Universidade Estadual de Campinas
Instituto de Computação

Hayato Fujii

Efficient Curve25519 Implementation for ARM Microcontrollers

Implementação Eficiente da Curve25519 para Microcontroladores ARM

CAMPINAS
2018

Hayato Fujii

Efficient Curve25519 Implementation for ARM Microcontrollers

Implementação Eficiente da Curve25519 para Microcontroladores ARM

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Diego de Freitas Aranha

Este exemplar corresponde à versão final da Dissertação defendida por Hayato Fujii e orientada pelo Prof. Dr. Diego de Freitas Aranha.

CAMPINAS
2018

Agência(s) de fomento e nº(s) de processo(s): CAPES; FUNCAMP
ORCID: https://orcid.org/0000-0002-1025-4536

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

F955e

Fujii, Hayato, 1992-
Efficient Curve25519 implementation for ARM microcontrollers / Hayato Fujii. – Campinas, SP : [s.n.], 2018.

Orientador: Diego de Freitas Aranha.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia de chaves públicas. 2. Criptografia de curvas elípticas. 3. Microprocessadores ARM. I. Aranha, Diego de Freitas, 1982-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Implementação eficiente da Curve25519 para microcontroladores ARM
Palavras-chave em inglês:
Public key cryptography
Elliptic curves cryptography
ARM microprocessors
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Diego de Freitas Aranha [Orientador]
Marco Aurélio Amaral Henriques
Julio César López Hernández
Data de defesa: 22-05-2018
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Hayato Fujii

Efficient Curve25519 Implementation for ARM Microcontrollers

Implementação Eficiente da Curve25519 para Microcontroladores ARM

Banca Examinadora:

- Prof. Dr. Diego de Freitas Aranha
  Instituto de Computação / UNICAMP
- Prof. Dr. Julio César López Hernández
  Instituto de Computação / UNICAMP
- Prof. Dr. Marco Aurélio Amaral Henriques
  Faculdade de Engenharia Elétrica e de Computação / UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 22 de maio de 2018

The world is full of obvious things which nobody by any chance ever observes.

(Sir Arthur Conan Doyle)

Acknowledgements

To my supervisor, Diego, whose patience and dedication in sharing experience and knowledge pointed me in the directions leading up to this work.

To my parents, Américo and Paweena, to the inspiration to push even more the borders of human knowledge.

To Renna, my eternal girlfriend (soon to be wife) supporting me, either by understanding the lack of my presence at home or abdications needed for further improve our lifes.

To the Laboratory of Security and Applied Cryptography (LASCA), in which infrastructure and work environment allowed to share knowledge and have important discussions. In special, to Armando Faz-Hernandéz to his numerous pointers and to Luan Cardoso dos Santos, of his in-depth knowledge of the target architecture, overall (graphical) design ideas and company in coffee-breaks.

To the faculty members, the staff and colleagues at the Institute of Computing at UNICAMP, which helped me to conclude this work in a way.

And finally, but not least, I would like to thanks Coordination of Superior Level Staff Improvement (CAPES), process number 1545323 and 1573303 for the early financial support and LG Electronics for the project's financial support which resulted in this work.

Abstract

With the advent of ubiquitous computing, the Internet of Things will undertake numerous devices connected to each other, while exchanging data often sensitive by nature. Breaching the secrecy of this data may cause irreparable damage. This raises concerns about the security of their communication and the devices themselves, which usually lack tamper resistance mechanisms or physical protection and even low to no security mesures. While developing efficient and secure cryptography as a mean to provide information security services is not a new problem, this new environment, with a wide attack surface, imposes new challenges to cryptographic engineering. A safe approach to solve this problem is reusing well-known and thoroughly analyzed blocks, such as the Transport Layer Security (TLS) protocol. In the last version of this standard, Elliptic Curve Cryptography options were expanded beyond government-backed parameters, such as the Curve25519 proposal and related cryptographic protocols. This work investigates efficient and secure implementations of Curve25519 to build a key exchange protocol on an ARM Cortex-M4 microcontroller, along the related signature scheme Ed25519 and a digital signature scheme proposal called qDSA. As result, performance-critical operations, such as a 256-bit multiplier, are greatly optimized; in this particular case, a 50% speedup is achieved, impacting the performance of higher-level protocols.

Resumo

Com o advento da computação ubíqua, o fenômeno da Internet das Coisas (de *Internet of Things*) fará que com inúmeros dispositivos conectem-se um com os outros, enquanto trocam dados muitas vezes sensíveis pela sua natureza. Danos irreparáveis podem ser causados caso o sigilo destes seja quebrado. Isso causa preocupações acerca da segurança da comunicação e dos próprios dispositivos, que geralmente têm carência de mecanismos de proteção contra interferências físicas e pouca ou nenhuma medida de segurança. Enquanto desenvolver criptografia segura e eficiente como um meio de prover segurança à informação não é inédito, esse novo ambiente, com uma grande superfície de ataque, tem imposto novos desafios para a engenharia criptográfica. Uma abordagem segura para resolver este problema é utilizar blocos bem conhecidos e profundamente analisados, tal como o protocolo Segurança da Camada de Transporte (de *Transport Layer Security*, TLS). Na última versão desse padrão, as opções para Criptografia de Curvas Elípticas (de *Elliptic Curve Cryptography* - ECC) são expandidas para além de parâmetros estabelecidos por governos, tal como a proposta *Curve25519* e protocolos criptográficos relacionados. Esse trabalho pesquisa implementações seguras e eficientes de *Curve25519* para construir um esquema de troca de chaves em um microcontrolador ARM Cortex-M4, além do esquema de assinatura digital *Ed25519* e a proposta de esquema de assinaturas digitais *qDSA*. Como resultado, operações de desempenho crítico, tal como o multiplicador de 256 bits, foram otimizadas; em particular, aceleração de 50% foi alcançada, impactando o desempenho de protocolos em alto nível.

List of Figures

Contents

Chapter 1

Introduction

The ubiquity of technology pervades every single area of knowledge, reaching into personal, professional, environmental, and industrial applications. Small-factor computing devices are deployed to identify an in-manufacturing item in a production line, to detect changes in the chemical properties of the soil where crops are cultivated, and to control life-supporting equipment, such as pacemakers and implantable defibrillators [1]. These devices are equipped with the ability to run private, safety-critical or legally liable activities, sensible data collection, manipulation, and transmission.

Along the designer's desire to have the product's dimensions as small as possible, embedded devices carry not-so-powerful computing capabilities in comparison to usual desktop computers, greatly limiting resources such as batteries and memory, having to cut out computationally expensive functionalities. Considering this environment, complex schemes providing data security are hardly implemented, despite the relevance of collected data. For example, sensor networks may collect personally identifiable information available in tags inside cars for surveillance purposes. As such, numerous attacks may be carried out such as physical attacks, since devices are often left unattended; eavesdropping and man-in-the-middle attacks, once communications capabilities are based on wireless protocols [2]. If machine-to-machine communications are used, breaking the chain of trust by faking identities is possible.

Passive attacks, like studying the time taken to compute an operation, is a relevant threat on small, unattended devices. Those attacks do not leave traces for further investigations [3]. More intrusive attacks also attempt to inject faults at precise execution times, in hope of corrupting execution state to reveal secret information [4].

The design and secure implementation of cryptographic algorithms is not a new area of study; however, on computationally weak devices with a wide attack surface, it still remains as a strong research area. Implementing countermeasures against side-channel attacks usually worsens the performance of cryptographic operations in comparison to their non-secured counterpart, thus obtaining a balance between the required security level and the usage of computational resources results in additional complexity to the overall engineering decisions. New algorithmic advances and novel implementation strategies are needed to alleviate this conundrum.

1.1 Motivation

A possible way to deploy security in new devices is to reuse well-known building blocks, such as the Transport Layer Security (TLS) protocol. In comparison with reinventing the wheel, using a new, under-analyzed option, this has a major advantage of avoiding risky security decisions that may repeat issues already solved in TLS. In the handshaking phase, asymmetric (or public key) cryptographic schemes are largely used; namely, key exchanges and digital signatures.

Both schemes are examples of where parties do not need to agree on a secret beforehand. A key exchange protocol establishes a shared secret over an insecure channel; this is the case when a web-browser connects to a TLS-secured server, where no prior opportunity to exchange secrets was available. Further care, like authenticating both parties before handling secrets and using an established long-term key to negotiate new ephemeral keys must be taken. In the first case, a malicious third party may intercept the protocol and exchange keys with both, becoming a relay. On the second, the channel is secured, never exposing the new keys in the plain. For authentication purposes, a digital certificate issued to the web server signed by a trusted third party guarantees that there are no middlemen in the handshake.

In the Request for Comments (RFCs) 7748 and 8032, published by the Internet Engineering Task Force (IETF), two asymmetric cryptographic protocols based on the Curve25519 elliptic curve and its Edwards form are recommended and slated for use in the TLS suite: the elliptic curve Diffie-Hellman key exchange using Curve25519 [5] called X25519 [6] and the Ed25519 digital signature scheme [7]. These schemes rely on a careful choice of parameters, favoring secure and efficient implementations of finite field and elliptic curve arithmetic with a smaller room for mistakes due to their overall implementation simplicity. There are Curve25519 optimized implementations for several platforms (x86_64 [8], AVR [9], ARM NEON [10]) and the protocols based on the curve are used in numerous softwares, including The Tor Project, OpenSSH and the Google Chrome web browser.

1.2 Objectives

This work sets out as the main objective to provide an efficient implementation of the Curve25519-based cryptographic protocols, resistant to side-channel attacks; in particular, timing and cache attacks. More specifically, this requires efficient underlying arithmetic modulo $2^{255} - 19$ with no conditional branches to avoid timing issues, leading to handcrafted implementations in Assembly code at this level. At the protocol level, elliptic curve group arithmetic must also be optimized, in particular, scalar multiplication.

1.3 Contributions

Contributions of this work consist of how to securely and efficiently implement arithmetic operations needed for the X25519, Ed25519 and qDSA protocols running on an entry-to-medium level ARM Cortex-M4-based microcontroller, selected as the main target platform. Those implementations were also evaluated in higher-end CPU cores, such as the ARM Cortex-A7, and the more recent ARM Cortex-A53.

In summary:

- An efficient and secure implementation of the 256-bit arithmetic needed for the Curve25519 operations. In particular, an efficient implementation of a 256-bit multiplier using Digital Signal Processing instructions of the Cortex-M4, which in turn speeds up the key exchange and the digital signature schemes using the mentioned curve (or its birrationally equivalent form).
- An alternative way to implement the conditional swap operation on ARMv7 and higher platforms, leading to a slightly faster version of the operation in comparison to implementations based on the classical algorithm [11].
- An optimized implementation of the Ed25519 and qDSA digital signature schemes geared towards the Cortex-M4 embedded microcontroller, having in mind the limited space in ROM to speed up the fixed-point multiplication on the curve.

Results include a 50% speedup in field multiplication, in comparison to the previous state-of-art work. This operation is performance-critical in both key-exchange, which runs in 0.9×10^6 cycles (41% speedup versus the previous state-of-art) using this optimization; and in digital signature schemes, taking approximately 0.5×10^3 cycles to sign a 5-byte message in the EdDSA scheme. This last implementation, as far as found in available literature, is the first one geared towards the ARM Cortex-M4. For the qDSA scheme, a 33% speedup is shown using the precomputed ladder algorithm.

Partial results of this work were published in the Fifth International Conference on Cryptology and Information Security in Latin America (Latincrypt) 2017 [12] and in the Seventh International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE) 2017 [13].

1.4 Structure

This work is structured as follows: Chapter 2 presents the theoretical background used in this work, discussing the target platform and its security concerns, and showing the mathematical concepts behind the cryptographic protocols. Chapter 3 discusses the implementation of the arithmetic investigated in this work, with a brief explanation of the state-of-the-art implementation techniques and more efficient alternatives. Chapter 4 presents the testing platforms and the performance numbers, measuring the methods shown in the previous chapter and discussing the implementation of the higher-level protocol, along with a brief comparison of related works. In the end, Chapter 5 concludes by briefly discussing this work in its entirety.

Chapter 2

Theoretical Background

Asymmetric cryptography relies on the mathematical aspects of problems which are computationally hard to solve, resulting in properties reflecting into the security level provided by protocols. Many attacks aim to lower the complexity of solving the underlying problem, requiring large parameters to make those unfeasible, and thus using more computational resources. The usage of elliptic curve protocols aims to reduce the size of the parameters, using the fact that not all attacks can be used in elliptic curve scenarios.

This chapter presents the underlying mathematical theory, building up to elliptic curve cryptographic protocols. Features of the target environment are also shown here, such as aspects of applicability and security concerns raised by them.

2.1 Asymmetric Cryptography

Symmetric (or secret-key) and asymmetric (also known as public-key) cryptography are the main two types of cryptography and both are used to build secure communication systems. Focusing on the last type, the concepts of a *private* and *public keys* are introduced – the first secluded as a secret and the second publicly available. Note that, given the private key, it's easy to compute the public information. The point of building one-way trapdoor functions, easy to compute in one direction but practically impossible to invert without additional information, is the main idea behind public-key cryptography [14].

Public key cryptography has two principal goals [15]:

- Key Agreement allows two parties to share a secret key for use in a symmetric cipher using an insecure communication channel, where an attacker may be retrieving exchange information between the parties.
- Digital Signatures allow an entity to generate a signature, given a message and a private key of the entity. Anyone can check the validity and the source of the communication, given the message and the public key related to the private one while no other party can forge the signature with non-negligible probability.

Key exchange protocols and digital signature schemes are building blocks for applications like key distribution schemes and secure software updates based on code

signing. These protocols are fundamental for preserving the integrity of software running in embedded devices and establishing symmetric cryptographic keys for data encryption and secure communication.

2.1.1 Practical Applications

The initial phases of the Transport Layer Security (TLS) protocol (and its early versions, named Secure Socket Layers (SSL)) are a prime example of how widespread is the application of public-key cryptography. Using the TLS protocol, every type of network connection may encrypt its messages in transit. This allows a web browser, for example, to securely receive hypertext hosted in an HTTP server.

Public key cryptography primitives are largely used in the *handshaking* part of the communication, i. e. the negotiation phase of the parameters to start a secure channel. Key exchange and signatures allow Alice and Bob to establish a secure communication channel over an insecure one:

1. Alice generates a pair of private and public keys and publishes the latter. A digital certificate, issued by a third party trusted by all parties in the communication channel, containing a digital signature confirming that the key belongs to Alice is also published.

2. Bob signals to Alice that he wants to start a secure communication channel.

3. Alice requests Bob to start a shared key exchange and also sends her share of the to-be-negotiated key.

4. Bob gets Alice's public key and, using the digital certificate, ensures that the public key was really generated by Alice. If the check passes, Bob generates his share of the shared key, encrypts it using Alice's public key and sends it to her. Bob also finishes his part of the key exchange and requests a change of the cipher protocol being used to a symmetric one using the shared key.

5. Alice deciphers Bob's message, and with both shares, the key exchange can be finished. Alice acknowledges the protocol change and both can start a secure communication using a symmetric cipher under the key they agreed.

The TLS 1.2 handshaking protocol (Figure 2.1) basically encompasses these steps, with additional negotiation in regards to the supported cipher suits in both sides of the communication channel. Albeit not common in practice, the server may request an additional authenticity check of the client. Further versions of this protocol introduce features to minimize communication required by the handshaking phase (and thus lowering the initial overhead of each connection) and limit the options of supported cipher suits in comparison to TLS 1.2. In special, elliptic curve cryptography is strongly recommended to execute this initial agreement phases.

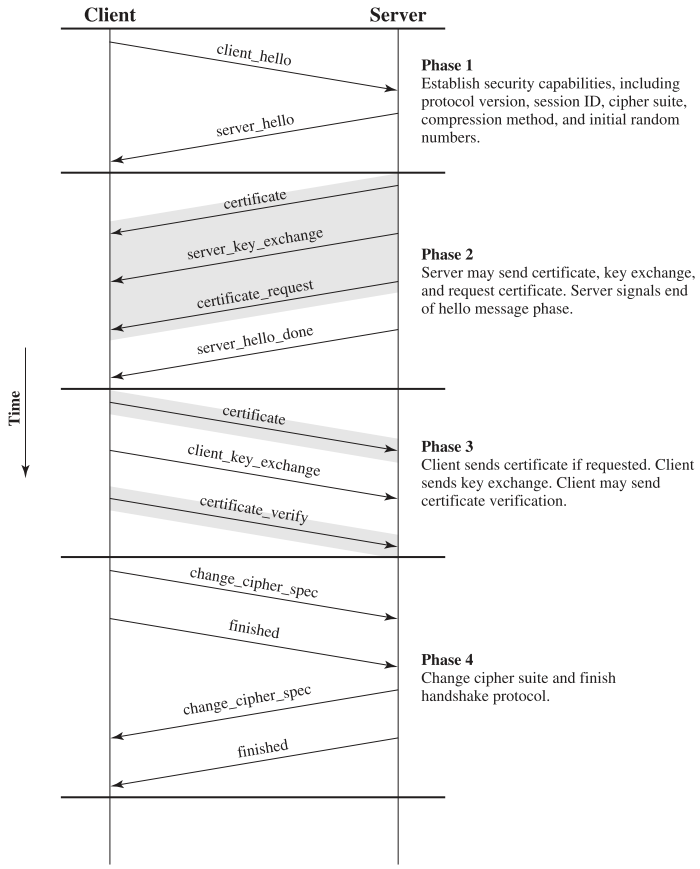


Figure 2.1: TLS 1.2 handshaking process (from [16]).

2.2 Embedded Devices and Data Security

With the miniaturization of the computing devices, most of the tasks formerly handled by traditional desktop computers with their (semi-)annual improvements were migrated to smaller, ubiquitous devices. Further advances to this kind of devices allowed them to be equipped with sensors and more powerful communication abilities, such as wireless technologies. Many migrated applications rely on security mechanisms, from protecting the vulnerable communication channel to enabling copy protection schemes on multimedia files. In mobile devices, as an example, where processing capabilities, memory space, and battery longevity are constrained, it is evident that some implementations may be not sufficiently efficient to the point that its performance matches the bandwidth of the communication link. In this case, inconvenience may arise, since delays are unacceptable from the customer point of view [17].

In addition to the usual tasks, embedded devices have been shown to be good solutions for problems present in new domains. Examples of new applications tailored to ubiquitous devices include mobile ticketing, monitoring environmental parameters for the preservation of perishable food, sensors to learn temperature preferences of homeowners and warn about domestic incidents, sensor devices in use for healthcare collecting real-time information of the patients for telemedicine solutions. Security requirements on those sensible domains is a relevant issue in designing embedded devices. Since increased levels of security might be required due to legal constraints or

be used as countermeasures against potential data breach threats during the lifespan of a product, implementations targeting data security might be sufficiently costly in terms of computational resources.

An alternative is to increment the set of instructions with customizations in a way that a given processor may run the needed cryptographic functionality in an optimized fashion. This approach follows the example set by the industry in regards to media processing and digital signal processing, where new instructions added to commercial CPUs accelerated related functionalities. In the literature, the approach is also shown to be possible in cryptography, given the multiple instruction proposals and their implementations [18, 19], such as the AES-NI extension set present in some Intel architectures [20] and the SHA-NI extension.

Efficient but non-specialized instructions may be also repurposed for cryptography use. For example, digital signal processors (DSPs) are equipped with instructions targeting video and audio processing; in particular, instructions to compute *multiply and accumulate* (MAC) operations may be implemented. Those, for example, may be used to speed up operations in elliptic curve cryptography [21].

2.3 Side-channel Attacks

Even with the availability of instructions to the purpose of speeding up cryptographic operations, insecure implementations are still vulnerable to attack which may breach data confidentiality. Attacks range from the logical side, using crafted inputs to exploit weaknesses of the implemented algorithm or bugs in the code, to physical readings and observations of the underlying hardware executing the code [22].

Attacks can be separated on two major classifications:

Invasive vs. Non-Invasive Some attacks may need to physically violate the device, ranging from removing the casing to delidding and decapping integrated circuits, obtaining direct access to components. Non-invasive options just observe available phenomena externally, such as execution times or energy consumption.

Active vs. Passive Attacks may try to change the device's functionality by, for example, introducing failures such as voltage spikes during execution. Passive attacks just collect data of the devices, without modifying its state of execution.

Unprotected implementations leak physical phenomena, measurable by an external entity [23]; such properties may characterize an execution pattern. For example, when energy consumption is higher than a baseline, it can be concluded that a high task load is being executed; analogically, when consumption is lower, it can be inferred that a low number of tasks are being done. Such analysis can be improved to make an attacking algorithm possible. Using as inputs the execution characteristics and the physical phenomena emanated by the hardware during the running of a program, the algorithm may output the data under process or reveal which type of operation was being conducted inside the execution units. These kinds of analysis, called *side-channel*

attacks, may violate data security principles, such as revealing secret data or the keys used for hiding it. Those kinds of attacks are more powerful in comparison to classical cryptoanalysis in the sense the first does not leave traces of invasion, since observations may be conducted in passive ways without leaving traces for further identification, albeit both attacks having similar objectives [23, 24]. Along with that, side-channel attacks can be done with low resources, becoming a potential threat for security-minded devices such as smart cards and RFID tags.

Countermeasures against side-channel attacks include hardening physical protections [25], hardware-tampering sensing capabilities, data leak mitigations (if impossible to avoid) [26] and the usage of constant-time (isochronous) implementation of algorithms [24].

Protected implementations may present lower performance in comparison to its non-protected version, since optimizations may be susceptible to attacks. Implementation alternatives must be searched on to find an equilibrium point between performance and the required level of security [27].

Various kinds of side-channel attacks can be enumerated:

Timing Attacks If the execution time depends on the bits of a secret, an attacker may measure those differences and reveal the secret. In a related attack, timing differences induced by the usage of cache memory may be detected if this optimization is used. An example can be found in AES implementations, where tables stored in cache memory may speed up operations, albeit timing differences related to caching potentially reveal the used key [22].

Power Attacks Energy consumption may vary during the execution of an algorithm in addition to its inputs. As an example, clearing or using the all-bits-set configuration in a register requires low and high amounts of energy, respectively. Direct measures of the usage may reveal secrets [11]. In related phenomena, power usage may induce disturbances in the electromagnetic spectrum or even make audible noise. Analyzing this noise is also known to be an attack vector [26].

Data Remanence Storage systems may retain its data even if the user explicitly erased the data from the filesystem due to its physical characteristics. Forensic analysis techniques may be able to recover secret data from physically analyzing the component [28].

Fault Injection Modifying clock signals with unsupported settings, variating the voltage of the power supply lines in order to glitch the device, using heat to introduce unwanted bit modifications are all examples of how to make an implementation to be erroneously executed on the system under attack. Faults may leak a single bit from a secret or entire substitution boxes to leak encrypted data [29].

2.4 ARM-based Processors

Unlike desktop and larger computers, embedded devices typically feature an ARM-based processing or micro-controller unit. The ARM family of CPUs consists of a reduced instruction set computing (RISC) architecture design available to licensees, which in turn combines other parts to produce complete devices, ranging from micro-controllers to entire system-on-chip (SoC) products. Albeit featuring variable characteristics across parts and manufacturers, such as auxiliary hardware to accelerate specific features like multimedia processing, the basic CPU core handles a mandatory set of instructions defined by the architecture.

The ARM architecture is a reduced instruction set computer using a load-store architecture. ARM processors are equipped with 16 registers: 13 general purpose, one for the program counter (`pc`), one for the stack pointer (`sp`), and the last one for the link register (`lr`). The latter can be freed up by saving it on slower memory and retrieving it after the register has been used. This family of cores is equipped with a barrel shifter allowing some bitwise operations, such as shifts and rotations, to be executed without taking an extra CPU cycle.

Memory access involving n registers in these processors takes $n+1$ cycles if there are no dependencies (for example, when a loaded register has the address for a consecutive store). This can happen either in a sequence of loads and stores or during the execution of instructions involving multiple registers simultaneously. Batch memory operations can be optimized by the pipeline, which has a different number of stages in different core families.

The focus is given on the Cortex-M and -A families, which support the ARMv7 architecture. The Cortex-M family is a low-cost design tailored to real-time embedded applications, such as computer peripherals, wireless sensors, and devices requiring basic computing abilities. The hardware does not have a Memory Management Unit (MMU), making it unable to run full-fledged operating systems. The number of pipeline stages in this family is limited to three, the bare minimum to optimize memory operations in batches.

The second family encompasses processors suitable to general consumer-grade devices, such as smartphones, TV set-top receivers and single-board computers. The ARM Cortex-A cores are computationally more powerful than their Cortex-M counterparts; and can run robust operating systems due to extra auxiliary hardware. Typically, processors based on this design are equipped with a Single Instruction-Multiple Data (SIMD) unit called NEON and have support for floating point arithmetic. Aside from that, those processors may have sophisticated out-of-order execution and extra pipeline stages in comparison to the Cortex-M core family. Processors using this design can heterogeneously combine different cores in order to properly scale computing needs, for example, combine two powerful Cortex-A53 and two power-saving Cortex-A7 cores.

The ARMv7E-M instruction set, present on the Cortex-M4 and superior cores, comprises of standard instructions for basic arithmetic (such as addition and addition with carry) and logic operations, but differently from other lower processors classes, the Cortex-M4 has support for the so-called DSP instructions, which include *multiply-and-*

accumulate (MAC) instructions:

- *Unsigned MULtiply Long:* `UMULL rLO, rHI, a, b` takes two unsigned integer words a and b and multiplies them; the upper half result is written back to `rHI` and the lower half is written into `rLO`.
- *Unsigned MULtiply Accumulate Long:* `UMLAL rLO, rHI, a, b` takes unsigned integer words a and b and multiplies them; the product is added and written back to a double word integer stored as (`rHI, rLO`).
- *Unsigned Multiply Accumulate Accumulate Long:* `UMAAL rLO, rHI, a, b` takes unsigned integer words a and b and multiplies them; the product is added with the word-sized integer stored in `rLO` then added again with the word-sized integer `rHI`. This double-word integer is then written back into `rLO` and `rHI`, respectively the lower half and the upper half of the result.

ARM's Technical Reference Manual of the Cortex-M4 core [30] states that all the mentioned MAC instructions take one CPU cycle for execution in the Cortex-M4 and above. However, those instructions deterministically take an extra three cycles to write the lower half of the double-word result, and a final extra cycle to write the upper half. Therefore, proper instruction scheduling is necessary in order to avoid pipeline stalls and to make best use of the delay slots.

2.5 Mathematical Preliminaries

2.5.1 Finite Fields

Many (asymmetric) cryptography primitives use underlying Number theory problems which, from the standpoint of algorithm complexity, are hard to solve, i. e. do not have known solutions running in polynomial time. Those problems usually rely on finite field mathematics; hence its importance to cryptography.

Before defining a field, the concept of a group must be presented first. A group consists of a set G with a binary operation $* : G \times G \to G$ with the following properties:

1. (Closure) For all $a, b \in G$, the result $a * b$ is also in G
2. (Associativity) For all $a, b, c \in G$, $a * (b * c) = (a * b) * c$
3. (Identity) There's an *identity* element $e \in G$ such as $a * e = e * a = a$ for all $a \in G$
4. (Inverse) For each $a \in G$, there's an *inverse* element $b \in G$ which satisfies $a * b = b * a = e$

If $a * b = b * a$ for every $a, b \in G$, the group is also said to be *abelian* or commutative.

The $*$ operation is usually the addition (+) or the multiplication ($\cdot$); in the first, the group is called to be an *additive* one, using *0* as the identity element and the *additive* inverse of a noted as $-a$. On the second, the group is *multiplicative*, employing *1* as the

identity element and the *multiplicative* inverse of a noted as a^{-1}. If the set of numbers is finite, then the group is said to be a finite group; the number of elements in the group is called the *order*.

If G is a finite multiplicative group containing n elements and $g \in G$, the smallest positive integer t such as $g^t = 1$ is the *order of the element* g. If G has an element g of order n, then G is a *cyclic group* and g is its *generator*. Note that additive groups hold the same concepts analogically.

A field consists of the following elements and properties:

- $(\mathbb{F}, +)$ is an abelian group with the additive identity denoted by 0
- $(\mathbb{F} \setminus 0, \cdot)$ is an abelian group with the multiplicative identity denoted by 1
- (Distributivity) $(a + b) \cdot c = a \cdot c + b \cdot c$, for all $a, b, c \in \mathbb{F}$

As an example, let p be a prime number; $\mathbb{F}_p = \{0, 1, 2, \ldots, p-1\}$ denotes the set of integers modulo p. $(\mathbb{F}_p, +)$ is an additive finite group of order p; likewise, $(\mathbb{F}_p^*, \cdot)$, where $\mathbb{F}_p^*$ denotes nonzero elements in $\mathbb{F}_p$, is a multiplicative group with $p - 1$ elements with multiplicative identity element 1. Joining those in the triple $(\mathbb{F}_p, +, \cdot)$, we form the *finite field* modulo p, denoted as $\mathbb{F}_p$.

2.5.2 The Discrete Logarithm Problem

Let G be a finite cyclic group of order n, denoting α as its generator, and let $\beta \in G$. A *discrete logarithm of* $\beta \in G$ *to the base* α, or $\log_\alpha \beta$, is the unique integer $x, 0 \leq x \leq n - 1$, such as $\beta = \alpha^x$. The *generalized discrete logarithm problem* is defined as finding such x, given G of order n, its generator α and an element $\beta \in G$.

This problem can be solved using exhaustive search, computing $\alpha^0, \alpha^1, \alpha^2, \cdots$ until β is found. This algorithm takes $O(n)$ operations, where n is the order of α. In cases of cryptographic interests, n is large so the search operation is inefficient.

If the group order q of G is not prime and its factorization is known, or easy to determine, the Pohlig-Hellman algorithm reduces the problem of finding discrete logarithms in G to that of finding discrete logarithms in prime-order subgroups of G [31]. This lowers the complexity of finding the discrete logarithm of G to finding the discrete logarithm of subgroups of G with order q', where q' is the largest prime dividing q. For this reason, prime-order groups are preferred for cryptographic uses.

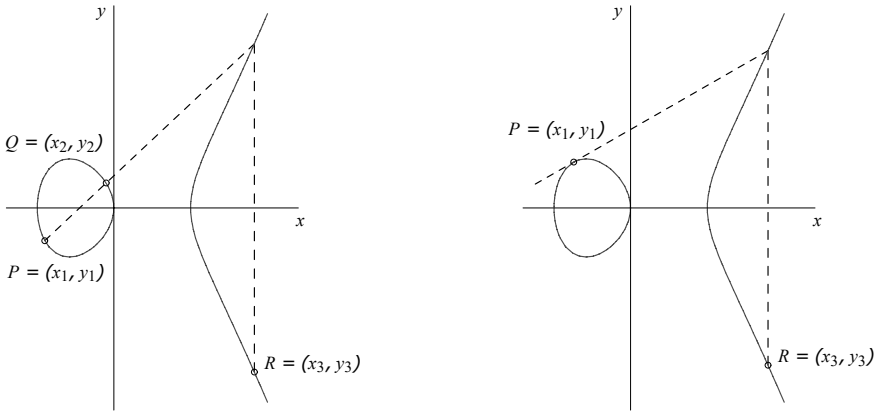
Sub-exponential algorithms to solve the DLP are known, hence the need for large parameters of protocols relying on finite fields. In special, index calculus uses a three-step process: first, a factor base is selected, consisting of a set of small prime numbers. Then, linearly independent relations with respect to the generator g must be found. For the second part, the linear system of equations must be solved to find the logarithms in the factor base. For the last part, the field element which logarithm has to be computed is multiplied by a random power of g, then factored in terms of the base. In addition, a quasi-polynomial time algorithm to solve the discrete logarithm in fields $\mathbb{F}_q^{kn}$, with $k \geq 2$ fixed and $n \leq q + d$ with small d is known [32].

2.5.3 Elliptic Curves

An elliptic curve E over a finite field $\mathbb{F}_q$ with $q \neq 2, 3$ is the set of solutions $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ which satisfy

$$E/\mathbb{F}_q : y^2 = x^3 + ax + b, a, b \in \mathbb{F}_q \tag{2.1}$$

Curves described in this format are called *(short) Weierstrass equations*. The set of points $E(\mathbb{F}_q) = \{(x, y) \in E(\mathbb{F}_q)\} \cup \{\mathcal{O}\} = \{P \in E(\mathbb{F}_q)\} \cup \{\mathcal{O}\}$ with an additive operation $\oplus$ form an additive group, with $\mathcal{O}$ as the identity element. Duplication, a special case of the addition operation, is also supported. Figure 2.2 illustrates those operations.



(a) Adding points $P + Q = R$. (b) Duplicating point $2P = R$.

Figure 2.2: Adding points and duplicating a point on an elliptic curve (from [33]).

Given an elliptic curve point $P \in E(\mathbb{F}_q)$ and a k in $\mathbb{Z}$, the operation kP, called scalar point multiplication, can be defined by

$$[k] : P \mapsto \underbrace{P \oplus \ldots \oplus P}_{k\ times} \tag{2.2}$$

This operation encodes the security assumption for Elliptic Curve Cryptography (ECC) protocols, basing their security on the hardness of solving the elliptic curve analogue of the discrete logarithm problem. The elliptic curve discrete logarithm problem (ECDLP) is defined as, given an elliptic curve E defined over a finite field $\mathbb{F}_q$, a point $P \in E(\mathbb{F}_q)$ of order n and a point $Q \in \langle P \rangle$, to find the integer $l \in [0, n-1]$ such as $Q = lP$. The integer l is called the *discrete logarithm of Q to the base P* [33].

Elliptic curve cryptosystems were independently proposed for cryptographic purposes in [34] and [35]. The former work points out that a Diffie-Hellman key-exchange can be to expressed as $x(P) \mapsto x([k]P)$, given a public base point P [36]. That means that Alice can compute $x([a]P)$ and publish it; the same applies to Bob but using different a secret b instead of a. Once both receives the computation results, a shared secret can be evaluated, since $x([b][a]P) = x([a][b]P)$.

The most trivial algorithm to solve the ECDLP is exhaustive search: compute $2P$, $3P$, $4P$, ... until Q is found, which may take n steps to complete. To circumvent this search, elliptic curve parameters can selected in a manner that n is big enough that a

large amount of computation has to be done (such as $n \geq 2^{80}$). A general-purpose attack known to solve this problem is a combination of the Pohling-Hellman algorithm and Pollard's ρ algorithm, which has a computational complexity of $O(\sqrt{p})$, where p is the largest prime divisor of n. To resist this attack, parameters must be set to make $\sqrt{p}$ large enough to make computation infeasible.

No sub-exponential algorithms are known for computing discrete logarithms in certain elliptic-curve groups. For a given security level, elliptic curve groups with small orders (with carefully chosen parameters) can be used in comparison with subgroups in $\mathbb{Z}_p^*$ containing a large number of elements. In practice, for any required security level, elliptic curve systems can use smaller parameters, with (asymptotically) faster group operations compared to finite field groups.

2.5.4 Other Elliptic Curve Models

A *Montgomery curve* over $\mathbb{F}_p$ is an elliptic curve defined by an affine equation

$$E/\mathbb{F}_p : By^2 = x(x^2 + Ax + 1). \tag{2.3}$$

Parameters A and B are in $\mathbb{F}_p$ satisfying $B \neq 0$ and $A^2 \neq 4$. This curve model is ideal for curve-based key exchanges, because it allows the scalar multiplication to be computed using x-coordinates only, albeit formulas for these kind of curve are not complete, i. e. they fail for certain inputs.

An *Edwards curve* over $\mathbb{F}_p$ is an elliptic curve defined by an affine equation

$$E/\mathbb{F}_p : ax^2 + y^2 = 1 + dx^2y^2, d \in \mathbb{F}_p \backslash \{0, 1\}. \tag{2.4}$$

Edwards curves benefits of complete addition formula, and, in comparsion to Weierstrass curves, are considered to have faster group operation.

2.6 Cryptographic Protocols

The points of an elliptic curve, in combination with an addition law, forms an additive group. The elliptic curve discrete logarithm problem is present in this construction; given a proper set of parameters of the curve, it's computationally unfeasible to solve it. This section introduces Curve25519 and cryptographic protocols based on this curve.

2.6.1 The X25519 Key Agreement Scheme

In order to reduce implementation difficulties and security risks involved with complex designs and set new speed records, Bernstein introduced Curve25519, a cryptossytem with 128-bit security level using the Montgomery model of an elliptic curve. The curve is defined by the equation

$$\text{Curve25519:} \quad y^2 = x^3 + 486662x^2 + x. \tag{2.5}$$

This curve serves as the mathematical background to build an elliptic curve Diffie-Hellman key exchange protocol based on a function called X25519, which is an scalar multiplication based on the Montgomery ladder.

A user selects a 32-byte long secret by taking the output of a cryptographically secure random number generator. Aside from sufficient randomness, the key must have set its topmost bit and the three lowest bits are set to 0, while its second-to-top bit is set to 1. With the 32-byte secret properly formatted, the user applies the X25519 scalar multiplication, using as inputs the base point 9 and the secret scalar, resulting in a 32-byte long public key. When two users exchange their respective public keys and multiply their received public keys by their secret scalar, they establish a shared value which can be used as a symmetric shared secret, given a suitable key-derivation function.

The requirement of transforming the secret key into a multiple of 8 serves as a countermeasure against small subgroup attacks, where an attacker can exchange a public key consisting of a point with a small order on the curve (or in its quadratic twisted form) in order to reveal information about the secret key. If those lower 3 bits are not cleared, the scalar multiplication will be performed modulo a small order b chosen by the attacker, remaining a small subset of possibilities for an attacker to brute-force [37]. On Curve25519, this attack fails since the order of the base point 9 is prime, so no small b exists. For the curve's twisted form, the smallest orders under 2^{252} are 1, 2, 4 and 8; since the secret is multiplied by 8, $s \mod b = 0$ for all secret s and orders b. The set second-most bit provides a fixed leading one, preventing optimizations trying to find a starting point of the ladder and avoiding branching (and possible timing attacks) due to the incompleteness of operations on the Montgomery format.

The X25519 Montgomery ladder takes any 32-byte x-coordinate of point P, ignores the value of the most significant bit and multiplies by an also 32-byte scalar s. The resulting value is the point $Q(x, z) = sP$ in the form $x \cdot z$. To get the final value, inversion is efficiently done by computing the $p - 2 = (2^{255} - 21)$-th power of z (as a consequence of Fermat's Little Theorem) using a chain of 254 squares and 11 multiplications, in an instance of the Itoh-Tsujii algorithm [38]. This value is then multiplied by x, yielding a 32-byte value, either a public key or a shared secret, depending on the P input being either the base point 9 or a public key respectively. Resulting public keys do not need verifications since all inputs of P may be accepted.

The scalar multiplication runs in a constant-time fashion using 255 "ladder steps" (one for each bit of the secret), each one performing a doubling and a differential addition, followed by two conditional swaps on the pair of coordinates (x_1, z_1) and (x_2, z_2). Algorithm 1 shows the steps of this method; in the listing, the conditional swap function `cswap` returns, in constant-time, (b, a) if the bit of the secret is zero; otherwise, it returns (a, b). Since no branching (secret key being scanned bit by bit to decide whenever the conditional swap must be done) neither array indexing access based on secret data is done, this algorithm is deemed safe against timing attacks or cache-based attacks.

Algorithm 1 X25519: scalar multiplication using the Montgomery ladder and inversion to obtain the x-coordinate of sP. Value s_t denotes the t^{th} bit of a 255-bit secret scalar s in its little-endian form; $p = 2^{255} - 19$ and $a24 = (A + 2)/4 = (486662 - 2)/4 = 121665$.

Input:
Scalar s, point P expressed as x-coordinate
Output:
sP, expressed as x-coordinate

$x_1 \leftarrow P, x_2 \leftarrow 0, z_2 \leftarrow 1, z_3 \leftarrow 1$
for $t = 254$ down to 0 **do**
  $(x_2, x_3) \leftarrow cswap(s_t, x_2, x_3)$ ▷ Algorithm 2.
  $(z_2, z_3) \leftarrow cswap(s_t, z_2, z_3)$
  $A \leftarrow x_2 + z_2$
  $AA \leftarrow A^2$
  $B \leftarrow x_2 - z_2$
  $BB \leftarrow B^2$
  $E \leftarrow AA - BB$
  $C \leftarrow x_3 + z_3$
  $D \leftarrow x_3 - z_3$
  $DA \leftarrow D \cdot A$
  $CB \leftarrow C \cdot B$
  $x_3 \leftarrow (DA + CB)^2$
  $z_3 \leftarrow x_1 \cdot (DA - CB)^2$
  $x_2 \leftarrow AA * BB$
  $z_2 \leftarrow E \cdot (AA + a24 \cdot E)$
  $(x_2, x_3) \leftarrow cswap(s_t, x_2, x_3)$
  $(z_2, z_3) \leftarrow cswap(s_t, z_2, z_3)$
end for
return $x_2/z_2 = x_2 \cdot z_2^{p-2} \mod p$

Algorithm 2 `cswap`: Conditional swap using bitwise operations.

Input:
Bit b and Points A and B.
Output:
(B, A) if $b = 1$; (A, B) otherwise.

$mask \leftarrow \neg(b - 1)$
$dummy \leftarrow mask$ AND $(A \oplus B)$
$A \leftarrow A \oplus dummy$
$B \leftarrow B \oplus dummy$
return (A, B)

2.6.2 The Ed25519 Digital Signature Scheme

The Edwards-curve Digital Signature Algorithm [7] (EdDSA) is a signature scheme variant of Schnorr signatures, instantiated with elliptic curves represented in (possibly twisted) Edwards model. One of the recommended parameters, as described in the RFC 8032 document, is the edwards25519 curve, which can be described by the equation

$$\text{edwards25519:} \quad -x^2 + y^2 = 1 - \frac{121655}{121666}x^2y^2. \tag{2.6}$$

When instantiated using edwards25519 (Equation 2.6), the EdDSA scheme is named Ed25519. Equation 2.6 is a birrationally equivalent curve of Curve25519; implying that the ECDLP difficulty for this signature scheme instantiated with edwards25519 is the same as solving ECDLP for Curve25519. In addition, the Edwards curves model benefits from complete addition group laws, aiding secure implementations of this protocol.

The security level target remains at 128-bit, as in the ECDH protocol based in Curve25519. This system uses 32-byte keys for both public and private keys. Signatures fit in 64 bytes, being significantly smaller than non-elliptic curve digital signature schemes, such as based on the RSA algorithm [15].

An Ed25519 secret key is a 256-bit string k. Let H be a hash function mapping arbitrary-length strings to 512-bit hash values. The public key is the scalar product $A = aB$, being a be the lower 32 bytes of the $H(k)$ with the 3 least significant and the most significant bit cleared, plus the second-most signficant bit set.

Let r be the hash of the concatenation of $H(k)$ and the message M. The signature of M and private key k under the Ed25519 scheme is a 512-bit string (R, S). Let r be the hash of the concatenation of $H(k)$ and the message M; then R is the scalar product of the base point B and the r hash interpreted as a little-endian integer. Value S is the integer of little-endian interpretation of $r + H(R, A, M)$ modulo the group order ℓ [7].

Verification works by parsing the signature components (R, S) and the public key A. The verifier computes $H(R, A, M)$ and checks if the group equation $S \equiv r + H(R, A = aB, M) \pmod{\ell}$ holds. If parsing fails and the equation doesn't hold, the check fails.

Like other discrete-log based signature schemes, EdDSA requires a secret value, or nonce, unique to each signature. For reducing the risk of a random number generator failure, EdDSA calculates this nonce deterministically, as the hash of the message and the private key. Thus, the nonce is very unlikely to be repeated for different signed messages. While this reduces the attack surface in terms of random number generation and improves nonce misuse resistance during the signing process, high quality random numbers are still needed for key generation.

The most expensive computations on key generation and message signing is the fixed-point scalar multiplication. In order to optimize these operations time-wise, lookup tables containing multipliers of the chosen point may be used in exchange of read-only memory usage. This can speed up the classic right-to-left binary method to evaluate a point multiplication by eliminating doublings, given a table with the precomputed points $2P, 2^2P, \ldots, 2^{t-1}P$, where t is the bit length of the scalar.

2.6.3 The qDSA Digital Signature Scheme

Although Ed25519 and X25519 can be used in conjunction and share the same underlying field arithmetic, keys generated under both schemes are not compatible. Alternatives to make such compability possible, such as XEdDSA [39], derived from previous proposals. A novel scheme, the Quotient Digital Signature Algorithm (qDSA), building on Montgomery curves, was proposed by Renes and Smith [40] and it is named after the scalar point multiplications are done in a algebraic variety generated by a quotient of an algebraic curve.

This signature scheme (if instantiated using Curve25519) allow that X25519 keys be used, without modifications, to sign data. From the fact that the underlying curve is of the Montgomery form, curve operations are done only on the x-coordinate of the points. As a drawback, given a qDSA signature, a second signature also passing verification can be computed. This opens possibilites to misuse, potentially becoming an effective attack.

qDSA is also based on the Schnorr signature scheme as in Ed25519, operating over a Kummer variety $\mathcal{K}$. This surface is the quotient of an (hyper-)elliptic curve E computed by $\mathcal{K} = E/\langle \pm 1 \rangle$, hence, in case of elliptic curves, points $P, -P \in E$ are mapped into a single element in $\mathcal{K}$. Group structure is not preserved, but scalar multiplications are still possible. If E is an elliptic curve, $\mathcal{K}$ turns out to be an one-dimensional space known as the x-line [40].

Let H be the a hash function mapping arbitrary-length strings to 512-bit hash values. A qDSA secret key is a 512-bit string $(d_0||d_1)$, computed by taking the digest from a random, 256-bit value. The public key is a 256-bit string x_Q computed by a scalar multiplication of a base point G and d_0 and compressing this result by taking the x-coordinate of d_0G divided by the z-coordinate of the same scalar product (Algorithm 3).

Let r be the digest value of d_1 concatenated with a message M of arbitrary length modulo ℓ. Let $R = (X_r, Z_r)$ be the scalar product of r and the base point G; x_R is the compressed form of rG using the same steps to compress a public key. Let h be the hash of the concatenation of x_R, the public key x_q and the messsage M. The signature of a message M using the signer's keys (d_0, d_1) and its public key x_q is the concatenation of x_R and the string s; this last one computed by subtracing the product hd_0 from r modulo ℓ (Algorithm 4).

Given an alleged signature $(x_R \parallel s)$ of a message M, the qDSA signature verification procedure (Algorithm 5) must determine whether x_R is the x-coordinate of $R_0 + R_1$, where $R_0 = sG$ and $R_1 = hQ$ for h defined as $h \equiv H(x_R \parallel x_Q \parallel M) \pmod{\ell}$. For that purpose, Algorithm 6 checks if $f(x_R) = 0$, where f is the quadratic polynomial $f(x) = f_2x^2 + f_1x + f_0$, such that:

$$\begin{aligned} f_2 &= (x_{R_0} - x_{R_1})^2 \,, \\ f_1 &= -2(x_{R_0}\, x_{R_1} + 1)(x_{R_0} + x_{R_1}) - 4A\, x_{R_0}\, x_{R_1} \,, \\ f_0 &= (x_{R_0}\, x_{R_1} - 1)^2 \,. \end{aligned} \tag{2.7}$$

Algorithm 3 qDSA: Key generation

Input:

$\mathcal{D}$, the domain parameters.

Output:

$(d_0, d_1) \in \{0,1\}^{2N}$ is a private key, and $x_Q \in \mathbb{F}_p$ is a public key

---

$d \xleftarrow{\$} \{0,1\}^N$
$(h_{2N-1}, \ldots, h_0)_2 \leftarrow H(d)$
$d_0 \leftarrow (h_{2N-1}, \ldots, h_N)_2$
$d_1 \leftarrow (h_{N-1}, \ldots, h_0)_2$
$Q = (X_Q : Z_Q) \leftarrow d_0 G$
$x_Q \leftarrow X_Q / Z_Q$
return (d_0, d_1) and x_Q

Algorithm 4 qDSA: Signature generation.

Input:

(d_0, d_1) and x_Q are the signer's keys; and $M \in \{0,1\}^*$ is a message.

Output:

$(x_R \parallel s)$ is the signature of M, where $x_R \in \mathbb{F}_p$ and s

---

$r \leftarrow H(d_1 \parallel M) \bmod \ell$
$R = (X_R : Z_R) \leftarrow rG$
$x_R \leftarrow X_R / Z_R$
$h \leftarrow H(x_R \parallel x_Q \parallel M)$
$s \leftarrow r - h d_0 \bmod \ell$
return $(x_R \parallel s)$

Algorithm 5 qDSA: Signature verification.

Input:

x_Q is the public key of the signer, $(x_R \parallel s)$ is a signature, and $M \in \{0,1\}^*$ is a message.

Output:

`True`, if the signature is valid; otherwise, `False`.

$Q \leftarrow (x_Q : 1)$
$h \leftarrow H(x_R \parallel x_Q \parallel M) \mod \ell$
$R_0 \leftarrow x(sG)$
$R_1 \leftarrow x(hQ)$
return Check(x_R, R_0, R_1)

Algorithm 6 Check $x_R \in \{x(P \pm Q)\}$.

Input:

$x_R, x_{R1}, x_{R2} \in \mathbb{F}_p$

Output:

`True` if $x_R = x_{R1} + x_{R2}$; otherwise `False`.

Let $f(x) \leftarrow f_2x^2 + f_1x + f_0$ such that f_i are defined as in Equation 2.7.
if $f(x_R) = 0$ **then**
    return `True`
else
    return `False`
end if

2.7 Summary

Chapter 2 presented the fundamental concepts, such as concerns raised by the sensitive data manipulation by not-so-powerful devices and lack of security mechanisms, and the mathematical background explored in this work, such as the finite field definition, elliptic curve concepts, and the complexity of related attacks. Protocols based on the Curve25519 and its birrationally equivalent curve are shown, powering two goals of public-key cryptography.

Chapter 3

Implementation of the Finite Field Arithmetic

Efficiently implementing the arithmetic operations of a cryptosystem is the most fundamental way to improve performance in cryptographic schemes. In order to speed up group operations in the Curve25519 and its Twisted Edwards birrationally equivalent curve, strategies to operate with 256-bit long numbers were designed and implemented to speed up basic arithmetic operations of numbers usually larger than natively handled by computer architectures.

In this chapter, implementations of the Finite Field modulo $2^{255} - 19$ arithmetic operations are explored, with emphasis on the most computationally expensive operations: multiplication and squaring.

3.1 Representation, Addition and Subtraction

Each 255-bit integer field element can be densely represented, using 2^{32}-radix, implying in eight "limbs" of 32 bits, each one in a little-endian format. This contrasts with the reference implementation [5], which uses 25 or 26 bits in 32-bits signed words, allowing carry values to be fit in the remaining bits. This require proper handling where those bits cannot be transmitted to other parties. Due to its simplicity, the first representation is used in this implementation.

The 256-bit addition is implemented by respectively adding each limb in a lower to higher element fashion in both options. In the cast of the first representation, the carry flag, present in the ARM status register, is used to ripple the carry across the limbs without overhead, with the Add-with-Carry (`ADC`) instruction handling it. The result must be always less than $2^{256} - 1$, fitting in the 8 32-bit long limbs, requiring further handling of the carry flag if it is set by the end of the routine.

A "weak" modular reduction modulo $2^{256} - 38$ is performed at the end of every field operation in order to avoid extra carry computations between operations, as suggested in [41]; this reduction must find a integer less than 2^{256} that is congruent modulo $2^{255} - 19$. In the case of addition, which might set flag the carry bit, subtracting $2p = 2^{256} - 38$ from the value may not yet clear the carry flag. This is the case when, for example, $a = 2^{256} - 1$

and thus $2a = 2^{257} - 2$; removing $2p$ yields in $2a - 2p = 2^{257} - 2^{256} + 36$, which overflows the 8×32 bits representation. To solve this, removing an extra $2p$ results in an 256-bits long integer. This subtraction can be efficiently computed by simply adding 38 to the first limb with no risk of setting the carry flag (and thus avoiding more carry handling).

Note that the second subtraction must be only performed when the addition operation sets the carry flag and if the first $2p$ subtraction does not set the borrow bit, which in most CPU architectures is stored as the complement of the carry bit present in status registers. This may raise concerns in regards to storing the bit, since the borrow flag may be set in the second subtraction. If it is the case, the generated carry bit counters the borrow effect.

To avoid timing issues, all those operations must be executed in constant time, taking advantage of complement of two arithmetic implemented in CPUs to conditionally generate masks. Listing 3.1 shows a constant-time (isochronous) ARM implementation of this strategy.

Listing 3.1: ARM assembly code handling carries on $\mathbb{F}_{2p=2^{256}-38}$ addition. Extra handling (last 6 instructions) is needed to make sure that result is in F_{2p}.

```
@ [r3 - r10] holds A + B

@ if c = 1 -> 2p
@ else     ->  0
MOV r14, #-1
ADC r14, r14, #0
MVN r14, r14
AND r12, r14, #-38

@ subtract if carry bit is set
SUBS  r3,  r3, r12
SBCS  r4,  r4, r14
SBCS  r5,  r5, r14
SBCS  r6,  r6, r14
SBCS  r7,  r7, r14
SBCS  r8,  r8, r14
SBCS  r9,  r9, r14
SBCS r10, r10, r14

@ need to remove 2p again if subtraction didn't require a borrow
@ do it by resetting the borrow bit and subtract 38
MOV r12,  #0
SBC r12, r12, r12       @ detect if borrowed on sub
MVN r12, r12
AND r12, r12, r14       @ detect if  carried on sum
AND r12, r12, #38       @ mask 38
ADD r3,  r3,  r12       @ add 38 if carried sum and didn't borrow
```

Subtraction follows the same strategy: each 32-bit limb of the 256-bit number is subtracted in a fashion similar to addition, with the help of the borrow flag to avoid performance overheads. If, by the final of the subtraction routine, the borrow flag is set, $2p$ is added to the current result conditionally. This still might not bring the final result

to a value greater than 0, so, again, an addition of $2p$ must be done. To compute this in an efficient manner, this can be implemented as a simple subtraction of the lowest 32-bit limb with 38 and resetting the borrow flag if the initial subtraction required a borrow and the first $2p$ addition didn't set the carry flag. As always, secure implementations of this algorithm must use masks and conditional operations to avoid timing differences.

Reference implementations do not have to follow strict rules as earlier defined, since the extra remaining bits of each limb may be used to handle overflow or underflow situations. Those must be treated in the modular reduction modulo $p = 2^{255} - 19$ procedure, used when results must be communicated to the other party. This operation is not used in-between operations because the "weak" modular reduction, in the case after multiplication and squarings, takes approximately 10% less cycles than a modular reduction modulo p.

3.2 Multiplication

In regard of the multiplication, two cases must be considered: multiplying a 256-bit number by a 32-bit number (here denominated as a "multiplication by a word") and multiplying two 256-bit numbers.

3.2.1 Multiplying a 256-bit number with a 32-bit word

The multiplication by a word operation is used a single time when doubling a point in X25519, where a 256-bit long integer must be multiplied by $d = 121666$. This operation follows the algorithm described in [42] and an ARM implementation is shown in Listing 3.2. The main idea revolves using the multiply-and-accumulate instructions to compute the multiplication between the least significant limb and the 32-bit word then adding back to the same limb. The same strategy remains for the upper parts of the 256-bit number, albeit the 32-bit number must be subtracted by one as the `UMAAL` instruction adds back the in-processing limb and the higher part of the first multiplication.

Listing 3.2: ARM assembly code to compute a multiply a 256-bit word with a 32-bit word. Adapted from [42].

```
@ r2 holds a 32-bit integer
@ [r3-r10] holds a 256-bit integer

UMULL  r3, r11,  r3, r2
  SUB  r2,  r2,  #1
UMAAL  r4, r11,  r4, r2
UMAAL  r5, r11,  r5, r2
UMAAL  r6, r11,  r6, r2
UMAAL  r7, r11,  r7, r2
UMAAL  r8, r11,  r8, r2
UMAAL  r9, r11,  r9, r2
UMAAL r10, r11, r10, r2
```

```
@ reducing r11
MOV   r14, #38
MOV   r12, #0

UMLAL  r3, r12, r11, r14
ADDS   r4,  r4, r12
ADCS   r5,  r5, #0
ADCS   r6,  r6, #0
ADCS   r7,  r7, #0
ADCS   r8,  r8, #0
ADCS   r9,  r9, #0
ADCS  r10, r10, #0
```

As in Listing 3.2, the result is stored in $(a_0, \ldots a_7, \texttt{rHI}) = (\texttt{r3}, \ldots r10, r11)$. To make this result fit in a 256-bit space, finding a number less than $2^{256} - 1$ congruent modulo p is needed. Since $2^{256} \equiv 38 \mod 2p$, thus, $a2^{256} \equiv a38 \mod 2p$, multiplying rHI by 38 then adding back to a_0 is sufficient. This addition may set up the carry flag; this bit must be rippled across a_1 to a_7. Again, when rippling the carry into a_7, a residual flag may be set again; this time, a $2p$ subtraction strategy like on the case of addition must be done.

3.2.2 Multiplying two 256-bit numbers

This operation relies on either operand (Algorithm 7) or product scanning (Algorithm 8) forms of integer multiplication of two W-bit words [33]. These algorithms are written in a form to take advantage of a multiplication operation resulting in a $(2W)$-bit result in a W-bit computer architecture. This requires a modular reduction modulo p. For the $\mathbb{F}_p$ implementation, a reduction modulo $2p$ is enough to fit in 256 bits.

Notation (UV) denotes a $(2W)$-bit quantity, obtained from the concatenation of U and V, both W-bit words; R_0, R_1, R_2, U, and V are W-bit words. Figure 3.1 exemplifies the operand scanning algorithm with $8 \times 8 \rightarrow 16$ words. Figure 3.2 shows the same operation but in the product scanning form. Each dot in the rhombus represents one of the n^2 W-bit word $\times$ W-bit word $\rightarrow 2W$ product, while thicker lines show a path to execute the multiplications.

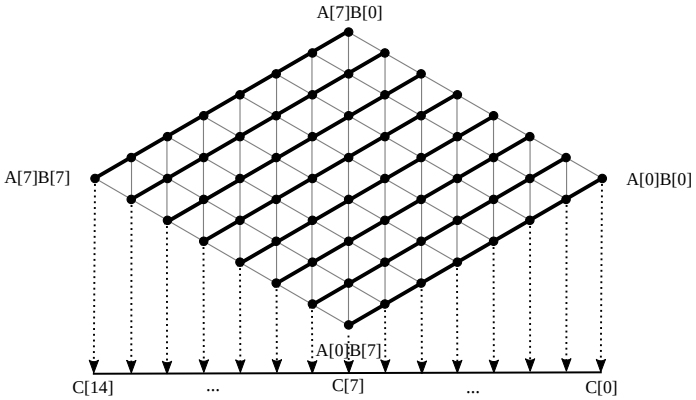


Figure 3.1: Integer multiplication of 8 words each, using the operand scanning form (adapted from [43]).

Algorithm 7 Integer multiplication, operand scanning form.

Input: Integers $a, b \in [0, p-1]$
Output: $c = a \cdot b$

$C[i] \leftarrow 0$ for $0 \leq i \leq t-1$
for i from 0 to $t-1$ **do**
  $U \leftarrow 0$
  for j from 0 to $t-1$ **do**
    $(UV) \leftarrow C[i+j] + A[i] \cdot B[j] + U$
    $C[i+j] \leftarrow V$
  end for
  $C[i+t] \leftarrow U$
end for
return c

Algorithm 8 Integer multiplication, product scanning form.

Input: Integers $a, b \in [0, p-1]$
Output: $c = a \cdot b$

$R_0 \leftarrow 0, R_1 \leftarrow 0, R_2 \leftarrow 0$
for k from 0 to $2t-2$ **do**
  for each $\{(i,j) | i+j = k, 0 \leq i, j \leq t-1\}$ **do**
    $(UV) \leftarrow A[i] \cdot B[j]$
    $(\varepsilon, R_0) \leftarrow R_0 + V$
    $(\varepsilon, R_1) \leftarrow R_1 + U + \varepsilon$
    $R_2 \leftarrow R_2 + \varepsilon$
  end for
  $C[k] \leftarrow R_0, R_0 \leftarrow R_1, R_2 \leftarrow 0$
end for
$C[2t-1] \leftarrow R_0$
return C

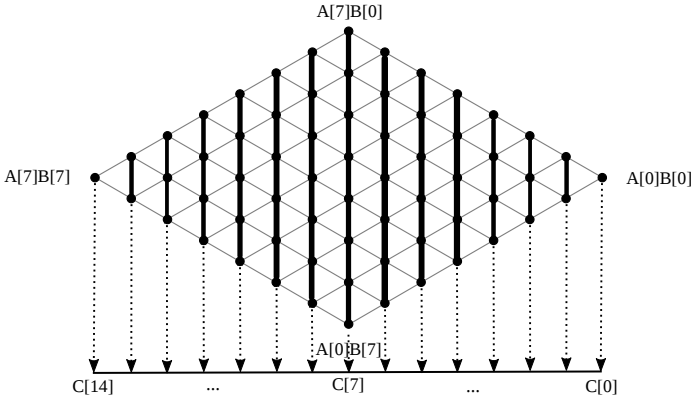


Figure 3.2: Integer multiplication with 8 words each, using the product scanning form (adapted from [43]).

Let n be the number of words enough to represent a large integer. The outer loop of the operand scanning algorithm loads the operand $A[i], 0 \leq i < n - 1$ and keeps the value constant inside the inner loop, which loads $B[j], 0 \leq k < n - 1$ word by word and multiplies with the loaded word from A. This product is then accumulated to an intermediate result on the same column, already buffered in a register or loaded from memory.

The product scanning approach processes partial products in a column-wise manner, allowing that operands of each column can be multiplied and added back to back in a multiply-accumulate scheme, resulting into a final word of the result for each column. No intermediate results must be stored or loaded; in addition, carry propagation can be easily handled, once this value can be simply added to the results of the next column. This method only needs five working registers to perform multiplication: 2 for operands and 3 for accumulators, making it a suitable option for devices with limited registers. On the memory access count, operand scanning takes $3n^2 + 2n$ memory operations to load the operands and write the results back; product scanning takes $2n^2 + 2n$ accesses to do the same [43].

Hybrid methods combining both operand and product scanning forms may be used to diminish the count of memory accesses; the main idea is to minimize the number of loads of the inner loops. This approach can be implemented using two nested loops: the outer one following the product scanning idea and the inner one using operand scanning. Since more accumulators are needed to run operand scanning internally, $2d + 1$ registers must be used, where the d parameter defines the number of rows to process within the inner loop. If $d = 1$, the hybrid method falls off to the product scanning approach; if $d = n$, hybrid multiplication is equal to the operand scanning algorithm. Figure 3.3 exemplifies this approach with $d = 4$.

Algorithms 7, 8, and hybrid approaches have quadratic complexity depending on the word size n. A divider-and-conquer approach called as Karatsuba's Algorithm can be used to lower this complexity to $\mathcal{O}(n^{\log_2 3})$ [33]. Let $n = 2l$ and $x = x_1 2^l + x_0$ and

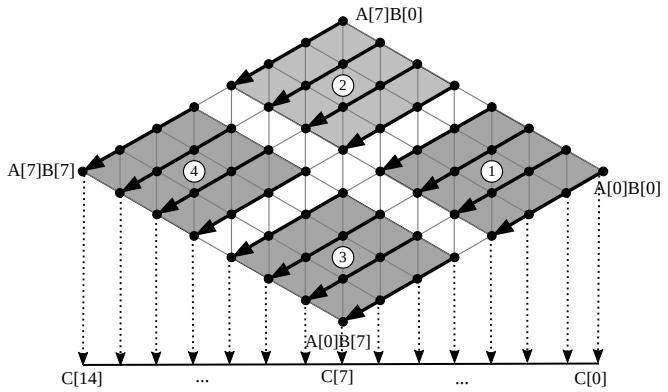


Figure 3.3: Integer multiplication with 8 words each, using hybrid approach (adapted from [43]).

$y = y_1 2^l + y_0$ are $2l$-bit integers; then we have:

$$\begin{aligned} xy &= (x_1 2^l + x_0)(y_1 2^l + y_0) \\ &= x_1 y_1 2^{2l} + [(x_0 + x_1) \cdot (y_0 + y_1) - x_1 y_1 - x_0 y_0] 2^l + x_0 y_0. \end{aligned}$$

Product xy thus can be computed using three multiplications of l-bits integers (as opposed of one $2l$-bit multiplication), along with two additions and two subtractions. For a large l, the linear cost of an addition or a subtraction is lower in comparison to the cost of a quadratic l-bit multiplication, offering better performance in these cases. As an example of such an application, De Santis and Sigl [42] field implementation on the Cortex-M4 features a two-level Karatsuba multiplier implementation, splitting a 256-bit multiplier down to 64-bit multiplications, each one taking four hardware-supported $32 \times 32 \rightarrow 64$ multiplication instructions. This algorithm is considered to require more resources and memory accesses on microcontrollers than the quadratic methods, given this algorithm exchanges multiplications for more than one addition plus the fact that modern architectures offer the same performance for addition and multiplication operations [44].

Memory accesses can be accounted for part of the time consumed by the multiplication routine. Thus, algorithms and instruction scheduling methods which minimize those memory operations are highly desirable, specially on not-so-powerful processors with slow memory access. This problem can be minimized using the product scanning strategy. However, following this scheme in its traditional way requires multiple stores and loads from memory, since the number of registers available may be not sufficient to hold the full operands. Improvements to reduce the amount of memory operations are present in the literature: namely, Operand Caching due to Hutter and Wegner [43], further improved by the Consecutive Operand Caching [45] and the Full Operand Caching, both due to Seo *et al.* [46].

Inspired by the hybrid approach, where operands are reloaded between iterations of the outer loop, Operand Caching introduces the concept of e *rows* on the product scanning form to compute partial results. By keeping operands in registers between blocks of the hybrid approach, the number of loads and save from memory operations

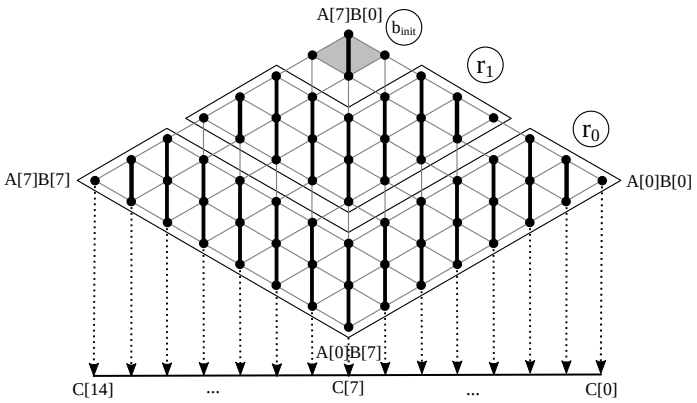


Figure 3.4: Operand Caching method to multiply two 8-word integers. Adapted from [43].

is reduced. This method is illustrated in Figure 3.4.

This method divides product scanning in two steps:

- **Initial Block**: The first step loads part of the operands, and proceeds to calculate the upper part of the rhombus using classical product scanning.
- **Rows**: In the rightmost part, most of the necessary operands are already loaded from previous calculations, requiring only some extra, low-count operand loads, depending on row width. Product scanning is done until the row ends. Note that, at the end of each column, parts of the operands are previously loaded, hence a small quantity of loads is necessary to evaluate the next column.

The e parameter defines how many words of each operand is saved in registers in a given time. Using the case of Figure 3.4, e is set to 3; in total, 9 registers are needed to compute this multiplication. Calculation must be separated in $r = \left\lfloor \frac{8}{3} \right\rfloor = 2$ rows r_0 and r_1, plus a smaller block b_{init}.

Note that at every row change, new operands need to be reloaded, since the current ones in registers are not useful at the start of the new row. Let n be the number of "limbs". Full Operand Caching further improves the quantity of memory access in two cases: if $n - re < e$, the Full Operand Caching structure looks like the original Operand Caching, but with a different multiplication order. Otherwise, Consecutive Operand Caching bottom row's length is adjusted in order to make full use of all available registers at the next row's processing.

Catching the Carry Bit. Using product scanning to calculate partial products with a double-word multiplier implies adding partial products of the next column, which in turn might generate carries. A partial column, divided in rows in a manner as described in Operand Caching, can be calculated using Algorithm 9; an example of implementation in ARM Assembly is shown in Listing 3.3. Notation follows as $(\varepsilon, z) \leftarrow w$ meaning $z \leftarrow w \mod 2^W$ and $\varepsilon \leftarrow 0$ if $w \in \left[0, 2^W - 1\right]$, otherwise $\varepsilon \leftarrow 1$, where W is the bit-size of a word; (AB) denotes a $2W$-bit word obtained by concatenating the W-bit words A and B.

Algorithm 9 Column computation in product scanning.

Input: Operands A, B; column index k; partial product R_k (calculated during column $k-1$); accumulated carry R_{k+1} (generated from sum of partial products).
Output: (Partial) product $AB[k]$; sum R_{k+1} (higher half part of the partial product for column $k+1$); accumulated carry R_{k+2} (generated from sum of partial products).

$R_{k+2} \leftarrow 0$
for all $(i, j) \mid i + j = k, 0 \leq i < j \leq n-1$ **do**
  $T \leftarrow 0$
  $(TR_k) \leftarrow A[i] \times B[j] + (TR_k)$
  $(\varepsilon, R_{k+1}) \leftarrow T + R_{k+1}$
  $R_{k+2} \leftarrow R_{k+2} + \varepsilon$
end for
$AB[k] \leftarrow R_k$
return $AB[k]$, R_{k+1}, R_{k+2}

Listing 3.3: ARM code for calculating a column in product scanning.

```
@ k = 6
@ r5 and r4 hold R_6, R_7 respectively
@ r6, r7, r8 hold A[3], A[4] and A[5] respectively
@ r9, r10, r11 hold B[3], B[1], B[2] respectively
MOV   r12, #0
MOV   r3,  #0
UMLAL r5,  r12, r8, r10 @ A5 B1
ADDS  r4,  r4,  r12
ADC   r3,  r3,  #0
MOV   r14, #0
UMLAL r5,  r14, r7, r11 @ A4 B2
ADDS  r4,  r4,  r14
ADC   r3,  r3,  #0
MOV   r12, #0
UMLAL r5,  r12, r6, r9 @ A3 B3
ADDS  r4,  r4,  r12
ADC   r3,  r3,  #0
@ r5 holds AB[6], r4 holds R_7, @ r3 holds R_8
```

One possible optimization is **delaying the carry bit**: eliminating the last addition of Algorithm 9. This addition can be deferred to the next column with the use of a single instruction to add the partial products and the carry bit. This is easier on ARM processors, where there is fine-grained control of whether or not instructions may update the processor flags. Other optimizations involve proper register allocation in order to avoid reloads, saving up a few cycles.

Carry Elimination. Storing partial products in extra registers without adding them avoids potential carry propagation. In a trivial implementation, a register accumulator may be used to add the partial values, potentially generating carries. The `UMAAL` instruction can be employed to perform such addition, while also taking advantage

of the multiplication part to further calculate more partial products. This instruction never generates a carry bit, since $(2^n-1)^2+2(2^n-1)=(2^{2n}-1)$, eliminating the need for carry handling. Partial products generated by this instruction can be forwarded to the next multiply-accumulate(-accumulate) operation; this goes on until all rows are processed. Algorithm 10 and Listing 3.4 illustrate how a column from product-scanning can be evaluated following this strategy.

Algorithm 10 Column computation in product scanning, eliminating carries.

Input: Operands A, B; column index k; m partial products $R_k[0,\ldots,m-1]$ (calculated during column $k-1$ and stored in registers).

Output: Partial product $AB[k]$; m partial products $R_{k+1}[0,\ldots,m-1]$ (higher half part of the calculated partial product for column $k+1$ stored in registers).

$t \leftarrow 1$
for all $(i,j) \mid i+j=k, 0 \leq i < j \leq n-1$ **do**
  $(R_k[t]R_k[0]) \leftarrow A[i] \times B[j] + R_k[0] + R_k[t]$
  $R_{k+1}[t-1] \leftarrow R_k[t]$
  $t \leftarrow t+1$
end for
$AB[k] \leftarrow R_k[0]$
return $AB[k]$, $R_{k+1}[0,\ldots,m-1]$

Listing 3.4: ARM code for calculating a column in product scanning without carries.

```
@ k = 6
@ r3, r4, r12 and r5 hold R_6[0,1,2,3]
@ r6, r7, r8 hold A[3], A[4] and A[5] respectively
@ r9, r10, r11 hold B[3], B[1], B[2] respectively
UMAAL r3, r4,  r8, r10 @ A5 B1
UMAAL r3, r12, r7, r11 @ A4 B2
UMAAL r3, r5,  r6, r9  @ A3 B3
@ r3 holds (partially) AB[6]
@ r4, r5 and r12 hold partial products for k = 7
```

Note that this strategy is limited by the number of working registers available. These registers hold partial products without adding them up, avoiding the need of carry handling, so strategies diving columns into rows like in Operand Caching fits in this case.

Figure 3.5 show an toy example of multiplication using the product scanning approach with 3-word sized operands A and B using the `UMLAL` and `UMAAL` instructions. Figure 3.6 show a toy example for the same case, but using the operand scanning approach.

3.2.3 Squaring a 256-bit number

The squaring operation can be seen as a specialization of the multiplication algorithm; hence, all algorithms for the later operation work when both arguments are the same.

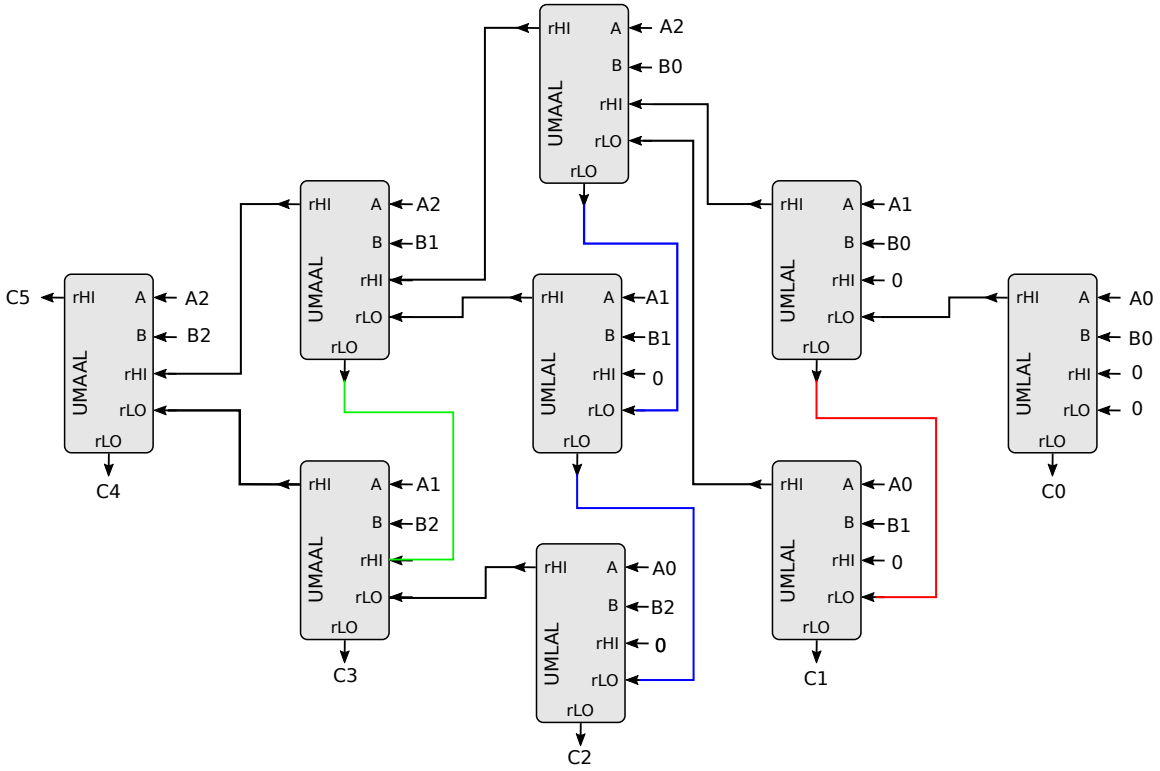


Figure 3.5: Multiplying two 3-word integers with the product scanning approach using the UMLAL and UMAAL instructions. Note that no carry values are generated.

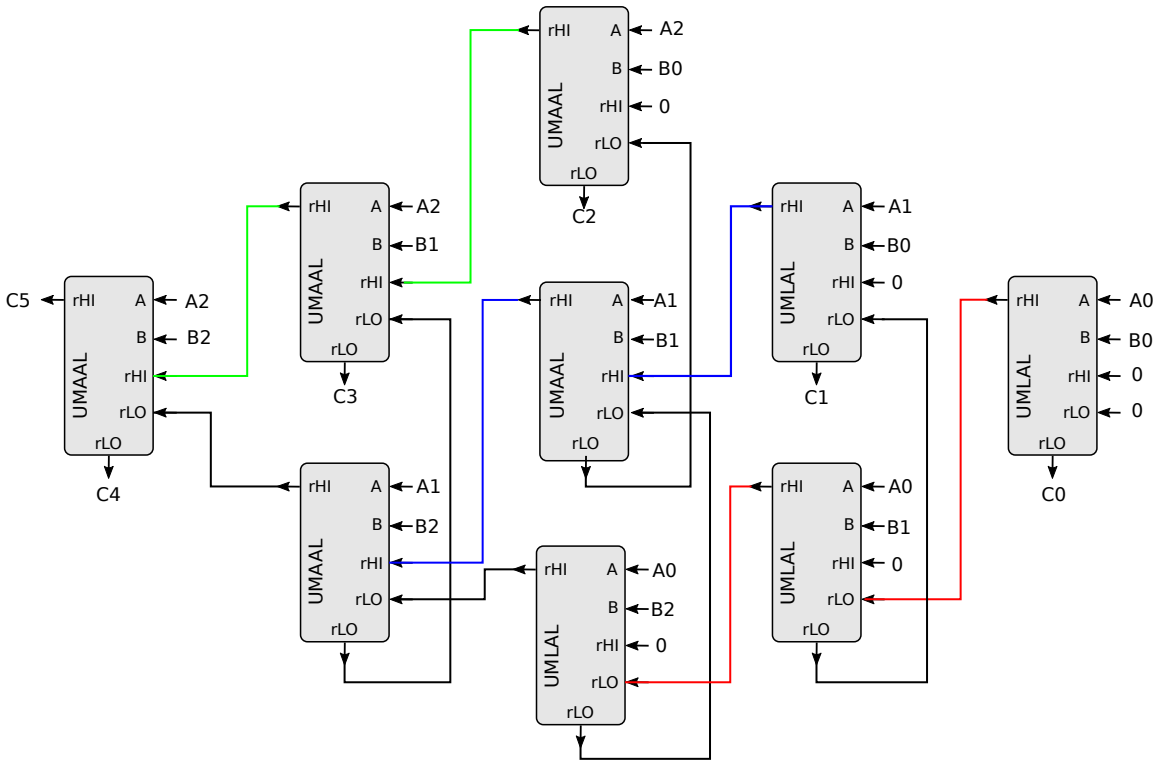


Figure 3.6: Multiplying two 3-word integers with the operand scanning approach using the UMLAL and UMAAL instructions. Note that no carry values are generated.

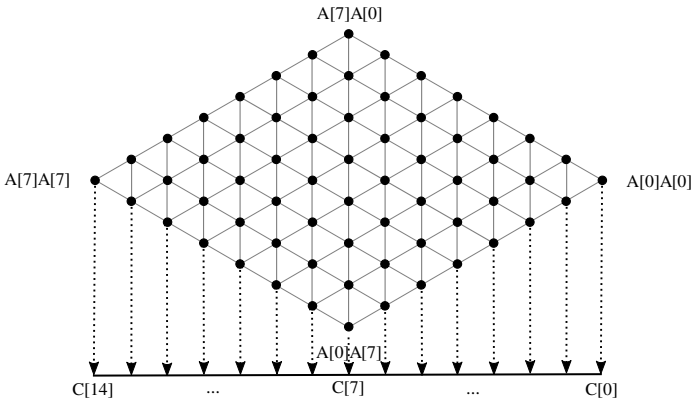


Figure 3.7: Squaring a 8-limb number using a full multiplication algorithm; black dots represent multiplications.

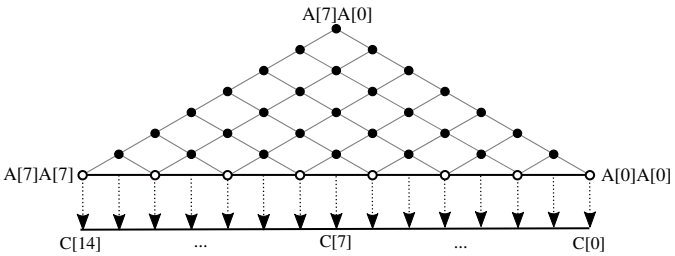


Figure 3.8: Squaring a 8-limb number halving the squaring structure; black dots represent multiplications; hollowed dots represent squarings.

Specialized implementations may have two advantages in comparison to multiplication implementations: first, due to the fact that only one operand is used for squaring computation, the number of loads can be halved in comparison to multiplication. Secondly, many registers used to hold operands can be freed, and can be used for other purposes such as for caching intermediate results or other values. The disadvantage of spending more program space two fit a generalized operation (multiplication) and the specialized one (square) must be accounted for when writing software for space-constrained environments.

A multiplication algorithm similar to the operand or product-scanning may be used (Figure 3.7), with optimizations towards removing unneeded operations and use cheaper alternatives to compute the same result (Figure 3.8). For example, the new algorithm may cut out redundant internal products and use efficient operations (such as left shift by one) to double needed internal products. In these cases, hybrid approaches combining both operand and product-scanning techniques may be used. [45] notes that this approach "lacks optimal use of working registers" and "inefficiently deals with the carry bit produced when adding two partial products", hence introducing the *Sliding Block Doubling* (SBD) algorithm.

To evaluate the square of a 8-part number as exemplified in Figure 3.9, SBD works by computing partial results of the square using the product-scanning method; to account the doubled products, a left shift by 1 is issued to the partial results and saved into memory. Afterwards, squares from parts of the operands are computed; then partial results saved in memory are retrieved and added to the partial squares.

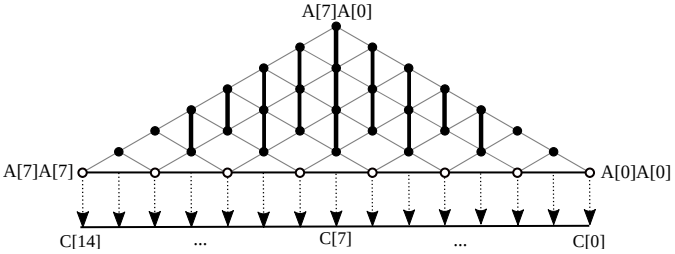


Figure 3.9: Sliding Block Doubling: black dots represent multiplications; hollowed dots represent squarings.

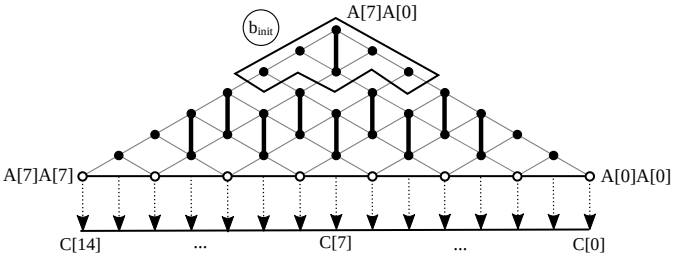


Figure 3.10: Sliding Block Doubling, computing an initial block beforehand.

This algorithm runs the following steps:

- **Partial Products of the Upper Part Triangle**: an adaption of product scanning calculates partial products (represented by the black dots at the superior part of the rhombus in Figure 3.9) and saves them to memory.
- **Sliding Block Doubling of Partial Products**: each result of the column is doubled by left shifting each result by one, effectively duplicating the partial products. This process must be done in parts because the number of available registers is limited, since they hold parts of the operand.
- **Remaining Partial Products of the Bottom Line**: bottom line multiplications are squares of part of the operand. These products must be added to their respective partial result of its above column.

With the usage of carry flag present in the ARM architecture, both *Sliding Block Doubling* and the *Bottom Line* steps can be efficiently computed. In order to avoid extra memory access, those two routines may be implemented without reloading operands; because of the need of the carry bit in both those operations, high register pressure may arise in order to save them into registers. Calculating some multiplications akin to the Initial Block step, as in the Operand Caching multiplication method, reduces register usage by spilling partial results in memory. This allows proper carry catching and handling in exchange for a few memory accesses. This method is exemplified in Figure 3.10.

This algorithm runs the following steps:

- **Triangular Initial Block**: A halved "initial block" calculates some of the multiplications in order to relieve register pressure for the next steps; partial results are spilled to memory.

- **Remaining Partial Products and Squares**: in a right-to-left fashion, each column of the product-scanning-based multiplication is computed. The result of a column is doubled by left-shifting by one; a possible overflow is caught and saved into a working register. If the column computes a square, the doubled partial result is added to the square. The carry bit can be also be set here: a working register must capture it for handling in the next column processing.

Note that the working registers freed up to by the first step are used to save carry bits generated by the doubling operation or during the addition of the squares. In the case of the example in Figure 3.10, by calculating the "initial block", each product-scanning column of the next phase is limited to height 2, meaning that only two consecutive multiplications can be handled without losing partial products. During evaluation of the remaining partial products, those cached products are loaded and added to the current results (using the `UMAAL` or `UMLAL`) instructions; afterwards, the saved carry bits must be handled.

3.3 Summary

Chapter 3 explored the main implementation strategy of $\mathbb{F}_{2^{255}-19}$ operations, emphasizing multiplication and squaring. Alternative algorithms were explored in attempts to speedup the multiplication operation, taking figures of Karatsuba's algorithm-based implementations as baseline. Additionally, with multiplication-accumulation instructions present in ARM architectures, an efficient implementation was developed taking advantage of those one cycle to execution instructions

Optimizations found in the multiplication operation were ported to squaring, since both have similar structure. Utilizing the CPU's capabilities, such as the carry flag and the barrel shifter, improvements to algorithms found in literature are presented.

Chapter 4

Protocol Implementation Details and Performance Evaluation

In this chapter, the performance figures for the implementations measured in cycle counts are shown, along with the compiled code size of them. A brief description on how the protocols were implemented is also presented, exploring where critical operations have bottlenecks in the explained cryptosystems and which approach was considered the best for decreasing their impact.

4.1 Testing Environment

The focus of this work is given to microcontrollers suitable for integration within embedded projects. Therefore, we choose some representative ARM architecture processors. Specifically, the implementations were benchmarked on the following platforms:

- **Teensy**: Teensy 3.2 board equipped with a MK20DX256VLH7 Cortex-M4-based microcontroller, clocked at 48 and 72 MHz.
- **STM32F401C**: STM32F401 Discovery board powered by a STM32F401C microcontroller, also based on the Cortex-M4 design, clocked at 84MHz.
- **Cortex-A7/A15**: ODROID-XU4 board with a Samsung Exynos5422 CPU clocked at 2 GHz, containing four Cortex-A7 and four Cortex-A15 cores in a heterogeneous configuration.

Code for the Teensy board was generated using GCC version 5.4.1 compiled with the `-O3 -mthumb` flags; same settings apply for code compiled to the STM32F401C board, but using an updated compiler version (7.2.0). For the Cortex-A family, code was generated with GCC version 6.3.1 using the `-O3` optimization flag. Cycle counts were obtained using the corresponding cycle counter in each architecture. Randomness, where required, was sampled through `/dev/urandom` on the Cortex-A7/A15 device. In the Cortex-M4 boards, NIST's `Hash_DRBG` is implemented with SHA256 and the generator is seeded by analogically sampling disconnected pins on the board. This sampling may not work in all cases, since (a) variation may be not big enough to change

states, generating no entropy or (b) disconnected pins may be pulled up or down by circuitry, permanently setting their electric tension.

Albeit not the most efficient for every possible target, the codebase is the same for every ARMv7 processor equipped with DSP instructions, being ideal to large heterogeneous deployments, such as a network of smaller sensors connected to a larger central server with a more powerful processor than its smaller counterparts. This helps code maintenance, avoiding possible security problems.

4.2 Implementation Details and Timings

Table 4.1 presents timings and Table 4.2 shows the code size for field operations with implementation described in Chapter 3. In comparison to the previous state-of-art [42], our addition/subtraction take 18% less cycles; the 256-bit multiplier with a weak reduction is almost 50% faster and the squaring operation takes 30% less cycles. The multiplication routine may be used in replacement of the squaring if code size is a restriction, since 1**S** is approximately 0.9**M**. Implementation of all arithmetic operations take less code space in comparison to [42], ranging from 20% savings in the addition to 50% in the 256-bit multiplier. As experimental evaluation, using the Karatsuba algorithm to implement the $\mathbb{F}_p$ multiplication, the operation takes 768 CPU cycles on the Teensy device.

For completeness, Table 4.1 shows the cycle numbers of both 256-bit multipliers implemented; the first is the Operand-Caching inspired version, and the second one is a operand scanning-based multiplier. Since the former takes less CPU cycles, the most efficient one is used to measure the performance of upper protocols.

As noted by Hasse [47], cycle counts on the same Cortex-M4-based controller can be different depending on the clock frequency set on the chip. Different clock frequencies set for the controller and the memory may cause stalls on the former if the latter is slower. For example, operations relying on memory operations, such as the 256-bit multiplication and squaring, use 10% more cycles when the controller is set to a 33% higher frequency. This behavior is also present on cryptographic schemes, as shown in Tables 4.3 and 4.4, since those are subject to compiler interference once those were implemented in a higher level language.

4.2.1 X25519 implementation

X25519 was implemented using the standard Montgomery ladder over the x-coordinate. Standard tricks like randomized projective coordinates and constant-time conditional swaps were implemented for side-channel protection. Cycle counts of the X25519 function executed on the evaluated processors are shown in Table 4.3 and code size in Table 4.2.

As a countermeasure against power analysis, projective coordinate randomization is implemented, resulting in an approximately 1% penalty in testing. This technique adds some multiple k of the group order l to the secret scalar s; multiplying $(k\ell + S)P$

Table 4.1: Timings in cycles for arithmetic in $\mathbb{F}_{2^{255}-19}$ on multiple ARM processors. Numbers for this work were taken as the average of 256 executions; Standard deviation is near zero, since implementation is inherently protected against timing attacks (constant time). Two 256-multiplication implementations are measured: one based on the Consecutive Operand Caching ("COC") and the other one based on Operand Scanning ("OpScan"). [a]Teensy board. [b]STM32F401C board. [c]STM32F407 board.

	Cortex	**Add/Sub**	**Mult**		**Mult by word**	**Square**	**Inversion**
De Groot [37]	M4	**73/77**	631		129	563	151997
De Santis [42]	M4	106	546		**72**	362	96337
			COC	**OpScan**			
This work	M4 @ 48 MHz[a]	91/89	**284**	329	95	**251**	**66681**
	M4 @ 72 MHz[a]	91/89	**311**	358	100	**290**	**77356**
	M4 @ 84 MHz[b]	91/89	**274**	321	92	**245**	**64955**
	A7	**55/53**	**291**	362	**77**	**235**	**63249**
	A15	**38/37**	**224**	197	**72**	**138**	**41136**
	Cortex	$\mathbb{F}_{p^2}$ **Add/Sub**	$\mathbb{F}_{p^2}$ **Mult**		**Mult by word**	$\mathbb{F}_{p^2}$ **Square**	$\mathbb{F}_{p^2}$ **Inversion**
Four$\mathbb{Q}$ [48]	M4[c]	84/86	358		-	215	21056

Table 4.2: Code size in bytes for implementing arithmetic in $\mathbb{F}_{2^{255}-19}$, X25519, Ed25519 and qDSA with Curve25519 protocols on the Cortex-M4. Code size for protocols considers the entire software stack needed to perform the specific action, including but not limited to field operations, hashing, tables for scalar multiplication and other algorithms.

	Add	**Sub**	**Mult**	**Mult by word**	**Square**
De Groot [37]	**44**	**64**	1284	300	1168
De Santis [42]	138	148	1264	116	882
This work	110	108	**622**	**92**	**562**
	Inversion	**X25519**	**Ed25519 Key Gen.**	**Ed25519 Sign**	**Ed25519 Verify**
De Groot [37]	388	4140	-	-	-
De Santis [42]	484	**3786**	-	-	-
This work	**328**	4152	**21265**	**22162**	**28240**
	qDSA Key Gen.	**qDSA Sign**	**qDSA Verify**		
Left to Right Montgomery	**14546**	**20720**	**15856**		
Right to Left Montgomery [8]	24762	29756	25516		

Table 4.3: Timings in 10^3 cycles for computing the Montgomery ladder in the X25519 key exchange. Numbers were taken as the average of 256 executions in chosen ARM processors. Standard deviation is near zero, since X25519 implementation is inherently protected against timing attacks (constant time). The measures for this work takes in account the best 256-bit multiplier shown on Table 4.1.

	Cortex	**X25519**
De Groot [37]	M4	1,816.3
De Santis [42]	M4	1,563.8
This work	M4 @ 48 MHz (Teensy)	**925.7**
	M4 @ 72 MHz (Teensy)	**1,036.6**
	M4 @ 84 MHz (STM32F401)	**913.8**
Schwabe, Bernstein [10]	A8	**527.1**
This work	A7	840.0
	A15	587.0
eBACS ref. code [49]	A15	**342.4**
	Cortex	**DH**
Four$\mathbb{Q}$ [48]	M4 (STM32F407)	542.9

results in $(sP + \mathbb{O}) = sP$, hence not modifying results. From a execution standpoint, randomizing bits of the secret ensures that the execution path is unpredictable while applying the Montgomery ladder each time [37].

Our implementation is 42% faster than De Santis and Sigl [42] while staying competitive in terms of code size.

A note on conditional swaps

The conditional swap operation is classically implemented using bitwise instructions. However, this approach opens a breach for a power analysis attack, as noted in [11], since all bits from a 32-bit-long register (in ARMv7 architectures) must be set or not depending on a bit derived from the secret key.

The conditional swap operation can be implemented in a alternative way by setting the 4-bit `ge`-flag in the Application Program Status Register (`ASPR`) and then issuing the `SEL` instruction, which pick parts from the operand registers in byte-sized blocks and writes them to the destination [30]. Note that setting `0x0` to the `ASPR.ge` flag and issuing `SEL` copies one of the operands; setting `0xF` and using `SEL` copies the other one. The `ASPR` bits cannot be set directly through a `MOV` with an immediate operand, so a Move to Special Register (`MSR`) instruction must be issued. Only registers may be used as arguments of this operation, so another one must be used to set the `ASPR.ge` flag. Therefore, at least 8 bits must be used to conditionally move data between registers. This may reduce the attack surface of a potential side-channel analysis, down from 32 bits, since less energy is spent to maintain the bits set.

In terms of performance, the `SEL`-based implementation of the conditional swap uses 92 CPU cycles. For comparison, an implementation of the same operation using bitwise instructions uses 128 cycles without any optimizations. It must be noted that calls done to an experimental, ARM assembly implementation of the `SEL`-based conditional swap

are subject to function call overheads, resulting in extra 30 CPU cycles. Implementations in higher level languages using bitwise instructions are not affected, once optimizing compilers inline the method, thus skipping function call overhead.

4.2.2 Ed25519 implementation

The cycle counts for the Ed25519 implementation on choosen platforms and performance numbers for comparable implementations are shown in Table 4.4. As in 2.6.2, the most critical operation is implemented through a comb-like algorithm proposed by Hamburg in [50]. The signed-comb approach recodes the scalar into its signed binary form using a single addition and a right-shift. This representation is then divided into blocks of bits and each one of those are divided in combs, much like in the multi-comb approach described in [33].

Let n be the order of a group G. To recode the scalar e to its signed binary form using $D > \log_2 n$ digits, enough to write the scalar, i. e.

$$\sum_{i=0}^{D-1} d_i \cdot 2^i \pmod{n} \text{ where } d_i \in \{\pm 1\}$$

Note that

$$\frac{e + 2^D - 1}{2} = \sum_{i=0}^{D-1} \frac{d_i + 1}{2} \cdot 2^i.$$

To convert e into signed binary using D digits, adding $2^D - 1$ ($0_b 11 \ldots 11$) and halving (a right shift by one) is needed. If e is even, adding the *odd* group order n makes halving $(e + 0_b 11 \ldots 11) \mod n$ by right shifting by one possible. Note that this operation must be done securely in order to avoid timing differences, so all possibilities must be accounted for. Then, the final result must be choosen by issuing a conditional swap. For the implementation under discussion, recoding the scalar into 255-bit signed binary was the preferred way.

Much like in the multi-comb algorithm, this representation must be divided into j disjoint blocks of bits B:

$$e \equiv \sum_{j=0}^{n-1} B_j, \text{where } B_j := \sum_{i=o_j}^{o_{j+1}-1} d_i \cdot 2^i,$$

where o_j is the *offset* of the j block, counting from the least significant bit, so the blocks B_j are "parts" of the scalar e in signed binary form. Note that division does not have to be even: for a 255-bit signed binary representation, one can write it as $B_0 = d_0 \cdot 2^0 + d_1 \cdot 2^1 + \ldots + d_{49} \cdot 2^{49}$, $B_1 := d_{50} \cdot 2^{50} + d_{51} \cdot 2^{51} + \ldots + d_{99} \cdot 2^{99}$ (i. e. bits d_{50} to d_{99}); and so on including B_3. B_4, however, must be fit with 55 bits to account e entirely: $B_4 = d_{200} \cdot 2^{200} + d_{201} \cdot 2^{201} + \ldots + d_{255} \cdot 2^{255}$. Note that those blocks can be represented as an array of bits, e. g. bits d_0 to d_{49}.

Combs must be built for each block by getting t_j bits (*teeth*) spaced by s_j bits between teeth; by the end, s_j combs will be extracted from each block B_j. Continuing from the

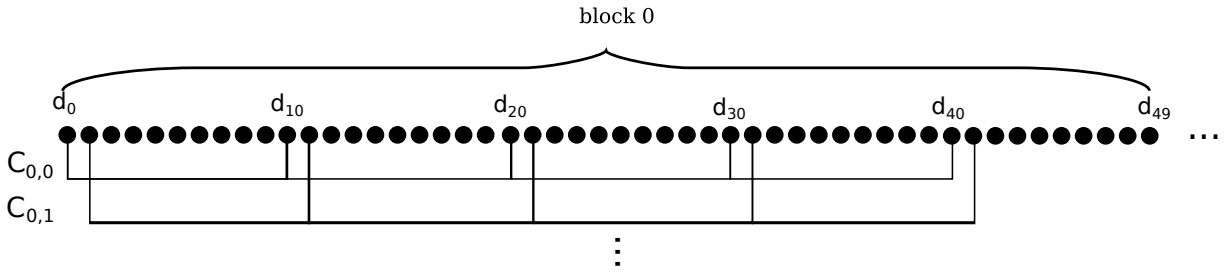


Figure 4.1: Building combs on a 50-bits block b_0.

example, 10 combs can be extracted from block B_0 (so $s_0 = 10$, hence $t_0 = 5$), getting:

$$\begin{aligned} C_{0,0} &= d_0 \cdot 2^0 + d_{10} \cdot 2^{10} + d_{20} \cdot 2^{20} + d_{30} \cdot 2^{30} + d_{40} \cdot 2^{40} \\ C_{0,1} &= d_1 \cdot 2^1 + d_{11} \cdot 2^{11} + d_{21} \cdot 2^{21} + d_{31} \cdot 2^{31} + d_{41} \cdot 2^{41} \\ &\vdots \\ C_{0,9} &= d_9 \cdot 2^9 + d_{19} \cdot 2^{19} + d_{29} \cdot 2^{29} + d_{39} \cdot 2^{39} + d_{49} \cdot 2^{49} \end{aligned}$$

Note that B_4 has 55 bits, so s_4 must be set to 11 and thus we get $C_{4,0}, C_{4,1}, \ldots, C_{4,9}, C_{4,10}$. These combs can be efficiently built using shifts and bitwise operations; for efficiency, one can bypass the block separation, building the combs directly. Figure 4.1 illustrates this process.

Mathematically, we have

$$B_j = 2^{o_j} \cdot \sum_{k=0}^{s_j - 1} 2^k \cdot C_{j,k} \text{ where } C_{j,k} := \sum_{i=0}^{t_j - 1} d_{o_j + s_j i + k} \cdot 2_{s_j i}$$

With the combs:

$$e \equiv \sum_{k=0}^{\max s_j - 1} 2^k \cdot \sum_{\substack{j=0 \\ s_j > k}}^{n-1} C_{j,k} = \sum_{k=0}^{\max s_j - 1} 2^k \cdot \sum_{\substack{j=0 \\ s_j > k}}^{n-1} \pm |C_{j,k}| \pmod{q}$$

Then, to compute a multiplication:

$$e \cdot P = \sum_{k=0}^{\max s_j - 1} 2^k \cdot \sum_{\substack{j=0 \\ s_j > k}}^{n-1} \pm |C_{j,k}| \cdot P$$

Values $C_{j,k} \cdot P$ must be precomputed beforehand for each block. Each block B_j must have its own table of $2^{t_j} - 1$ precomputed multipliers of P once the offsets varies between blocks. For this work, implementing the simple double-addition ladder in a simple script and inputting all possible $C_{j,k}$ (and the generator g as the point P) to output the precomputed values suffices.

Recall that the combs use bits from the signed binary form of the scalar e. This means

that the most significant bit works like the sign bit in signed integers; if that's true, the index must be changed to its two's complement and the looked up value must be negated. Since negation on extended projective coordinates can be efficiently computed (because $-P(x, y, z, t) = P(-x, y, z, -t)$), the lookup table's size can be reduced by half. Note that both handling of the "negative bit" of the comb and the evaluation the additive inverse element must with no timing differences to avoid leaking secret bits, so conditional swaps must be used.

In short:

1. Precompute $2^{t_j} - 1$ multiplies of P for each block. Take account the *offset* of the block to calculate the tables.

2. Convert the scalar into its signed binary notation; separate (blocks and) combs.

3. Accumulate precomputed values using the combs as index for the tables, handling the "sign" bit, in a top-to-bottom fashion.

4. Double the accumulator and start over step 3. If it's the last comb level (least significant), there's no need to double the result.

On cases where the CPU has cache memory, protection against cache attacks must be implemented. This can be done by scanning the entire precomputed table in a linear fashion, as shown in [51]. This puts an upper bound on the t_j value, since increasing it makes secure table access expensive. Using the parameters explained along, to effectively calculate the scalar multiplication, the implementation requires 50 point additions and 254 point doublings. Five lookup tables of 16 points each in Extended Projective coordinate format with $z = 1$ are used, adding up to approximately 7.5 KiB of data.

Verification requires a double-point multiplication involving the generator B and point A using a w-NAF interleaving technique [33], with a window of width 5 for the A point, generated on-the-fly, taking approximately 3 KiB of volatile memory. The group generator B is interleaved using a window of width 7, implying in a lookup table of 32 points stored in Extended Projective coordinate format with $z = 1$ taking 3 KiB of ROM. Note that verification has no need to be securely executed, since all input data is (expected to be) public. Decoding uses a standard field exponentiation for both inversion and square root to calculate the y-coordinate as suggested by [52] and [7]. This exponentiation is carried out by a chain of squares akin to computing the inverse element of the prime field, providing an efficient way to calculate the missing coordinate.

Timings for computing a signature (both protected and unprotected against cache attacks) and verification functionality in the evaluated processors can be found in Table 4.4. Arithmetic modulo the group order in Ed25519-related operations closely relates to the previously shown arithmetic modulo $2^{255} - 19$, but multiplication is implemented through classical schoolbook algorithm and Barrett reduction is used instead.

Table 4.4: Timings in 10^3 cycles for key generation, signature and verification of a 5-byte message in the Ed25519 scheme. Key generation encompasses taking a secret key and computing its public key; signature takes both keys and a message as inputs to generate its respective signature. Numbers were taken as the average of 256 executions in chosen ARM processor. Standard deviation is near zero, since implementation is inherently protected against timing attacks (constant time), in exception to the Verification procedure. Cache attacks safety on the Cortex-M4 is attained due to the lack of cache memory, while side-channel protection is explicitly needed in the Cortex-A. Performance penalties for side-channel protection can be obtained by comparing the implementations with Safe = Y over N in the same platform. These measures take account the best 256-bit multiplier shown on Table 4.1. Refer to subsection 4.3 to an overview of the works compared in. [a]Teensy board. [b]STM32F401C board. [c]STM32F407 board.

	Safe	**Cortex**	**Ed25519 Key Gen.**	**Ed25519 Sign**	**Ed25519 Verify**
This work	Y	M4 @ 48 MHz[a]	**353.0**	**501.7**	**1,323.0**
	Y	M4 @ 72 MHz[a]	**385.6**	**537.2**	**1,463.6**
	Y	M4 @ 84 MHz[b]	**394.7**	**549.0**	**1,360.8**
Schwabe, Bernstein [10]	Y	A8	-	**368.2**	**650.1**
This work	N	A7	-	426.7	1,182.3
	Y	A7	400.6	529.5	-
	N	A15	-	267.5	807.4
	Y	A15	248.7	309.0	-
eBACS ref. code [49]	Y	A7	**241.6**	**245.7**	**730.0**
Floodyberry [53]	Y	M4 @ 48 MHz[a]	693.9	750.5	1,967.7
	Y	M4 @ 72 MHz[a]	738.4	796.7	2,026.4
	Y	A7	602.6	641.3	1,744.4
	Y	A15	269.9	286.9	775.3
Floodyberry [53] + This work's $\mathbb{F}_{2^{255}-19}$	Y	A7	374.6	409.8	658.8
	Y	A15	**240.4**	**255.8**	**497.3**
	CT	**Cortex**	**Schnoor$\mathbb{Q}$ Key Gen.**	**Schnoor$\mathbb{Q}$ Sign**	**Schnoor$\mathbb{Q}$ Verify**
Four$\mathbb{Q}$ [48]	Y	M4[c]	265.1	345.4	648.6

Table 4.5: Timings in 10^3 cycles for key generation, signature and verification of a 5-byte message in the qDSA scheme. Key generation encompasses taking a secret key and computing its public key; signature takes both keys and a message as inputs to generate its respective signature. Numbers were taken as the average of 256 executions in chosen ARM processors. Standard deviation is near zero, since protocol implementation is inherently protected against timing attacks (constant-time).

	Cortex	qDSA Key Gen.	qDSA Sign	qDSA Verify
Left to Right Montgomery	M4 @ 48 MHz (Teensy)	**927.9**	**1,059.1**	**1,746.2**
Right to Left Montgomery	M4 @ 48 MHz (Teensy)	**614.5**	**744.8**	**1,451.8**
	A7	**544.5**	**658.0**	**1,292.7**
	A15	**369.0**	**426.1**	**890.0**

4.2.3 qDSA implementation

The qDSA signature scheme is also impacted by the fixed-base scalar multiplication. Although a comb-like algorithm could be used in this case, a right-to-left implementation of the Montgomery ladder [8] with an auxiliary table containing multiples of the generator point replaces the double-and-add algorithm or the usual left-to-right Montgomery ladder evaluating a scalar multiplication with an already known point. This approach was choosen over the linear scan needed to protect table accesses in cases where cache memory is present, causing an extra performance overhead proportional to the table size. The cycle count measurements for both the original implementation and the improved one are shown in Table 4.5. Note that a trade off between speed and code size is present, as can be seen in Table 4.2.

Right-to-left scalar multiplications use a regular execution pattern of elliptic curve operations, aiding secure implementations against side-channel attacks, plus avoiding secret lookup indexes by using a counter to access the table. This table contains the values $\mu_i = (x_i + 1)(x_i - 1)^{-1}$, such that $(x_i, y_i) = 2^i G$ for $0 \leq i \leq 255$. The execution pattern uses two accumulators Q_0 and Q_1 and scans bits k_i of the secret scalar to decide which of the accumulators should be used to compute a differential addition with the value from the table; if $k_i \oplus k_{i-1} = 0$, Q_0 must be accumulated using Q_1 as a difference, otherwise Q_0 is accumulated using Q_1 as difference. No point doublings are required in this algorithm, in contrast to the left-to-right Montgomery algorithm.

Q_0 and Q_1 must be properly initialized since the formula for computing a differential addtion on the Montgomery model is not complete, i. e. when computing the differential addition $+_R$ with P and Q with $R = P - Q$, the formula fails when $R \in \mathcal{O}, (0, 0)$. On the original Right-to-Left Montgomery work [8], the Diffie-Hellman X25519 function has to compute the point $8kG$, but the choosen algorithm computes $kG + S$. To remove the S, a scalar multiplication with the cofactor h, obtaining, in this case, hkG. For this, S is choosen as a point of order four (i. e. $4S = \mathcal{O}$); doubling $kG + S$ thrice results in $8kG$.

Vulnerabilities originating from the use of low order point as inputs may arise [54], needing protective countermeasures. This can be achieved using the fact that the order of G is odd, as in the case of Curve25519. In this case, the point S is no longer needed, but if $\mathcal{O}$ is used as replacement, the algorithm fails. This indicates that the algorithm computes scalar multiplications if k is odd. One way to enforce that is calculating

$k'G$ instead of kG, once both maps to the same element in the Kummer variety. k' is computed by subtracting the scalar k from the order ℓ of G. If ℓ is odd, k' will be odd. Selecting whether k or k' should be processed by the ladder has to be done in a safe way using conditional swaps in order to avoid branching and, thus, timing attacks.

4.3 Comparison to other work

Implementations of this software stack are widely reported in the literature, focusing on a wide range of architectures. The ECRYPT Benchmarking of Cryptographic Systems (eBACS) project tests and benchmarks code in a variety of hardware. Open-source implementations are also integrated within the framework, allowing reproduction of reported results [49].

Düll *et al.* [41] implements X25519 and its underlying field implementation on a Cortex-M0 based processor, equipped with a simple $32 \times 32 \rightarrow 32$-bit multiplier. Due to this limitation, this multiplier is abstracted as a smaller one ($16 \times 16 \rightarrow 32$) to facilitate a 3-level Refined Karatsuba implementation, taking 1294 cycles to complete this routine on the same processor. Their 256-bit squaring uses the same multiplier strategy with standard tricks to save up repeated operations, taking 857 cycles. Putting it together, an entire X25519 operation takes about 3.6 million cycles with approximately 8 KiB of code size.

On the Cortex-A family of CPU cores, implementers may use the NEON instructions, a SIMD instruction set executed in its own unit inside the processor. Bernstein and Schwabe [10] report 527,102 Cortex-A8 cycles for the X25519 function. In the elliptic curves formulae used in their work, most multiplications can be handled in a parallel way, taking advantage of NEON's vectorization unit and Curve25519's parallelization opportunities.

Literature suggests there is no Ed25519 implementation specifically targeted at the Cortex-M4 core. Floodyberry's implementation `ed25591-donna`, geared towards non-specific 32-bit and 64-bit architectures, takes about 693.9 thousand CPU cycles to generate keys, 750.5 thousand cycles to sign a 5-byte message and 750.5 thousand cycles to verify a signature when run on a board equipped with a Cortex-M4 CPU, clocked at 48MHz. Due to its portable nature, no MAC instructions present on the Cortex-M4 and superior CPU cores are used as proposed in Chapter 3.

Bernstein's work, depending on Cortex-A8's NEON capabilities [10], reports 368,212 cycles to sign a short message and 650,102 cycles to verify its validity. The authors point out that 50 and 25 thousand cycles of signing and verification are spent by the choosen SHA-512 implementation, with room for further improvements.

Liu *et al.* reports [48] a 559,200 cycle count on a ARM Cortex-M4 based processor for their 32-bit implementation of the Diffie-Hellman Key Exchange over the Four$\mathbb{Q}$ curve. This curve, equipped with two efficiently computable endomorphisms, provides efficient variable scalar multiplication, surpassing Curve25519's variable-base scalar multiplication routine in terms of performance.

Generating keys and Schnorr-like signatures over Four$\mathbb{Q}$ takes about 796 thousand

cycles on a Cortex-M4 based CPU, while verification takes about 733 thousand cycles on the same CPU [48]. Key generation and signing are aided by a 80-point table taking 7.5KiB of ROM, and verification is assisted by a 256-point table, using 24 KiB of memory. qDSA, a Digital Signature scheme relying only on the x-coordinate, instantiated with a elliptic curve with the same underlying field as the Curve25519, takes about 3 million cycles to sign a message and 5.7 million cycles to verify it in a Cortex-M0. This last scheme does not rely on an additional table for speedups since low code size is an objective given the target architecture, although this can be done using the ideas from [8] with increased ROM usage.

The implementations presented in this work are competitive in comparison to the mentioned works, given the performance numbers shown. Albeit direct comparisons cannot be drawn due to differences on the underlying processors (and capabilities), numbers are small enough to claim the mark of "under a million cycles to run X25519 on a Cortex-M4" and point out the possibility of an efficient implementation of a signature scheme targeting a microcontroller with small memory space.

4.4 Summary

Chapter 4 presented the performance figures of the execution of higher-level protocols based on field arithmetic modulo 2^{255-19}, mostly gained from the optimized multiplication and squaring operations discussed in Chapter 3. As expected, optimizing lower level performance-critical operations impacted performance of elliptic curve operations, achieving the under a million cycles mark for the X25519 operation.

Using the same optimizations, the Ed25519 implementation also gained speedups; further performance gain was obtained by using an efficient algorithm to compute fixed-base scalar multiplication, while still taking comparable code size in relation with implementations freely available or discussed in literature. Similar gains were shown in the qDSA signature scheme; in this case, a trade-off between code size and speed is explored in order to speedup operations.

In the end, an comparative discussion between related works is presented. While not directly comparable owing to different testing platforms, implementations shown in this work are competitive in regards to the portability across different ARM platforms, allowing code reuse and avoiding security problems from different codebases.

Chapter 5

Final Remarks

This work investigated the full implementation stack of Curve25519-based cryptographic protocols for use in the ARM Cortex-M4 microcontroller. As the most performance-critical operation on the entire scheme, the $256 \times 256 \rightarrow 512$ multiplier $= \mod 2^{256} - 38$ based on the Operand Caching method is considered as the main feature of this work, impacting the performance numbers of the execution of cryptographic protocols. For further analysis and improvements, an ARM implementation is publicly available at `https://github.com/hayatofujii/curve25519-cortex-m4`.

Instead of a Karatsuba-based implementation as suggested by literature, handcrafted and carefully optimized implementations of either operand or product scanning methods has been seen as better options given the size of the operands, albeit the first method having, theoretically, better computational complexity. This finding is in line with observations seen in [50, 55]. With the usage of the DSP instructions, the 256-bit multiplier using the product scanning strategy can be implemented without using the carry bit present on CPU registers, thus avoiding expensive handling of it.

With a similar structure, the squaring operation can, in addition to the usual techniques to implement it, inherit optimizations found in the multiplication implementation. Proper usage of the features of the CPU avoids excessive operand reload and memory accesses, costly operations which slowdowns performance.

As a evidence of how arithmetical operations impacts the overall performance of higher-level protocols, reducing the time of the multiplication and squaring operations brought the X25519 function cycle cost to under a million cycles in tested platforms, making ECDH more attractive for CPUs of the ARM family. Further speedups may include precomputed tables to assist the scalar multiplication, as proposed in [8], at the expense of ROM, a limited resource on the platform. On the security side, a new way to implement the conditional swap operation, presented in this work, theoretically reduces the attack surface of a power analysis attack, as lesser bits are exposed, in comparison to the implementations publicly available.

To implement the Ed25519 signature scheme targeting a limited-space platform, an efficient algorithm runs the fixed point scalar multiplication, a performance critical operation on the scheme. With optimizations on field arithmetic, the implementation evidence that signature schemes can be efficiently implemented on Cortex-M4 platforms with little ROM required. At the expense of execution time, the code size can be fine-

tuned if needed by changing the parameters of the fixed-point scalar multiplication.

CPUs based on the ARM architecture are extremely powerful in regards to conduct integer arithmetic, mostly with its DSP instructions executing multiply-accumulate(-accumulate) operations in a single CPU cycle. In addition, usage of the carry flag is of relevant importance, but also a big headache if careful manipulation has to be done, such as by saving it into a register or doing complex handling of the flag. Unusual operations, such as additional carry logic, are expensive, taking precious CPU cycles to handle it properly. The DSP multiplication instructions fall in this category: albeit having results that might overflow, the `UMLAL` instruction does not (re)set the carry flag, requiring treatment by hand.

5.1 Future Works

Further developments may target protocols based on the Goldilocks/edwards448 curve, which have a similar structure in comparison to those based on Curve25519 but regarded to have a higher security level. In order to save ROM, the Karatsuba algorithm may be used to step up the multiplier from a 256-bit output to a 448-bit result, required on the underlying finite field.

Further optimizations on the 256-bit multiplication and squaring may be possible by, instead of reducing modulo p or $2p$ at the final of the operation, on-the-fly reduction may be executed between the product scanning columns, saving up scratch registers. Reducing register pressure aids in avoiding memory operations, which have a big impact on implementations targeting ARM microcontrollers.

An optimized implementation using compiler intrinsics or inline assembly of the conditional swap operation as suggested in Section 4.2.1 may further speedup the X25519 operation. This was not done since the experimental implementation is a handcrafted ARM assembly code, thus not subject to any compiler optimizations. As proof of concept, implementation of them aforementioned strategy in different CPU architectures is also desirable.

Bibliography

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[2] Rodrigo Roman, Jianying Zhou, and Javier López. On the features and challenges of security and privacy in distributed internet of things. *Computer Networks*, 57(10):2266–2279, 2013.

[3] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[4] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[5] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

[6] Daniel J. Bernstein. 25519 naming. Available on `https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html`, August 2014.

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

[8] Thomaz Oliveira, Julio López, Hüseyin Hisil, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-)compute a ladder - improving the performance of X25519 and X448. In *SAC*, volume 10719 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2017.

[9] Michael Hutter and Peter Schwabe. Nacl on 8-bit AVR microcontrollers. In *AFRICACRYPT*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2013.

[10] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012.

[11] Erick Nascimento, Lukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking embedded ECC implementations through cmov side channels. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 2016.

[12] H. Fujii and D. F. Aranha. Curve25519 for the Cortex-M4 and Beyond. In *Progress in Cryptology – LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America 2017, Proceedings (to appear)*, Lecture Notes in Computer Science. Springer International Publishing, September 2017.

[13] Armando Faz-Hernández, Hayato Fujii, Diego F. Aranha, and Julio López. A secure and efficient implementation of the quotient digital signature algorithm (qdsa). In *SPACE*, volume 10662 of *Lecture Notes in Computer Science*, pages 170–189. Springer, 2017.

[14] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 245 8th Street, San Francisco, CA 94103, 2018.

[15] Jeffrey Hoffstein, Jill Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[16] William Stallings. *Network Security Essentials - Applications and Standards (4. ed., internat. ed.)*. Pearson Education, 2010.

[17] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Cryptography in embedded systems: An overview. In *in Proc. of the Embedded World 2003 Exhibition and Conference*, pages 18–20, 2003.

[18] Sandro Bartolini, Roberto Giorgi, and Enrico Martinelli. Instruction set extensions for cryptographic applications. In *Cryptographic Engineering*, pages 191–233. Springer, 2009.

[19] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. Securing software cryptographic primitives for embedded systems against side channel attacks. In *ICCST*, pages 1–6. IEEE, 2014.

[20] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 19:20, 2008.

[21] David Reis Jr., Arnaldo J. de Almeida Jr., and Marco A. A. Henriques. Evaluation of elliptic curves operations over optimal extension filed on tms320vc5402 digital signal processor. In *II Congreso Iberoamericano de Seguridad Informática - CIBSI 2003*, pages 421–434. October 2003. Avaliable on `http://www.criptored.upm.es/guiateoria/gt_m400a.htm`.

[22] Jude Angelo Ambrose, Roshan G. Ragel, Darshana Jayasinghe, Tuo Li, and Sri Parameswaran. Side channel attacks in embedded systems: A tale of hostilities and deterrence. In *ISQED*, pages 452–459. IEEE, 2015.

[23] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer, 2010.

[24] Armando Hernández, Roberto Cabral, Diego Aranha, and Julio López. Implementação eficiente e segura de algoritmos criptográficos. In *Minicursos - XV Simpósio Brasileiro em Segurança da Informação e Sistemas Computacionais*, pages 93–140. Sociedade Brasileira de Computação, 2015.

[25] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs - Extended version. *J. Cryptographic Engineering*, 5(2):95–112, 2015.

[26] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014.

[27] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. Avaliable on `http://safecurves.cr.yp.to`, 2014.

[28] Sergei P. Skorobogatov. Data remanence in flash memory devices. In *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[29] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[30] ARM. Cortex-M4 Devices Generic User Guide. Avaliable on `http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.dui0553a%2FCHDBFFDB.html`, 2010.

[31] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[32] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014.

[33] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[34] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–203, jan 1987.

[35] Victor S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.

[36] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic. *Journal of Cryptographic Engineering*, Mar 2017.

[37] Wouter de Groot. *A Performance Study of X25519 on Cortex-M3 and M4*. PhD thesis, Eindhoven University of Technology, September 2015.

[38] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in gf(2ˆm) using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[39] Trevor Perrin (editor). The XEdDSA and VXEdDSA Signature Schemes. Avaliable on `https://whispersystems.org/docs/specifications/xeddsa/`, October 2016.

[40] Joost Renes and Benjamin Smith. qDSA: Small and Secure Digital Signatures with Curve-Based Diffie-Hellman Key Pairs. In *ASIACRYPT (2)*, volume 10625 of *Lecture Notes in Computer Science*, pages 273–302. Springer, 2017.

[41] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptography*, 77(2-3):493–514, 2015.

[42] Fabrizio De Santis and Georg Sigl. Towards Side-Channel Protected X25519 on ARM Cortex-M4 Processors. In *SPEED-B*, Utrecht, The Netherlands, October 2016.

[43] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer, 2011.

[44] Michael Scott. Missing a trick: Karatsuba variations. *Cryptography and Communications*, 10(1):5–15, 2018.

[45] Hwajeong Seo, Zhe Liu, Jongseok Choi, and Howon Kim. Multi-precision squaring for public-key cryptography on embedded microprocessors. In *INDOCRYPT*, volume 8250 of *Lecture Notes in Computer Science*, pages 227–243. Springer, 2013.

[46] Hwajeong Seo and Howon Kim. Consecutive operand-caching method for multi-precision multiplication, revisited. *J. Inform. and Commun. Convergence Engineering*, 13(1):27–35, 2015.

[47] Björn Haase. Memory bandwidth influence makes Cortex M4 benchmarking difficult. Avaliable on `https://ches.2017.rump.cr.yp.to/fe534b32e52fcacee026786ff44235f0.pdf`, sep 2017.

[48] Zhe Liu, Patrick Longa, Geovandro C. C. F. Pereira, Oscar Reparaz, and Hwajeong Seo. Four \mathbb Q on embedded devices with strong countermeasures against side-channel attacks. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 665–686. Springer, 2017.

[49] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. Avaliable on `https://bench.cr.yp.to`.

[50] Mike Hamburg. Fast and compact elliptic-curve cryptography.

[51] Julio López and Armando Faz-Hernández. Speeding up the Elliptic Curve Cryptography on the P-384 Curve. In *XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, 2016. Avaliable on `http://sbseg2016.ic.uff.br/pt/files/anais/completos/ST4-1.pdf`.

[52] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[53] Andrew Moon. Implementations of a fast Elliptic-curve Digital Signature Algorithm. Available at `https://github.com/floodyberry/ed25519-donna`, March 2012.

[54] Junfeng Fan, Benedikt Gierlichs, and Frederik Vercauteren. To infinity and beyond: Combined attack on ECC using points of low order. In *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2011.

[55] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. Cryptology ePrint Archive, Report 2018/286, 2018. Avaliable on `https://eprint.iacr.org/2018/286`.