction set: its design allowed for parallelism, which contributed to an implementation that would perform 28% faster when compared to the best-known implementation of this cipher. Other candidates are reported to

not benefit as much as Rijndael [AL00]. In "*Cryptographic Properties and Analysis*" session, attacks on reduced rounds versions of some candidates were presented and discussed, but no major or significant breakthroughs were made.

In the final session of the conference, "*Algorithm Submitter Presentations*", submitters were invited to give a presentation and then answer questions from the audience. All submitters highlighted the strengths of their ciphers and also design motivations. For instance, Ross Anderson, one of Serpent's designers, emphasized that the proposed algorithm was designed with security in mind and thus being able to provide security well into the next century, as stated in the AES first announcement [oST97a]. Serpent also provided satisfactory performance when compared to Triple-DES across different platforms. When submitters were asked which proposal they would select as the winner of the AES contest, other than their own, Vicent Rijmen, submitter of Rijndael, indicated RC6 as his option and the remaining four indicated Rijndael, if extended to 18 or more rounds — Rijndael performed 10, 12 or 14 rounds for 128, 192 and 256-bit keys.

At the end of the conference, NIST distributed a feedback form to all attendees, with a total of 246 excluding NIST personnel, and received back 167. For the question "*Which algorithms definitely SHOULD be selected for the standard*", there were 86 answers indicating Rijndael, and 59, 31, 23, and 13, for Serpent, Twofish, RC6, and MARS, respectively [oST00a]. The total sum being 212 means that a non-negligible number of people considered more than one algorithm to be a good option. After the Round's 2 comments were received and published [oST00b], the final report was made available on the 2nd of October of 2000 and later included in NIST's journal [NBB+01]. It announced that the Rijndael would be proposed as the Advanced Encryption Standard. In the final remarks of the mentioned report, it is stated that "*none of the finalists is outstandingly superior to the rest*". After due process, in 2001, FIPS 197 was published [oST01]. The design details of Rijndael are thoroughly discussed in the book "*The design of Rijndael: AES — the Advanced Encryption Standard*" written by the authors of the cipher, Joan Daemen and Vincent Rijmen [DR02].