POLITECNICO DI TORINO
SCUOLA DI DOTTORATO

Dottorato in Ingegneria Informatica e dei Sistemi – XVII ciclo

Tesi di Dottorato

Automatic generation of high speed elliptic curve cryptography code

Daniele Canavese

Tutore
Antonio Lioy

Coordinatore del corso di dottorato
Matteo Sonza Reorda

Maggio 2016

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

If you have to keep something secret it's because you shouldn't be doing it in the first place!

One Day
David Nicholls

Apparently trust is a rare commodity when power, money or life itself are at stake. History is full of examples. Julius Caesar did not trust his generals, so that:

> If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.

Life of Julius Caesar
Suetonius

And so the history of cryptography began moving its first steps. Nowadays, encryption has decayed from being an emperor's prerogative and became a daily life operation. Cryptography is pervasive, ubiquitous and, the best of all, completely transparent to the unaware user. Each time we buy something on the Internet we use it. Each time we search something on Google we use it. Everything without (almost) realizing that it silently protects our privacy and our secrets.

Encryption is a very interesting instrument in the toolbox of security because it has very few side effects, at least on the user side. A particularly important one is the intrinsic slow down that its use imposes in the communications. High speed cryptography is very important for the Internet, where busy servers proliferate. Being faster is a double advantage: more throughput and less server overhead. Using weak algorithms with a reduced key length to increase the performance of a system can lead to catastrophic results and it is not a suitable solution.

In 2007, TJX Cos., a U. S. apparel and home goods company, was victim of one of the biggest security breaches so far. A number of criminals intercepted the wireless transfers at two Miami-area stores and, due to weak encryption, the credit and debit cards of millions of users were compromised. In March 2007 TJX said that at least 46 millions of MasterCard and Visa cards were jeopardized [1]. Six months later the statistic was updated to 94 millions [2]. A harsh lesson for learning that the security must be taken seriously, as well as an exemplification of a bad choice in a security-performance trade-off.

Informally, a couple of algorithm $E(\cdot)$ and $D(\cdot)$ can be respectively used to encrypt and decrypt a message m with two suitable values k_E and k_D (the encryption and decryption *keys*), if and only if the following property holds:

$$D_{k_D}(E_{k_E}(m)) = m.$$

This is a necessary but not sufficient condition, since also the security of the algorithms must be taken into account. If $k_E = k_D$, $E(\cdot)$ and $D(\cdot)$ are said to be *symmetric algorithms*, otherwise if $k_E \neq k_D$ they are known as *asymmetric algorithms* or *public-key algorithms*, since usually one key is publicly available, while the other is private.

Public key algorithms have some peculiar properties making them particularly well suited for tasks such as digital signatures and sharing secrets amongst a number of parties (e.g. another key or a nonce). When encrypting huge amounts of data however, symmetric algorithms are normally preferred since they have superior performance. The high speed public-key cryptography challenge is a very practical topic with serious repercussions in our technocentric world. This is particularly interesting for the servers, where achieving a faster asymmetric encryption can speed up the opening of secure connections and the certificate validations.

Public key cryptography began its journey in 1976, when Diffie and Hellman [3] published their now famous paper «New directions in cryptography». They proposed to leverage the Discrete Logarithm Problem (DLP) to remotely and securely exchange a secret between two parties without having to meet in person, implementing a protocol now known as Diffie-Hellman (DH). However, 1978 was the time when the first true modern public key crypto-system appeared: RSA, proposed by Rivest, Shamir, and Adleman [4] and derived from the Integer Factorization Problem (IFP). Some years later, in 1985, ElGamal [5], influenced by the work of Diffie and Hellman, introduced the first asymmetric algorithm based on the DLP.

In 1985, Miller [6] and Koblitz [7] independently proposed to use the group of rational points of an elliptic curve over a finite field to create an asymmetric algorithm. Elliptic Curve Cryptography (ECC) is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP) and offers several advantages with respect to other systems based on the DLP and the IFP. The main benefit is that it requires smaller keys to provide the same security level since breaking the ECDLP is significantly harder. Figure 1 shows the recommended keys sizes for 2016 as suggested in the *Kryptographische Verfahren: Empfehlungen und Schlussellängen* [8] and the *Commercial National Security Algorithm Suite* [9], respectively for the commercial and the top secret uses.

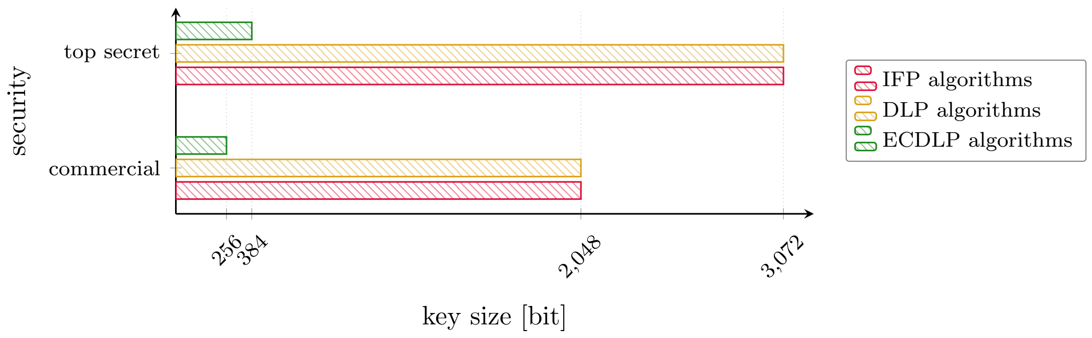


Figure 1: Suggested key lengths for 2016.

Elliptic Curve Cryptography is quite diffused nowadays. For instance, several of the most used cryptographic libraries contains at least one elliptic curve based implementation (e.g. OpenSSL and Crypto++ both offers ECDH, the elliptic curve version of DH). In addition, a number of standards exist to regulate the use of ECC in various data protection protocols (e.g. IPsec [10], TLS [11] and SSH [12]). A few documents also recommend some specific elliptic curve, most notably the *FIPS PUB 186-4 – Digital Signature Standard (DSS)* [13], that suggests 15 curves (5 on prime fields and 10 on binary fields).

The wide adoption of ECC is primarily due to the fact that, if implemented correctly, it can be very efficient both in time and memory consumption, thus being a good candidate for performing high speed public key cryptography. In addition, some elliptic curve based techniques are known to be extremely resilient to quantum computing attacks, such as the Supersingular Isogeny Diffie-Hellman (SIDH) [14].

Traditional elliptic curve cryptography implementations are optimized by hand taking into account the mathematical properties of the underlying algebraic structures, the target machine architecture and the compiler facilities. This process is time consuming, requires a high degree of expertise and, ultimately, error prone. This dissertation' ultimate goal is to automatize the whole optimization process of cryptographic code, with a special focus on ECC. The framework presented in this thesis is able to produce high speed cryptographic code by automatically choosing the best algorithms and applying a number of code-improving techniques inspired by the compiler theory. Its central component is a flexible and powerful compiler able to translate an algorithm written in a high level language and produce a highly optimized C code for a particular algebraic structure and hardware platform. The system is generic enough to accommodate a wide array of number theory related algorithms, however this document focuses only on optimizing primitives based on elliptic curves defined over binary fields.

The framework presented in this dissertation has its roots in the project started in my M. Eng. thesis [15]. It is both an extension of my previous work, but also a revolution. It is an extension because they share the same idea: automatic generation and optimization of cryptographic code. It is a revolution because this new project offers a wider array of cryptographic algorithms and provides a truly optimizing compiler w.r.t. to its predecessor that was a sort of smart translator.

The improvements achieved by the investigations presented in this dissertation and its companion framework are three folds.

First, two DSLs for describing cryptographic algorithms in a concise and natural way are proposed and described in Chapter 5. Most of the available cryptographic languages have a very limited area of application and lack a way to express some custom mathematical properties, which can be exploited by a compiler to further optimize the code. Two prominent examples are Cryptol [16], suitable only for describing simple low level cryptographic routines, and CAO [17], fitting only for implementing high level curve operations. The new languages presented in this document can be used to express both low and high level routines with an arbitrary complexity. In addition they also allow the use of Java-like annotations to specify the mathematical properties of a function such its commutativity.

Second, a new algorithm, named DCEA [18, 19], for computing the multiplicative inverse on binary fields is presented in Chapter 3. The inversion is by far the slowest field operation and having at disposal a new faster method can benefit not only the ECC area, but also other cryptographic realms based on binary fields such as hyperelliptic curves [20] and trace zero varieties [21, 22]. Experimental results show that DCEA is the fastest algorithm for smaller fields on the ARM family of processors w.r.t. to other state-of-the-art methods. This algorithm represents the culmination of a joint work with Emanuele Cesena, Rachid Ouchary, Marco Pedicini and Luca Roversi.

Third, an optimizing framework, and most notably its compiler, is proposed and discussed in Chapters 4 and 6. The current scientific research provides very few examples of cryptographic compilers, bu most of them are only tools with some smart translation capabilities. The CAO language comes with an ad-hoc compiler [17] and, apparently, it represents one of the most complete cryptographic compilers so far (excluding this thesis' work). In spite being used in the CACE FP7 European project [23], it supports only a small selection of the code-improving techniques exploited by the framework presented in this document. The experimental results consolidate the pay-off of the approach described in this dissertation, showing that the automatically generated code of the scalar multiplication is up to 7.1 times faster than the OpenSSL implementation and up to 1123.6 times faster than its Crypto++ counterpart.

Bibliographic foundation

Most of the work presented in this thesis is unpublished and it is an extension of my M. Eng. thesis (in Italian): Daniele Canavese. «Generazione automatica di codice crittografico ottimizzato». M. Eng. thesis. Politecnico di Torino, 2010.

The following published papers instead represent the basis of Chapter 3:

- Daniele Canavese et al. «Can a Light Typing Discipline Be Compatible with an Efficient Implementation of Finite Fields Inversion?» In: *FOPARA 2013, proceedings of the 3rd International Workshop on Foundational and Practical Aspects of Resource Analysis*. Bertinoro (Italy), 2014. DOI: `10.1007/978-3-319-12466-7_3`;

- Daniele Canavese et al. «Light combinators for finite fields arithmetic». In: *Science of Computer Programming* 111 (3 2015), pp. 365–394. DOI: `10.1016/j.scico.2015.04.001`.

Thesis organization

This dissertation is structured as follows.

Chapter 1 introduces some real-world scenarios were the use of high speed cryptography should be, or was, beneficial and the related works. For the sake of completeness, it reports also some interesting works and publications non directly related to the ECC.

Chapter 2 presents the mathematical theory behind the elliptic curves and their algorithms. Its content is mainly taken from *Handbook of Elliptic and Hyperelliptic Curve Cryptography* [24] and *Guide to Elliptic Curve Cryptography* [25]. It can be freely skipped if the reader is already familiar with these concepts.

Chapter 3 is basically a reworked version of the papers «Can a Light Typing Discipline Be Compatible with an Efficient Implementation of Finite Fields Inversion?» [18] and «Light combinators for finite fields arithmetic» [19]. It deals with the mathematics behind the create of the DCEA method, an original algorithm for computing a multiplicative inverse in a binary field. It can be freely skipped if the reader is not interested in its internals.

Chapter 4 contains the architectural overview of the compilation framework. It provides a brief description of its main components and their internal structure.

Chapter 5 describes the DSLs for implementing a generic cryptographic primitive, their features and instruction sets. In addition, their internal representation in the compiler is also introduced.

Chapter 6 has its roots in the compiler theory. It contains the list of analysis, transformation and translation passes supported by the framework for optimizing the code performance. Furthermore, it presents also a simple, but complete optimization example on a real ECC function.

Chapter 7 provides the experimental results. It reports and comments the times obtained by the generated code on two different hardware platforms (an Intel i7 and an ARMv7) and compares the results against two cryptographic libraries (OpenSSL and Crypto++) and other kind of curves.

Appendix A reports the proof rules for the logic families introduced in Chapter 3.

Appendix B contains the grammars for HRL and aXiom, the two DSLs described in Chapter 5.

Appendix C presents all the annotations supported by the aforementioned languages.

Appendix D provides several additional experimental results.

Acknowledgments

First of all, I would like to thank prof. Antonio Lioy for giving me the opportunity to work in the Computer and Network Security Group and working with such great people in all these years.

I would like to thank also Emanuele Cesena, which was very supportive during my M. Eng. thesis and during my first months in the security group.

Many special thanks to Emanuele (again), Rachid Ouchary, Marco Pedicini and Luca Roversi for letting me work with them and learn new interesting stuff about the Implicit Computational Complexity (ICC). Chapter 3 would not exist without them, making this thesis a bit less long but much less interesting.

Finally, I want to thank Cataldo (Aldo) Basile for reviewing my thesis even if not being my official co-supervisor. Beyond the call of duty...

Chapter 1

High speed cryptography

There are two types of encryption: one that will prevent your sister from reading your diary and one that will prevent your government.

BRUCE SCHNEIER

This chapter is devoted to exploring the current front-line of high speed cryptography. It is split into two parts. The first one lists a series of real world scenarios where the application of high speed cryptography has been or will be beneficial, while the second part contains a selection of related papers, documents and reports.

1.1 Real world applications

This section presents some interesting examples of real world scenarios where *encrypting faster* was particularly advantageous and/or where we are still waiting this to happen.

1.1.1 Domain name system security

The DNS is a hierarchical system for automatically associating domain names with various other kinds of information, most prominently their IP addresses. It provides the Internet a vital infrastructure for its correct operation, so that its security is a fundamental prerequisite for our modern interconnected world. This case is one of the most conspicuous examples where the cryptographic speed played, and still plays, a pivotal role.

Figure 1.1 shows the time-line of the DNS security.

The origin of the DNS dates back to the ARPANET network, but the first modern ideas of its implementation appeared in 1983 [26, 27]. The initial specifications of the DNS infrastructure did not include any form of security since the Internet, at the time, was still in its infancy. This started, however, to be a severe issue, especially after the growth of the Internet in the nineties. Without any kind of protection, an attacker could easily inject fake records into a resolver's cache in order to divert the traffic somewhere else. These attacks are collectively known as *DNS spoofing* or *DNS cache poisoning* attacks.

The first serious attempt to solve these problems arrived only in 1999, when DNSSEC [28] was proposed. The idea behind this improved system is that each answer from a DNSSEC protected zone is digitally signed via a DNS server's key and, by verifying such signature, a resolver is able to

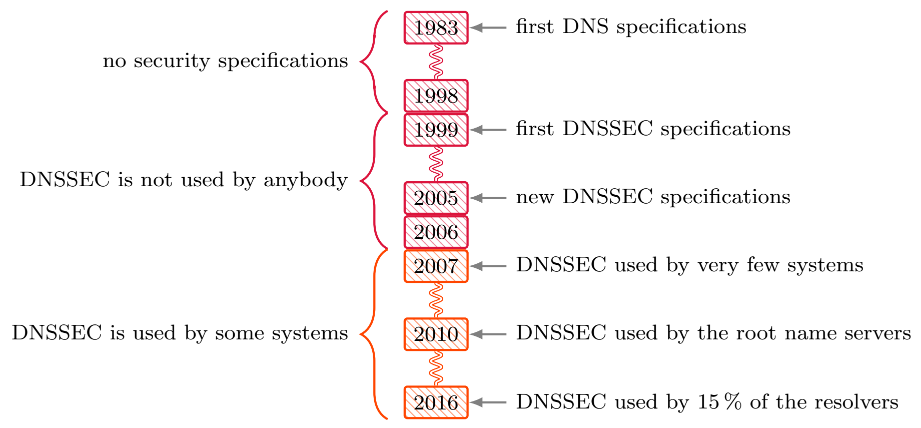

no security specifications
DNSSEC is not used by anybody
DNSSEC is used by some systems
1983
1998
1999
2005
2006
2007
2010
2016
first DNS specifications
first DNSSEC specifications
new DNSSEC specifications
DNSSEC used by very few systems
DNSSEC used by the root name servers
DNSSEC used by 15 % of the resolvers

Figure 1.1: DNS security time-line.

check its authenticity. Note that DNSSEC offers only message authentication and does not provide confidentiality of data. In a few months, it quickly became evident that this solution was not able to scale for large networks and the whole Internet. Eventually, no DNS server actually implemented such system. This was mainly due to the latency added by signing the answer messages and the increased number of the exchanged packets.

In 2005, the IETF modified the original DNSSEC protocol to overcome some scalability issues [29, 30, 31]. The main difference was the management of a child server's public key update. In the original protocol, when a child server changes its public key it must send to its parent server all its records, then the parent server signs them with its private key and send them back to the child. In the new version, when a child server updates its public key, it has only to send it, conveniently signed, to its parent server, which stores it in a special record, called a Delegation Signer resource record (DS). This change is supposed not to decrease the security of the system [32] and has the advantage of requiring less encryption and network traffic. However, its drawback is that the resolvers must perform a double signature verification (the child server's answer signature and the parent server's DS) instead of only one as in the original version.

In 2007, a very small number of name servers and resolvers became DNSSEC-capable, such as TDC, a Danish ISP provider [33]. However, it took another three years of technological advancement to see the first worldwide real applications of DNSSEC. After several months of testing, it was successfully deployed to all the thirteen root name servers in 2010 [34], but nothing more. After six years, in 2016, there are currently 1252 top level domains and 1090 are signed [35], but only about 15 % of the resolvers is DNSSEC-ready [36].

DNSCurve [37] is a proposed security extension for the DNS by Bernstein and an alternative to DNSSEC. The latter system leverages RSA [29] and uses large per-recordset signatures. On the other hand, DNSCurve exploits elliptic curves on 256 bit and performs smaller per-packet encryption and authentication. In 2010, OpenDNS, one of the biggest world-wide companies offering DNS resolution services, started to use DNSCurve [38]. This technology however has not yet been officially adopted by the IETF.

Thirty-three years have passed after the DNS initial specifications, but its security level is still far from being adequate. The lack of fast (and cheap) software and hardware cryptographic implementations has surely influenced the adoption's delay of a trusted domain name system, and still does.

1.1.2 Google encrypted search

If an attacker eavesdrops a communication between a victim and a web search engine, he can accesses several private and (probably) unwanted information about the target individual. These might include the victim's interests, hobbies, political tendencies, sexual preferences and so on. Confidentiality via encryption can provide a safe barrier for securing the users' privacy. This, however, poses several technical challenges such as implementing a fast cryptographic system.

Google Inc. is one of the biggest technology companies in the world. It was founded in 1998 by Larry Page and Sergey Brin. The most widely used and known product that Google offers is its search engine known as *Google Search*, launched in 1997.

Figure 1.2 sketches the Google encrypted search time-line.

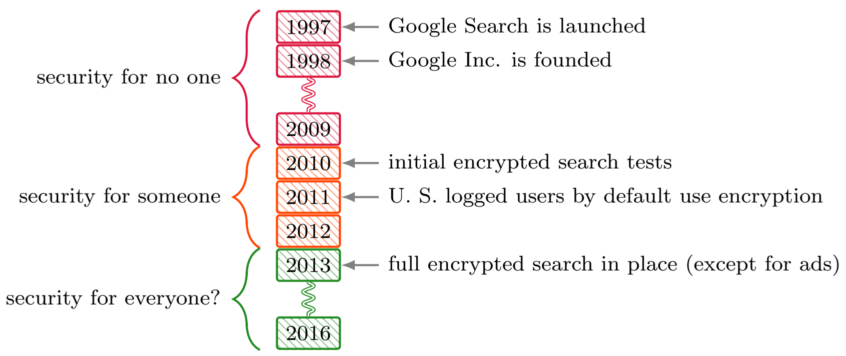

security for no one
security for someone
security for everyone?
1997
1998
2009
2010
2011
2012
2013
2016
Google Search is launched
Google Inc. is founded
initial encrypted search tests
U. S. logged users by default use encryption
full encrypted search in place (except for ads)

Figure 1.2: Google encrypted search time-line.

Up to the early months of the 2010 every search conducted via the Google engine (accessible through the URL `http://www.google.com/`) was using the traditional HTTP protocol, without any kind of protection or security mechanism. This allowed anybody to easily intercept a query by means of a sniffing tool, without recurring to cryptanalytic attacks.

In May 2010 Google started to test a secure search via the HTTPS protocol using a specialized URL: `https://www.google.com/`, later changed to `https://encrypted.google.com/` [39].

One year later, in 2011, Google started to automatically redirect all the U. S. users logged in their account to the encrypted search engine by default [40].

At the end of 2013 Google decided to (silently) moving all the web search queries to HTTPS except for the clicks on advertisements [41].

Interestingly, up to 2013 Google used RSA-1024 in their certificates, due to performance reasons. Only at the end of that year they completely switched their systems to make use of RSA-2048 [42]. In 2003, Shamir and Tromer [43] showed that RSA-1024 can be theoretically broken in about one year by making use of a hypothetical custom device costing about 10 million of dollars.

Table 1.1 shows the supposed TLS 1.2 cipher-suites priorities for `https://www.google.com/` in 2016. The list was populated via a scan performed with the *`cipherscan`* tool[1]. Interestingly, the first 4 preferred cipher-suites are ECC-based and all the elliptic curve algorithms works on the P-256 curve, whose security is debated [44]. The cipher-suites are obviously not in a strictly decreasing security order, but some speed constraints were take into account to sort them.

[1]*`cipherscan`* can be freely downloaded from `https://github.com/jvehent/cipherscan`.

priority	cipher-suite
1	ECDHE(P-256)-RSA-2048 + CHACHA20-POLY1305 + SHA-256
2	ECDHE(P-256)-RSA-2048 + AES-128-CBC + SHA-256
3	ECDHE(P-256)-RSA-2048 + AES-128-CBC + SHA-1
4	ECDHE(P-256)-RSA-2048 + RC4 + SHA-1
5	RSA-2048 + AES-128-GCM + SHA-256
6	RSA-2048 + AES-128-CBC + SHA-1
7	RSA-2048 + AES-128-CBC + SHA-256
8	RSA-2048 + 3DES-CBC + SHA-1
9	RSA-2048 + RC4 + SHA-1
10	RSA-2048 + RC4 + MD5
11	ECDHE(P-256)-RSA-2048 + AES-256-GCM + SHA-384
12	ECDHE(P-256)-RSA-2048 + AES-256-CBC + SHA-1
13	ECDHE(P-256)-RSA-2048 + AES-256-CBC + SHA-384
14	ECDHE(P-256)-RSA-2048 + AES-128-CBC + SHA-256
15	RSA-2048 + AES-256-GCM + SHA-384
16	RSA-2048 + AES-256-CBC + SHA-1
17	RSA-2048 + AES-256-CBC + SHA-256

Table 1.1: Supposed TLS 1.2 cipher-suite priorities for Google Search in 2016.

1.1.3 Mobile phone protection

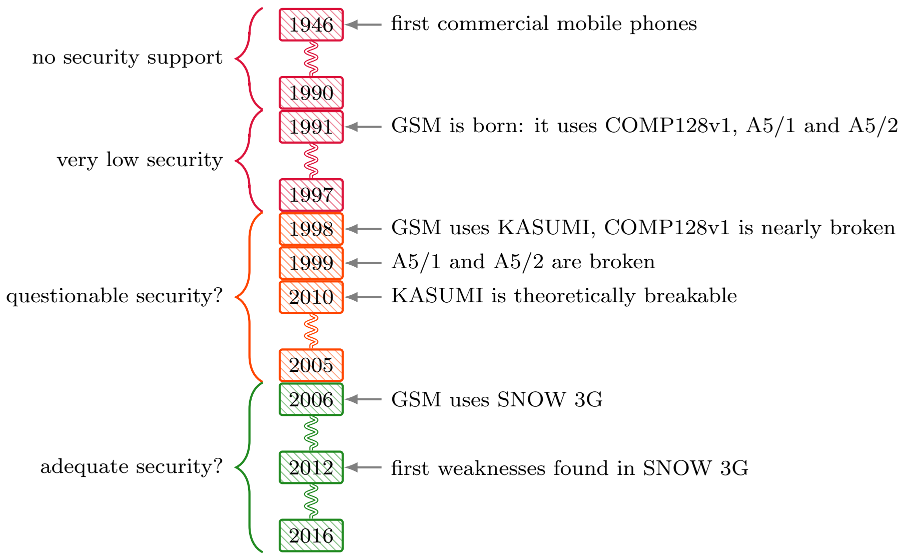
Telephony was one of the biggest technological revolutions achieved by the mankind. Intrinsically pervasive, it became much more ubiquitous when the first cellular networks eventually started to be available to the public. Ensuring a secure communication channel over the air is important for several reasons. Intercepting other's private calls can disclose not only personal matters, but also sensitive information such as business-related discussions. This necessity for security is further exacerbated in modern times, where a user can buy products and browse the dangerous Internet directly with his smart-phone .

In this scenario, cryptography is a key element since its plays a twofold role. First, it should be used to authenticate users (or, better, their devices) in the network. Second, it should be used to ensure the confidentiality of the traffic. Modern mobile phone operators have to cope with thousands of transmissions per second and they must deal with tebibytes per second bandwidths. All these exchanged information must be adequately protected and fast encryption is mandatory to be able to keep up with such enormous data transfers. History, however, tells that not always the right choices were made when dealing with the troublesome speed-security trade-off.

Figure 1.3 shows the time-line of the security in the mobile phone world.

The first mobile telephone technology was completely analog in nature and without any kind of protection. Just after the Second World War several initial commercial attempts were made, such as the Mobile Telephone Service (MTS) launched in 1946 in the United Stated and operated by Bell Systems and Motorola.

The year 1991 was a milestone in the communication area: the GSM [45] system was born. GSM introduced the use of SIMs, small integrated circuits used to securely store contacts, messages and other sensitive information. A SIM card contains a 128 bit key known as the *authentication key* used to perform various operations. These primarily include the user authentication within the network (performed through a challenge protocol) and generating shared keys for securing the voice calls via symmetric encryption. In this context, the authentication key effectively has the role of a pre-shared secret between the mobile device and the operator, allowing the GSM to completely



no security support
very low security
questionable security?
adequate security?
1946
1990
1991
1997
1998
1999
2010
2005
2006
2012
2016
first commercial mobile phones
GSM is born: it uses COMP128v1, A5/1 and A5/2
GSM uses KASUMI, COMP128v1 is nearly broken
A5/1 and A5/2 are broken
KASUMI is theoretically breakable
GSM uses SNOW 3G
first weaknesses found in SNOW 3G

Figure 1.3: Mobile phone security time-line.

avoid the use of public key algorithms. This choice was done for a number of reasons, most notably because asymmetric encryption is notoriously slow and complex to implement in hardware and software. Unfortunately this lead to some very bad architectural decisions. Most of the SIM manufacturers chose to implement the COMP128v1 algorithm (in hardware) used in the tasks of the user authentication and key exchange. COMP128v1 receives in input the 128 bit authentication key and another 128 bit random number to produce two outputs: a 32 bit and a 64 bit values. The algorithm code was confidential and it was never officially published, however, in 1998 it was fully reconstructed via reverse engineering[2]. Note that this approach completely violates the Kerckhoff's principle. The same year, shortly after the COMP128v1 reconstruction, Wagner, Goldberg, and Briceno [46] showed how to mount a chosen-challenge attack on COMP128v1 and extract the authentication key in less than 8 hours, allowing to fully clone a SIM. The symmetric ciphers used in the original GSM specifications were A5/1 and its faster (and weaker) brother A5/2. In 1999, Biryukov, Shamir, and Wagner [47] cryptanalyzed A5/1 in real-time by recovering the key in one second after listening only two minutes of cipher-text. In the same year, Goldberg, Wagner, and Green [48] made use of a similar approach to break A5/2.

In 1998, the GSM standard was upgraded and new encryption algorithms were introduced. New versions of COMP128 were proposed and, as usual, never officially published, but in 2013 they were reverse engineered and partially recovered[3]. The new stream cipher A5/3, nicknamed KASUMI, was also introduced. It is a faster (and significantly weaker) version of another cryptographic algorithm known as MISTY1 [49]. In 2010, Dunkelman, Keller, and Shamir [50] proved that KASUMI can be broken in two hours with a simple PC. The same attack does not affect MISTY1.

In 2006, the ETSI included in the latest GSM specifications the support for a new cipher, SNOW 3G [51]. In 2012, Kircanski and Youssefr [52] showed that related-key attacks are possible, questioning the real SNOW 3G security.

Even if the most recent GSM updates mitigate some attack types, about 52 % of the connections

[2]The code of COMP128v1 is available at http://www.scard.org/gsm/a3a8.txt.

[3]The code of COMP128v2 is available at http://www.hackingprojects.net/2013/04/secrets-of-sim.html.

in 2015 was through a 2G technology [53], that is they were still making use of the oldest GSM encryption algorithms such as A5/1 and A5/2.

In the current literature there exist a number of publications proposing how to include fast public key systems, and ECC in particular, into the GSM to increase its security [54, 55]. Their suggestions, however, still remain on paper.

1.2 Related works

The following paragraphs introduce the literature related to the project presented in this dissertation and the world of elliptic curve cryptography.

1.2.1 Cryptography-aware DSLs and compilers

In the last years, the research has produced very few works on how to automatically generate cryptographic code. This section splits the current literature in three parts: automatic generation of cryptographic functions in software (this dissertation's area of interest), in hardware and automatic generation and verification of cryptographic protocols. Although the last two arguments are out of scope w.r.t. to this thesis' investigations, they are still mentioned since they offer several innovative ideas.

Cryptographic functions in software

The scientific literature has a surprising low number of dissertations on the topic of producing efficient cryptographic function implementations purely in software.

In 2003, Abdelrahman, Talkhan, and Shaheen [56] designed a language named CL, an acronym of Cryptography Language, allowing a developer to describe very low level primitives as the ones for the block ciphers. The authors also developed a compilation tool-chain that supports multiple back-ends, claiming that they can generate different implementations both in C and VHDL starting from CL. Differently from the compiler described in this thesis, the CL one does not perform any kind of optimization and supports only low level operations.

CAO, which stands for C And Occam, is a DSL proposed in 2005 by Barbosa et al. [17]. It comes with an ad-hoc compiler able to produce optimized C code. It was used in the CACE FP7 European Project [23], aimed to facilitate the development of hardware and software encryption products. CAO is an imperative language created for describing high level cryptographic primitives such as operations on elliptic curves, but it lacks the constructs needed to develop low level routines (such as the arithmetic operations on fields). Differently, the languages described in Chapter 5 have a wider range of applicability, allowing to implement both high level and low level primitives. In addition, the CAO compiler implements only a very limited subset of the optimization techniques performed by this thesis' framework (for instance it cannot make use of assembler instructions and perform bit vector optimizations).

Another interesting cryptographic DSL is qhasm and its related tools. It was initially presented by Bernstein [57] in a talk at the Fast Software Encryption 2005 conference[4]. It is a sort of hardware independent assembly language that allows to use most of the instructions available in the traditional assembler languages, but with a more readable syntax and without directly referencing the machine registers. It was recently complemented with a lexical preprocessor called maq by Schwabe [58]. It bears several similarities to the HRL language (Section 5.1.1), but, differently,

[4]The slides are available at `https://cr.yp.to/talks/2005.02.21-2/slides.pdf`.

qhasm cannot handle high level types such as bit vectors and completely lacks the ability to call other functions. This severely hinder its applicability to higher level cryptographic primitives.

In 2010, Perez and Scott [59] designed a cryptographic compiler for pairing-based cryptography. The tool can automatically choose the most appropriate pairing function of an elliptic curve and generate its code in C. This is an interesting development, although not directly related to generating encryption primitives.

Cryptographic functions in hardware

A number of interesting papers present how to produce optimized hardware description language implementations for manufacturing cryptographic primitives directly on integrated circuits.

In 1997, Ananian [60] proposed an optimizing compiler for the Tiger programming language, a DSL introduced by Appel [61]. This tool can translate and optimize the code into behavioral VHDL by applying a quite limited number of code improving techniques.

Cryptol is a language for describing cryptographic algorithms whose first documentations appeared in 2003 and it is currently owned by Galois Inc. [16]. It was developed with the participation of the NSA and it was originally designed for expressing primitives that can be efficiently implemented in hardware. It is a purely functional language and as such does not directly support looping instructions, which can only simulated via recursion. Differently, the languages presented in Chapter 5, are imperative, support loops in various forms and they were created for developing efficient implementations in software.

Cryptographic protocols

Several recent studies investigate how to produce secure cryptographic protocols and mainly focus on how to formally prove their correctness rather than improving their efficiency.

In 2005, Lucks, Schmoigl, and Tatl [62] proposed both a high level language to describe cryptographic protocols and a compiler than can generate Java code implementations. Their framework performs only the translation, without any type of optimizations nor security verifications.

Bhargavan et al. [63] in 2008 developed a compiler for generating cryptographic protocol implementations starting from high level multi-party sessions. The tool can produce the final code in ML, a general purpose functional programming language. In addition, the compiler is also able to prove the security of the protocol via a special typing system for ML.

Bangerter et al. [64] introduced in 2011 cPLC, a language that is able to describe virtually any kind of cryptographic protocol. The authors also developed a related compiler that can produce well structured and documented C and Java code, but without any form of optimizations nor proof of its security properties.

ZKCrypt is an optimizing cryptographic compiler developed in 2012 by Almeida et al. [65]. It supports only zero knowledge protocols [66], but it is able to verify and prove a number of security properties about its specifications. It can also generate optimized code in C or Java. The authors do not detail the optimization algorithms, but suggests that at least algebraic simplifications are supported.

1.2.2 Elliptic curve cryptography

Elliptic Curve Cryptography is quite diffused nowadays, so this section presents only a small but interesting selection of ECC related publications.

History

In 1985, Miller [6] and Koblitz [7] independently proposed to use the group of rational points defined by an elliptic curve as a way to build a public key crypto-system. They both identified two main advantages over more traditional asymmetric algorithms such as RSA. First, a greater flexibility in the choice of the group and, second, the absence of a sub-exponential algorithm to break an elliptic curve (if some proper parameters are chosen).

In 1992, Rivest et al. [67] presented ECDSA, the elliptic curve version of DSA. This was in a response to the NIST's request for public comments on their first proposal for their Digital Signature Standard (DSS).

In 1995, Schoof [68] published three different algorithms for computing the number of points in an elliptic curve defined over a finite field. This important result greatly helped to efficiently build elliptic curves suitable for cryptographic applications.

In 1996, Boneh and Lipton [69] proved several important properties of ECDH, the elliptic curve version of DH. In particular, they showed that if no sub-exponential algorithm exists for breaking the ECDLP, then no sub-exponential algorithm also exists for breaking the ECDHP (see Section 2.1.4 for further information).

In 1997, the Certicom[5], a U. S. cryptography company now acquired by BlackBerry Limited, started the ECC challenge[6]. They challenged the public to break some elliptic curves offering a prize up to 100 000 $. The challenges are split in level I and II. The latter are believed to be computationally infeasible and all of them are still unsolved.

In 1998, Law et al. [70] introduced ECMQV, a new key agreement algorithm using elliptic curves and based on the Diffie-Hellman scheme.

Respectively in 2002 and 2004 the ECC challenges for a 109 bit curve on a prime and a binary field were solved[6]. They are the biggest factorized elliptic curves so far.

In 2005, the NSA presented its strategy for securing top secret data and recommended the use of ECC for key agreement and digital signatures [71].

In 2014, Feo, Jao, and Plût [14] proposed the Supersingular Isogeny Diffie-Hellman (SIDH), a quantum-resistant elliptic curve variation of DH based on the difficulty of finding isogenies between supersingular elliptic curves.

Standards

There are a number of standards, best practices and recommendations regarding ECC. A few of them also suggest some specific elliptic curves and fields.

In 2000, the IEEE published the *IEEE Standard Specifications for Public-Key Cryptography* [72]. It proposes several key agreement and digital signature schemes based on ECC. In addition, it gives several advices on how to choose the curve parameters and how to handle the keys.

In 2005, the ANSI produced the *ANSI X9.62:2005, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)* [73]. It describes how ECDSA should be implemented and contains some suggestions about the elliptic curve domain parameters. In the same year, the German company ECC Brainpool introduced the *ECC Brainpool Standard Curves and Curve Generation* [74]. In this document they recommend how to generate strong pseudo-random elliptic curves and give the parameters for a set of 14 suggested curves on prime fields.

[5]See https://www.certicom.com/.

[6]More information are available at https://www.certicom.com/index.php/the-certicom-ecc-challenge.

In 2006, Bernstein [75] proposed the Curve25519 defined on a prime field with order $2^{255} - 19$. Though not an official standard, this curve gained some popularity, especially in open source projects.

In 2010, Certicom Research presented the *SEC 2: Recommended Elliptic Curve Domain Parameters* [76], a document describing a set of 20 recommended elliptic curves (8 on prime fields and 12 on binary fields).

In 2011, the ANSI proposed the *ANSI X9.62:2011, Public Key Cryptography for the Financial Services Industry - Key Agreement and Key Transport Using Elliptic Curve Cryptography* standard [77]. It details how various ECC key agreement and transport algorithms should be implemented and suggests some selection criteria for the elliptic curve domain parameters.

In 2013, the NIST proposed the *FIPS PUB 186-4 – Digital Signature Standard (DSS)* [13]. It contains a series of recommendations about how to implement a number of algorithms for performing digital signatures such as DSA, RSA and ECDSA. In addition, it gives some advices on how to generate the curve parameters and suggests 15 elliptic curves (5 on prime fields and 10 on binary fields).

Applications

One of the most important application of ECC is in a number of data protection protocols such as IPsec [10], TLS [11] and SSH [12] which implement several elliptic curve base techniques. Table 1.2 lists the supported ECC algorithms in these protocols.

protocol	ECDH	ECDSA	ECMQV
IPsec	✓	✓	○
TLS	✓	✓	○
SSH	✓	✓	✓

Table 1.2: Protocol supports of the ECC algorithms.

A Wireless Sensor Network (WSN) is a distributed system constituted by small network nodes, known as *motes*, containing a battery, some processing hardware and a sensor. WSNs are used to continuously monitor physical conditions in an environment and their usage is quickly spreading in several areas such as industrial machinery control and building structural health monitoring. Elliptic Curve Cryptography has started becoming a popular alternative to other conventional asymmetric algorithm due to its speed and flexibility. Several publications proposed key exchange and digital signature schemes based on elliptic curves, customized for this environments. For instance, in 2008, the TinyECC library was introduced by Liu and Ning [78] and quickly became well-known in the WSN field due to its performance and flexibility. More recently, in 2013, Kodali et al. [79] presented a novel method for performing the scalar multiplication on an elliptic curve specialized for the WSN's platforms. There are also some hardware proposals, such as the one described by Ahmed et al. [80], which presents a FPGA implementation of a 160 bit ECC processor with excellent area results for a mote.

Similarly, a Mobile Ad-hoc NETwork (MANET) is an infrastructure-less network of wireless mobile devices that automatically configure themselves. MANETs have a wide area of applications, from vehicle communications, where are known as VANETs to smart-phones. In 2010, Dahshan and Irvine [81] proposed a fast key distribution method based on the elliptic curve discrete logarithm problem for low power and processing constrained devices such as the MANET nodes. Manvi, Kakkasageri, and Adiga [82] presented the use of ECDSA in VANETs to perform high speed message authentication between vehicles. Huang, Yeh, and Chien [83] presented an original ECC approach for anonymous batch authentication and key agreement in vehicular networks, showing that their method provides less delays and message losses w.r.t. to the more common ECDSA.

1.2.3 Other high speed cryptographic techniques

Elliptic curves are not the only alternative to traditional public key algorithms. This section reports some other techniques for the sake of completeness.

In 1978, McEliece [84] developed a crypto-system based on the hardness of decoding a general linear code. Even if it never gained a practical use in the real world, recently it received some focus since it is very resilient to quantum computing attacks. Even if it is faster than RSA, the algorithm's main disadvantage is that its key size is much larger. In 2008, Bernstein, Lange, and Peters [85] suggested a public key size of 1 537 536 bit (about 187.7 KiB) for coupling the McEliece method with a symmetric cipher on 128 bit.

In 1989, Koblitz [20] extended the same basic idea behind the elliptic curve cryptography to the rational points over an *hyperelliptic curve*. This algebraic structures allow even smaller key sizes w.r.t. ECC since they are theoretically harder to break, but so far no protocol or standard implements or regulates them.

In 1998 and later in 2001, Frey [21, 22] proposed to use *trace zero varieties* over elliptic curves in cryptography. They are sub-groups of the rational points defined over an elliptic curve with some peculiar properties that allow them to achieve better performance w.r.t. the traditional elliptic curves. Trace zero varieties have still to receive an official standardization.

In 2001, Patarin [86] proposed the *hidden field equations* as a public key crypto-system having a key size comparable to the ECC's ones. They make use of different polynomials on finite extension fields and are based on the hardness of finding the solutions to a system of multivariate quadratic equations. The 'hidden' adjective comes from the fact that the algorithm tries to hide the extension fields and their polynomials by making use of private affine transformations.

Chapter 2

Elliptic curve cryptography

In mathematics you don't understand things. You just get used to them.

JOHANN VON NEUMANN

Elliptic curves are known to the mathematicians for over a hundred years and have been used to solve a wide array of problems, such as proving the Fermat's Last Theorem [87]. One of their most recent and prominent roles is in the cryptography sector, where they are used to design strong and efficient public-key cryptographic systems.

This chapter introduces both the mathematical and algorithmic background of ECC. In order to keep the discussion short, all the theorem's proofs are omitted. The core of the following sections is mostly taken from the following sources:

- Roberto Avanzi et al. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman & Hall/CRC, 2005. ISBN: 1584885181;
- Darrel Hankerson, Alfred John Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2003. ISBN: 038795273X;
- Kenny Fong et al. «Field Inversion and Point Halving Revisited». In: *IEEE Transactions on Computers* 53 (8 2004), pp. 1047–1059. DOI: 10.1109/TC.2004.43;
- Daniele Canavese et al. «Can a Light Typing Discipline Be Compatible with an Efficient Implementation of Finite Fields Inversion?» In: *FOPARA 2013, proceedings of the 3rd International Workshop on Foundational and Practical Aspects of Resource Analysis.* Bertinoro (Italy), 2014. DOI: 10.1007/978-3-319-12466-7_3.

2.1 Mathematical background

This section is structured in a progressive way, starting with the most basic notions and culminating with the discussion of a complete cryptographic protocol, that is ECDH.

2.1.1 Groups

Groups constitute not only the basis for building a fully working ECC system, but they have also a pivotal role in most of the current cryptographic algorithms.

Definition (Group)**.** *A* group $(\mathbb{G}, \circ)$ *is a set* $\mathbb{G}$ *coupled with a binary operation* $\circ \colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}$*, called the* group operation*, which satisfies the following properties:*

- *the* closure*, that is* $a \circ b \in \mathbb{G},\ \forall a, b \in \mathbb{G}$*;*
- *the* associativity*, that is* $(a \circ b) \circ c = a \circ (b \circ c),\ \forall a, b, c \in \mathbb{G}$*;*
- *the* existence of an identity*, that is* $\exists e \in \mathbb{G} : e \circ a = a \circ e = a,\ \forall a \in \mathbb{G}$*;*
- *the* existence of inverses*, that is* $\exists a^{-1} \in \mathbb{G} : a \circ a^{-1} = a^{-1} \circ a = e,\ \forall a \in \mathbb{G}$*.*

Definition (Abelian group)**.** *A* group $(\mathbb{G}, \circ)$ *is an* Abelian group *(or* commutative group*) if the group operator is* commutative*, that is if* $a \circ b = b \circ a,\ \forall a, b \in \mathbb{G}$*.*

From now on, each group that will be discussed is assumed to be Abelian.

Example. The natural numbers $\mathbb{N}$ and the usual addition operation + do not form a group, while $(\mathbb{Z}, +)$, $(\mathbb{Q}, +)$, $(\mathbb{R}, +)$ and $(\mathbb{C}, +)$ are all valid (Abelian) groups.

Definition (Order)**.** *The* order *of a group* $\mathbb{G}$ *is the number of elements that contains and its is denoted by* $|\mathbb{G}|$*.*

Definition (Finite group)**.** *A group* $(\mathbb{G}, \circ)$ *is a* finite group *if* $|\mathbb{G}| < \infty$*.*

Finite groups are the most interesting ones in the computer science since they can be represented and stored in a machine's memory.

2.1.2 Fields

The notion of field is relatively trivial and can be though as an extension of a group.

Definition (Field)**.** *A* field $(\mathbb{F}, \circ, \square)$ *is a set* $\mathbb{F}$ *coupled with two binary operations* $\circ \colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ *and* $\square \colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$*, called the* field operations*, such that:*

- $(\mathbb{F}, \circ)$ *is an abelian group with the identity element denoted by* e*;*
- $(\mathbb{F} \setminus \{ e \}, \square)$ *is an abelian group;*
- *the* distributivity *holds, that is* $(a \circ b) \square c = (a \circ b) \square (b \circ c),\ \forall a, b, c \in \mathbb{F}$*.*

In order to simplify the formulas, and by abuse of notation, the symbol $\mathbb{F}$ will be used to specify a field having the addition and the multiplication as its operations, i.e. $(\mathbb{F}, +, \cdot)$. In $\mathbb{F}$ we can then define:

- the element $0 \in \mathbb{F}$ as the additive identity;
- the element $1 \in \mathbb{F}$ as the multiplicative identity;
- the *additive inverse* of a, denoted by $-a$, such that $a + (-a) = 0,\ \forall a \in \mathbb{F}$;
- the *multiplicative inverse* of a, denoted by a^{-1}, such that $a \cdot a^{-1} = 1,\ \forall a \in \mathbb{F} \setminus \{ 0 \}$;
- the *subtraction* of a and b as $a - b = a + (-b),\ \forall a, b \in \mathbb{F}$;
- the *division* of a and b as $a/b = a \cdot b^{-1},\ \forall a \in \mathbb{F} \land \forall b \in \mathbb{F} \setminus \{ 0 \}$.

Definition (Order)**.** *The* order *of a field* $\mathbb{F}$ *is the number of elements that contains and it is denoted by* $|\mathbb{F}|$*.*

Definition (Finite field)**.** *A field* $\mathbb{F}$ *is a* finite field *if* $|\mathbb{F}| < \infty$.

From now on, the notation $\mathbb{F}_p$ will be used to denote a finite field with order p whose operations are the addition and multiplication.

Definition (Characteristic)**.** *The* characteristic *of a field* $\mathbb{F}$, *denoted by* $\mathrm{char}(\mathbb{F})$, *is zero if the field is not finite, otherwise it is the smallest positive integer* n *such that:*

$$\underbrace{1 + \ldots + 1}_{n \text{ addends}} = n \cdot 1 = 0.$$

Proposition. *The characteristic of any finite field is prime.*

An interesting fact is that the smallest finite field is $\mathbb{F}_2$, containing only 0 and 1, that are the two identity elements. In $\mathbb{F}_2$ the addition operation is the exclusive or, so that $1 + 1 = 1 \oplus 1 = 0$. From this it follows that $\mathrm{char}(\mathbb{F}_2) = 2$.

Definition (Field homomorphism)**.** *Let be* $\mathbb{F}$ *and* $\mathbb{F}'$ *two fields. A* field homomorphism $\psi : \mathbb{F} \to \mathbb{F}'$ *is an application that:*

- $\psi(a + b) = \psi(a) + \psi(b), \ \forall a, b \in \mathbb{F}$;
- $\psi(a \cdot b) = \psi(a) \cdot \psi(b), \ \forall a, b \in \mathbb{F}$;
- $\psi(1) = 1$.

Definition (Extension field)**.** *Given two fields* $\mathbb{F}$ *and* $\mathbb{F}'$, *we say that* $\mathbb{F}'$ *is an* extension field *of* $\mathbb{F}$ *if there exists a field homomorphism* $\psi : \mathbb{F} \to \mathbb{F}'$. *This will be denoted as* $\mathbb{F}'/\mathbb{F}$.

Extension fields are an effective and simple way of building a bigger field from a smaller one. Frequently the base field is $\mathbb{F}_2$ since it represents a single bit, and its extensions an ordered sequence of bits.

Definition (Degree)**.** *Given an extension field* $\mathbb{F}'/\mathbb{F}$, *its dimension is called* degree *and it is denoted by* $\deg(\mathbb{F}'/\mathbb{F})$.

Example. Obviously $\mathbb{C}/\mathbb{R}$ since each complex number can be represented by means of two real numbers, hence $\deg(\mathbb{C}/\mathbb{R}) = 2$.

Definition (Binary field)**.** *A field with characteristic* 2 *is called a* binary field.

The notation $\mathbb{F}_{2^n}$ will denote a binary (finite) field of degree n coupled with the usual addition and multiplication operations. Obviously $\mathbb{F}_{2^n}/\mathbb{F}_2$.

Remark. An element $a \in \mathbb{F}_{2^n}$ can be represented as a polynomial in t with binary coefficients modulo an irreducible polynomial $m(t)$ of degree n, called the *modulus*. This will be denoted by $\mathbb{F}_{2^n}[t]/m(t)$.

Example. The AES cipher [89] internally make use of the finite field $\mathbb{F}_{2^8}[t]/t^8 + t^4 + t^3 + t + 1$.

2.1.3 Elliptic curves

Elliptic curves are planar curves built upon a field and their points create a group that follows some peculiar properties.

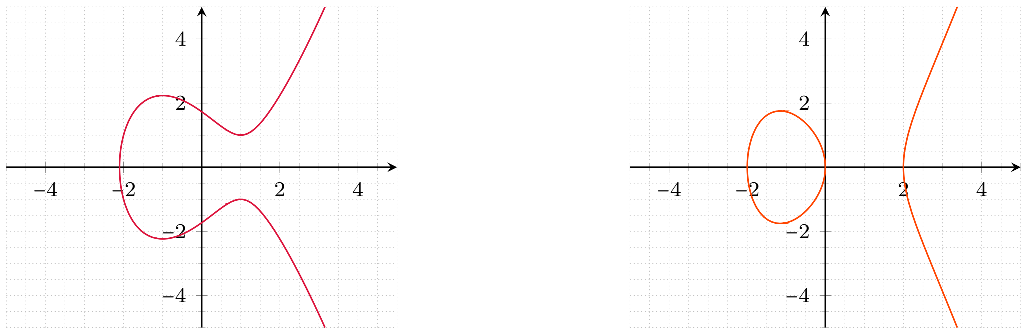
Definition (Rational point)**.** *Given a field* $\mathbb{F}$, *a* $\mathbb{F}$-rational point *is a point on an algebraic curve with all of its coordinates in* $\mathbb{F}$.

Definition (Elliptic curve)**.** *An elliptic curve $\mathcal{E}$ over a field $\mathbb{F}$ is denoted by $\mathcal{E}/\mathbb{F}$. It is a smooth curve with at least one $\mathbb{F}$-rational point and defined by the* Weierstraß equation*:*

$$\mathcal{E} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6. \tag{2.1}$$

Since an elliptic curve is smooth it has no points of self-intersection or cusps.

Example. Figure 2.1 depicts the plots of two elliptic curves over $\mathbb{R}$.



(a) Plot of $y^2 = x^3 - 3x + 3$.

(b) Plot of $y^2 = x^3 - 4x$.

Figure 2.1: Plots of two elliptic curves over $\mathbb{R}$.

Definition (Set of points)**.** *Given an elliptic curve $\mathcal{E}/\mathbb{F}$, the notation $\mathcal{E}(\mathbb{F})$ denote the set containing all the points laying on the curve $\mathcal{E}$.*

Definition (Point at infinity)**.** *The set of all the points of a curve $\mathcal{E}/\mathbb{F}$ is equal to:*

$$\mathcal{E}(\mathbb{F}) = \left\{ (x, y) \in \mathbb{F}^2 : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \right\} \cup \{ P_\infty \}.$$

The point P_∞ is called the point at infinity*.*

Intuitively the point at infinity is the point where an elliptic curve intersects an horizontal axis laying at infinity.

Proposition. *Let be $\mathcal{E}/\mathbb{F}$ an elliptic curve. Then, $(\mathcal{E}(\mathbb{F}), +)$ is a group and the operation $+$ is known as* point addition*. Given three points $P(x_P, y_P)$, $Q(x_Q, y_Q)$, $R(x_R, y_R)$ with $P \neq Q$, the following formulas hold:*

$$\begin{cases}
R = P + P_\infty & \Longrightarrow \begin{cases} x_R &= x_P \\ y_R &= y_P \end{cases} \\
R = -P & \Longrightarrow \begin{cases} x_R &= x_Q \\ y_R &= -y_Q - a_1x_Q - a_3 \end{cases} \\
R = P + P & \Longrightarrow \begin{cases} \lambda &= (3x_P^2 + 2a_2x_P + a_4 - a_1y_P)/(2y_P + a_1x_P + a_3) \\ x_R &= \lambda^2 + a_1\lambda - a_2 - 2x_P \\ y_R &= \lambda(x_P - x_R) - y_P - a_1x_R - a_3 \end{cases} \\
R = P + Q & \Longrightarrow \begin{cases} \lambda &= (y_P - y_Q)/(x_P - x_Q) \\ x_R &= \lambda^2 + a_1\lambda - a_2 - x_P - x_Q \\ y_R &= \lambda(x_P - x_R) - y_P - a_1x_R - a_3. \end{cases}
\end{cases}$$

The *point subtraction* can be easily defined as $P - Q = P + (-Q)$.

Remark. The point at infinity is the identity element for the point addition, that is given any elliptic curve $P\mathcal{E}/\mathbb{F}$, then:

$$P + P_\infty = P_\infty + P = P, \quad \forall P \in \mathcal{E}(\mathbb{F}).$$

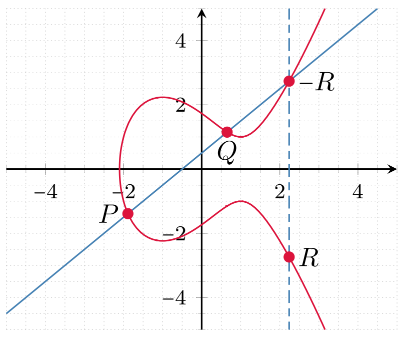
In order to keep the formulas compact, the notation $2P = P + P$ will be used to denote the addition of a point to itself. This operation is also known as *point doubling.*

The point addition and doubling have two simple geometrical explanations if the underlying field is $\mathbb{R}$, as depicted in Figure 2.2. The addition $R = P + Q$ can be performed graphically by:

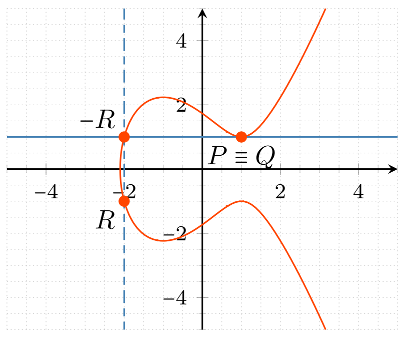
1. tracing the line passing through P and Q;
2. finding $-R$ as the third intersection point between the line and the elliptic curve;
3. finding R as the vertically opposite point of $-R$.

Analogously, $R = 2P$ can be found by:

1. tracing the elliptic curve tangent line passing through P;
2. finding $-R$ as the second intersection point between the line and the elliptic curve;
3. finding R as the vertically opposite point of $-R$.



(a) Graphical explanation of $R = P + Q$.



(b) Graphical explanation of $R = 2P$.

Figure 2.2: Point addition and doubling in $\mathcal{E}/\mathbb{R}$.

By generalizing the idea behind the point doubling, it can be defined the operation of *scalar multiplication*, also known as *point multiplication*, as:

$$n \cdot P = \underbrace{P + \ldots + P}_{n \text{ addends}}, \quad \forall P \in \mathcal{E}(\mathbb{F}) \wedge \forall n \in \mathbb{N}.$$

Note that, obviously, $0 \cdot P = P_\infty$.

In cryptography, the family of elliptic curves defined over the binary fields are of particular interest. This is mainly due to the fact that they can be implemented both in hardware and software in a very efficient way.

Definition (Admissible change of variables)**.** *Let $\mathcal{E}/\mathbb{F}$ be an elliptic curve defined by the Equation 2.1. Let be $u \in \mathbb{F} \setminus \{0\}$ and $r, s, t \in \mathbb{F}$. An* admissible change of variables *is defined by the mapping:*

$$\begin{cases} x & \mapsto u^2x' + r \\ y & \mapsto u^3y' + u^2sx' + t. \end{cases}$$

By making use of a proper admissible change of variables, the Weierstraß equation can be significantly simplified.

Definition (Supersingular elliptic curve)**.** *Let $\mathcal{E}/\mathbb{F}_{2^n}$ be an elliptic curve defined by the Equation 2.1 where $a_1 = 0$. The admissible change of variables $(x, y) \mapsto (x + a_2, y)$ transform the curve $\mathcal{E}$ into a* supersingular elliptic curve *defined by the equation:*

$$y^2 + a_3y = x^3 + a_4'x + a_6' \quad \textit{with } a_3 \neq 0.$$

Supersingular curves are not normally used in cryptography, so their group law is not further investigated in this context. Additional information about their security is given in Section 2.1.4.

Definition (Non-supersingular elliptic curve)**.** *Let $\mathcal{E}/\mathbb{F}_{2^n}$ be an elliptic curve defined by the Equation 2.1 where $a_1 \neq 0$. The admissible change of variables $(x, y) \mapsto (a_1^2x + a_3/a_1, a_1^3y + (a_1^2a_4 + a_3^2)/a_1^3)$ transform the curve $\mathcal{E}$ into a* non-supersingular elliptic curve *defined by the equation:*

$$y^2 + xy = x^3 + a_2'x^2 + a_6' \quad \textit{with } a_6' \neq 0.$$

The group law equations for a non-supersingular curve can then be simplified in:

$$\begin{cases} R = P + P_\infty & \Longrightarrow \begin{cases} x_R & = x_P \\ y_R & = y_P \end{cases} \\ R = -P & \Longrightarrow \begin{cases} x_R & = x_Q \\ y_R & = x_Q + y_Q \end{cases} \\ R = P + P & \Longrightarrow \begin{cases} \lambda & = x_P + y_P/x_P \\ x_R & = \lambda^2 + \lambda + a_2' \\ y_R & = \lambda(x_P + x_R) + x_R + y_P \end{cases} \\ R = P + Q & \Longrightarrow \begin{cases} \lambda & = (y_P + y_Q)/(x_P + x_Q) \\ x_R & = \lambda^2 + \lambda + x_P + x_Q + a_2' \\ y_R & = \lambda(x_P + x_R) + x_R + y_P. \end{cases} \end{cases}$$

2.1.4 Security of elliptic curve crypto-systems

In public key cryptography, each party possesses two keys: a public key and a private key. These two values are linked together by a relationship by means of a particular one-way function.

Definition (One-way function)**.** *Let Σ^* be the set of all the binary strings and f a function such that $f\colon \Sigma^* \to \Sigma^*$. We say that f is a* one-way function *if:*

- *f is injective;*
- *$f(x)$ is at most polynomially longer or shorter than x;*
- *$\forall x \in \Sigma^*$, $f(x)$ can be computed with* PTime *complexity;*

- *$\forall y \in \Sigma^*$, there is no* PTIME *algorithm that return either $x \in \Sigma^*$ such that $y = f(x)$ or 'error' if $y \notin \mathrm{Im}(f)$.*

In other words, a one-way function f has PTIME complexity, but its inverse f^{-1} must not have PTIME complexity. Chosen a private key k, a public key can be quickly computed with $K = f(k)$ and safely published, since retrieving the private key by computing $k = f^{-1}(K)$ is computationally unfeasible.

Interestingly, there are no proofs of any true one-way function's existence, since this would imply proving that PTIME ≠ NPTIME. This means that the security of a public-key crypto-system relies on the unproven hypothesis of the hardness of some computational problem. The elliptic curve scalar multiplication is supposed to be a one-way function, that is computing the product $Q = n \cdot P$ is easy, but retrieving n knowing P and Q is hard.

Definition (ECDLP)**.** *Let be $\mathcal{E}/\mathbb{F}$ an elliptic curve and $P, Q \in \mathcal{E}(\mathbb{F})$ two points. Finding the smallest integer n such that $Q = n \cdot P$ is called the* Elliptic Curve Discrete Logarithm Problem (ECDLP)*. The value n is known as the* discrete logarithm *of Q and it is denoted by $n = \log_P Q$.*

For historical reasons, the point division needed to find $n = Q/P$ is called logarithm in this context. This is due to the ECDLP similitude to the DLP, another hard problem widely used in cryptography. In fact, all the DLP-based algorithms can be transformed into an elliptic curve variant (e.g. DH becomes ECDH and DSA becomes ECDSA).

The order of an elliptic curve $\mathcal{E}/\mathbb{F}$ is its number of points $|\mathcal{E}(\mathbb{F})|$. Since Equation 2.1 has at most two solutions for each $x \in \mathbb{F}$, it is easy to prove that:

$$1 \leq |\mathcal{E}(\mathbb{F})| \leq 2\,|\mathbb{F}| + 1.$$

Counting the exact number of points in a curve is very important. Unfortunately there are no equations to perform such task, only algorithmic approaches such as the Schoof-Elkies-Atkin's [90], Satoh's [91] and the Arithmetic Geometric Mean (AGM) [92] algorithms.

In cryptography, the key size of an elliptic curve crypto-system is usually the bit length of the integer factor n in the ECDLP. This value can be computed as $l = \log_2(|\mathcal{E}(\mathbb{F})|)$.

While there is no formal proof that the ECDLP is intractable, no known algorithm exists to compute the discrete logarithm in PTIME (up to now). All the known methods have an EXPTIME complexity, starting with the simplest one: the brute-force attack, which has an expected running time of $O\left(2^l\right)$. Several better approaches are known in literature, such as the baby-step giant-step [93] and Pollard's rho [94] algorithms, all of them having a complexity of $O\left(2^{k/2}\right)$.

When picking a binary elliptic curve, there are several parameters that must be chosen. Interestingly, however, not all the valid values produces equally strong curves. Given an elliptic curve $\mathcal{E}/\mathbb{F}_{2^n}$ some of the minimum security checks include:

- $|\mathcal{E}(\mathbb{F}_{2^n})|$ should be large enough (at least 2^{160}) to avoid brute-force attacks;
- $\mathcal{E}$ should be non-supersingular, since supersingular curves are more susceptible to the Tate pairing attacks [95], making much easier to solve the ECDLP on them;
- n should be prime to avoid Weil descent attacks [22];
- $|\mathcal{E}(\mathbb{F}_{2^n})|$ should be prime (at least 2^{160}) or almost prime (i.e. $h \cdot p$ where p is prime and $1 \leq h \leq 4$). This is needed to avoid the Pohlig-Hellman [96] and Pollard's rho [94] attacks;
- $2^{kn} - 1$ should not be divisible by $|\mathcal{E}(\mathbb{F}_{2^n})|$ for all $1 \leq k \leq c$, where c is an integer large enough so that the ECDLP is considered intractable (at least 20) to avoid Weil [97] and Tate [95] pairing attacks.

Since choosing a strong elliptic curve is not an easy task, a common approach is not to generate a pseudo-random curve on-the-fly, but to rely on a standard curve, such as one suggested in the *FIPS PUB 186-4 – Digital Signature Standard (DSS)* [13].

2.1.5 ECDH

Elliptic Curve Diffie-Hellman (ECDH) is the elliptic curve variation of DH. Several of the most used security protocols implement it, such as IPsec [10], TLS [11] and SSH [12].

It is an anonymous key agreement protocol that allows two end-points to establish a shared secret, usually a value used to derive a cryptographic key needed to encrypt data later via a symmetric cipher.

Let suppose that A wants to communicate with B. The protocol works by following these steps, also graphically depicted in Figure 2.3:

1. A chooses an elliptic curve $\mathcal{E}/\mathbb{F}$, a random *base point* $G \neq P_\infty$ on $\mathcal{E}$ and sends them to B;
2. A and B respectively generate two random integers $n_A \neq 0$ and $n_B \neq 0$;
3. A computes $R_A = n_A \cdot G$ and sends it to B;
4. B computes $K = n_B \cdot R_A$;
5. B computes $R_B = n_B \cdot G$ and sends it to A;
6. A computes $K = n_A \cdot R_B$.

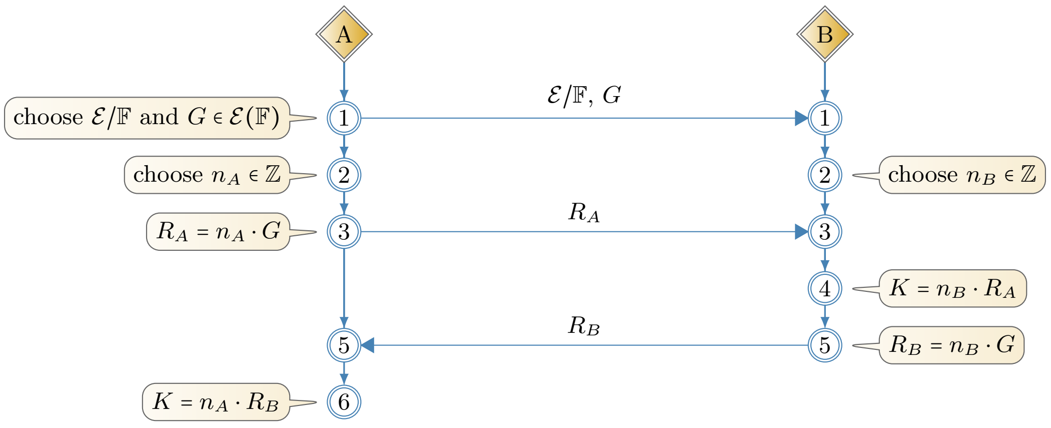

A
B
$\mathcal{E}/\mathbb{F}$, G
choose $\mathcal{E}/\mathbb{F}$ and $G \in \mathcal{E}(\mathbb{F})$
choose $n_A \in \mathbb{Z}$
choose $n_B \in \mathbb{Z}$
R_A
$R_A = n_A \cdot G$
$K = n_B \cdot R_A$
R_B
$R_B = n_B \cdot G$
$K = n_A \cdot R_B$

Figure 2.3: Steps of the ECDH protocol.

At the end of the communication, K is the secret shared between A and B. This can easily proved since $K = n_A \cdot Q_B = n_A \cdot (n_B \cdot G) = n_B \cdot (n_A \cdot G) = n_B \cdot Q_A = K$. To derive a symmetric key, a common technique is to compute an hash value of K's x coordinate.

The only private data are the two integers n_A and n_B, while all the other values are public. In order to recover the shared secret K, an attacker must intercept Q_A (or Q_B) and compute the value $n_A = Q_A/G$ (or $n_B = G_B/G$). In other words, the attacker must solve the ECDLP, assumed to be computationally unfeasible, at least if a proper curve is chosen (Section 2.1.4).

Definition (ECDHP)**.** *Let be $\mathcal{E}/\mathbb{F}$ an elliptic curve and $G, A, B \in \mathcal{E}(\mathbb{F})$ three points where $A = a \cdot G$ and $B = b \cdot G$. The problem of finding the point $K = a \cdot b \cdot G$ knowing only A, B, G is called the* Elliptic Curve Diffie-Hellman Problem (ECDHP)*.*

If an attacker can solve the ECDHP, then ECDH is broken. Note also that if the ECDLP can be efficiently solved, then the ECDHP can also be easily solved. First $a = A/G$ (or $b = B/G$) can be computed and then $K = a \cdot B$ (or $K = b \cdot A$) can be easily obtained. Thus ECDHP is no harder than ECDLP, however, in the general case, it is not yet known if the ECDHP is equally hard as the ECDLP.

Definition (ECDDHP)**.** *Let be $\mathcal{E}/\mathbb{F}$ an elliptic curve, $G, A, B, K \in \mathcal{E}(\mathbb{F})$ four points where $A = a \cdot G$, $B = b \cdot G$ and $K = k \cdot G$. The problem of finding if $K = a \cdot b \cdot G$ knowing only G, A, B, K is called the* Elliptic Curve Decision Diffie-Hellman Problem (ECDDHP)*.*

The ECDHP is concerned with computing the secret point (the shared secret). The ECDDHP puts a much harder constraint: the adversary must not learn any information about the shared secret, or, in other words, the attacker must not be able to distinguish the secret point from any other random point. Also, in this case, if the ECDHP can be efficiently solved, also the ECDDHP can be easily solved. In fact, the adversary can compute $K' = a \cdot b \cdot G$ (solving the ECDHP) and then comparing K' with K. It follows that the ECDDHP is no harder than the ECDHP. The only known complexity lower bound is $\sqrt{|\mathcal{E}/\mathbb{F}|}$ when the group order is prime [98].

The ECDH and DH protocols suffer from a series of man-in-the-middle attacks that follow the schema sketched in Figure 2.4. The idea is that an intruder E can intercept all the communications between A and B, impersonates B when communicating with A and vice-versa. The intruder can then establish a double ECDH connection with A and B, decrypting and encrypting all the traffic that the two end-points are exchanging.

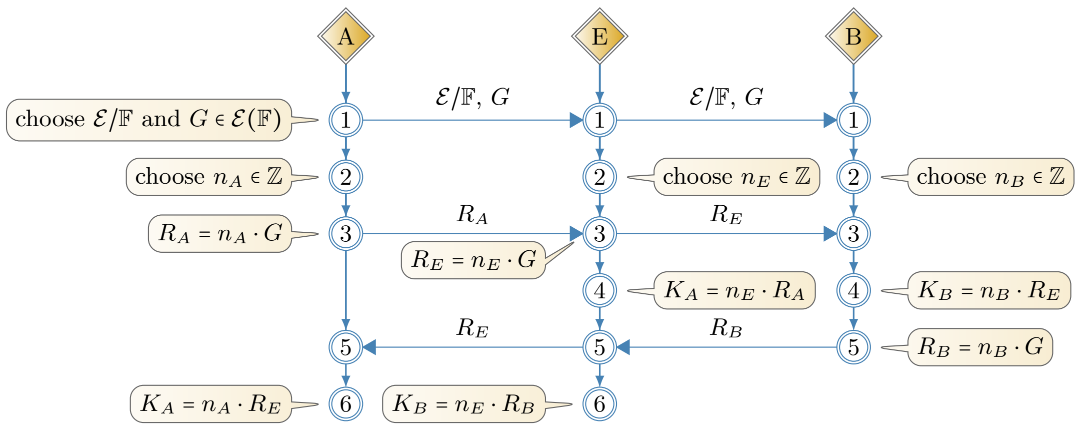

A
E
B
$\mathcal{E}/\mathbb{F}$, G
$\mathcal{E}/\mathbb{F}$, G
choose $\mathcal{E}/\mathbb{F}$ and $G \in \mathcal{E}(\mathbb{F})$
choose $n_A \in \mathbb{Z}$
choose $n_E \in \mathbb{Z}$
choose $n_B \in \mathbb{Z}$
R_A
R_E
$R_A = n_A \cdot G$
$R_E = n_E \cdot G$
$K_A = n_E \cdot R_A$
$K_B = n_B \cdot R_E$
R_E
R_B
$R_B = n_B \cdot G$
$K_A = n_A \cdot R_E$
$K_B = n_E \cdot R_B$
1
2
3
4
5
6

Figure 2.4: Man-in-the-middle attack on the ECDH protocol.

Fortunately, avoiding such family of attacks is relatively easy. The solution is to provide some mutual authentication method for the two end-points (e.g. certificates). The authenticated version of ECDH is also known as the Station-To-Station (STS) protocol.

2.2 Algorithmic background

This section is completely devoted to the algorithms needed to efficiently perform arithmetics over binary fields and binary elliptic curves. The procedures listed in the following paragraphs are designed to be implemented in software, since an optimal hardware design requires a different approach. All these algorithms are known in the current literature (with small variations) except one: the DCEA (Algorithm 2.9), which is also the core discussion of Chapter 3.

As convention, all the following algorithms works on an elliptic curve $\mathcal{E}$ defined over a generic field $\mathbb{F}_{2^n}[t]/m(t)$ with degree n and modulus $m(t)$.

2.2.1 Binary field arithmetic

Binary fields are frequently used in cryptography since their implementations offer good performance both in software and hardware.

A binary field $\mathbb{F}_{2^n}$ contains 2^n polynomials. They can be represented as a list of their n binary coefficients (a bit vector). For instance, the polynomial $t^4 + t^2 + t + 1$ over $\mathbb{F}_{2^5}$ can be expressed as $(1,0,1,1,1)_2$.

In this document, the leftmost element in a bit vector is the coefficient of the highest degree monomial. This representation allows a developer to easily model in a programming language a generic binary field element. To avoid confusion, the notation $a(t)$ will be used to denote a generic polynomial in t, while a indicates its bit vector representation.

In literature, it is quite common to write the binary field algorithms by making use of machine word arrays. In the following paragraphs, however, the presented routines are rewritten in a higher level in order to work on bit vectors. This makes them easier to read, but also closer to how the actual implementations are in the HRL and aXiom languages (Chapter 5).

Modular reduction

The modular reduction is used to reduce a polynomial $a(t)$ with $\deg(a(t)) < 2n$ into another polynomial $b(t) \in \mathbb{F}_{2^n}[t]/m(t)$ with $\deg(b(t)) < n$.

From a mathematical point of view, the only requirement is that $m(t)$ to be irreducible (see Section 2.1.2). In practice, however, if the modulus $m(t)$ is known in advance and $m(t) = t^n + r(t)$ with $\deg(r(t)) \leq \lfloor n/2 \rfloor$, a special reduction procedure can be implemented. This algorithm is much faster than the generic one [25] and its code is shown in Algorithm 2.1.

```
algorithm REDUCE(a field element $a(t)$) return $a(t) \bmod m(t)$

$u \leftarrow a[0 \rightarrow n-1]$
foreach exponent $e_i$ of $r(t)$ do
    $u \leftarrow u \oplus (a \gg n) \ll e_i$
end
$v \leftarrow u[0 \rightarrow n-1]$
foreach exponent $e_i$ of $r(t)$ do
    $v \leftarrow v \oplus (u \gg n) \ll e_i$
end
return $v[0 \rightarrow n-1]$
```

Algorithm 2.1: Binary field modular reduction algorithm.

The algorithm speed depends solely on the modulus degree and on the number of non-null monomials in the modulus. For this reason, a common approach is to make use of trinomial and pentanomial moduli [13].

Addition and subtraction

The addition in a binary field is the fastest operation since it is a simple bitwise exclusive or between the two addends. This can intuitively proved since $a + b = a \oplus b$ in $\mathbb{F}_2$ and this must also hold in any extension field $\mathbb{F}_{2^n}$. For sake of completeness, its code is given in Algorithm 2.2.

```
algorithm ADD(two field elements $a(t)$ and $b(t)$) return $a(t) + b(t)$
return $a \oplus b$
```

Algorithm 2.2: Binary field addition algorithm.

Since the exclusive or is the inverse operation of itself, the addition and the subtraction are the same operation, that is $a(t) + b(t) = a(t) - b(t)$.

Interestingly, this algorithm is independent from the field modulus, but not from its degree.

Multiplication

The field multiplication is a crucial operation in the ECC since it is used very frequently during the point calculations. In literature there exist several methods to efficiently compute the product between two binary polynomials, but two techniques are particularly prominent: the comb and the Karatsuba-Ofman methods. In addition, a special technique can be also implemented if some cryptographic hardware support is available.

Note that all these algorithms return a polynomial with $2n$ bits, so that it must be further reduced.

The *comb* method [25], also known as *comba*, is a sliding-window technique shown in Algorithm 2.3, where w represents the window width. Suppose that the goal is to obtain $a(t) \cdot b(t)$. This method works in two distinct phases. First, all the products of $a(t) \cdot c(t), \forall c(t)$ with $\deg(c(t)) \leq w$ are computed and stored in a table. Second, $b(t)$ is analyzed w bit at times and the final product is constructed by summing the right entries in the pre-computation table multiplied by a proper factor t^d.

```
algorithm COMB(two field elements $a(t)$ and $b(t)$) return $a(t) \cdot b(t)$
$u \leftarrow b[0 \rightarrow n-1]$
$table[0] \leftarrow 0$
$table[1] \leftarrow u$
for $i \leftarrow 2$ to $2^w$ do
    if $i \bmod 2 = 0$ then
        $table[i] \leftarrow table[\lfloor i/2 \rfloor] \ll 1$
    else
        $table[i] \leftarrow table[i-1] \oplus u$
    end
end
for $i \leftarrow 0$ to $\lceil n/w \rceil$ do
    $index \leftarrow$ convert $a[i \cdot w \rightarrow i \cdot (w+1)]$ into an integer
    $v \leftarrow v \oplus table[index] \ll (j \cdot w)$
end
return $v$
```

Algorithm 2.3: Binary field comb multiplication algorithm.

When the window width equals to 1, this method becomes the so called *school-book method* where the multiplication is performed bit-by-bit, the most naive and slow approach.

Obviously, the value of w plays a major role in the algorithm performance. On one hand, if it is too small, the pre-computation phase is very fast, but the second half of the algorithm is significantly slowed down. On the other hand, if w is very large, the table construction step dominates too much the total computation time. The optimal size of the window width heavily depends on the target architecture and the field degree.

While the comb method is a window-based method, the Karatuba-Ofman [99] relies on the idea to simplify a bigger multiplication into three smaller and simpler ones. The factors $a(t)$ and $b(t)$ can be split into two smaller polynomials, so that:

$$\begin{aligned} a(t) &= a_H(t) \cdot t^{\lceil n/2 \rceil} + a_L(t) \\ b(t) &= b_H(t) \cdot t^{\lceil n/2 \rceil} + b_L(t). \end{aligned}$$

Thus, the product $a(t) \cdot b(t)$ can be written as:

$$\begin{aligned} a(t) \cdot b(t) = &(a_H(t) \cdot t^{\lceil n/2 \rceil} + a_L(t)) \cdot (b_H(t) \cdot t^{\lceil n/2 \rceil} + b_L(t)) = \\ &= a_H(t) b_H(t) t^n + \\ &+ ((a_H(t) + a_L(t))(b_H(t) + b_L(t)) + a_H(t) + b_H(t) + a_L(t) + b_L(t)) t^{\lceil n/2 \rceil} + \\ &+ a_L(t) b_L(t). \end{aligned}$$

The smaller field multiplication operator can be implemented through a comb or another Karatsuba-Ofman algorithm. Initially proposed to perform product of multi-digit integer numbers, this approach can be easily modified to work on binary fields.

The Algorithm 2.4 implements the previously mentioned ideas.

```
**algorithm** KARATSUBA(two field elements $a(t)$ and $b(t)$) **return** $a(t) \cdot b(t)$
$a_L \leftarrow a[0 \rightarrow \lceil n/2 \rceil - 1]$
$a_H \leftarrow a[\lceil n/2 \rceil \rightarrow n]$
$b_L \leftarrow b[0 \rightarrow \lceil n/2 \rceil - 1]$
$b_H \leftarrow b[\lceil n/2 \rceil \rightarrow n]$
$u \leftarrow a_H \cdot b_H$
$v \leftarrow a_L \cdot b_L$
$w \leftarrow (a_H \oplus a_L) \cdot (b_H \oplus b_L)$
$w \leftarrow w \oplus u \oplus v$
$z_1 \leftarrow 0$
$z_1[n \rightarrow \lceil n/2 \rceil - 1 + n] \leftarrow u$
$z_2 \leftarrow 0$
$z_2[0 \rightarrow \lceil n/2 \rceil - 1] \leftarrow v$
$z_3 \leftarrow 0$
$z_3[\lceil n/2 \rceil \rightarrow n - 1] \leftarrow w$
**return** $z_1 \oplus z_2 \oplus z_3$
```

Algorithm 2.4: Binary field Karatsuba-Ofman multiplication algorithm.

On some processors, a third, faster approach to tackle the multiplication can be developed, if some special assembler instructions are supported (Section 6.3.3). If these facilities are available, the code shown in Algorithm 2.5 can be used, where w represents a proper window width.

The algorithm treats the multiplication by simplifying it in several $w \times w$ products. For instance, on some Intel CPUs, a 64×64 bits multiplication can be fully performed via a special assembler operation (Section 6.3.3).

algorithm WindowMultiply(two field elements $a(t)$ and $b(t)$) **return** $a(t) \cdot b(t)$

```
u ← 0
for $i \leftarrow 0$ to $\lceil n/w \rceil$ do
    for $j \leftarrow 0$ to $\lceil n/w \rceil$ do
        $u \leftarrow a[i \cdot w \rightarrow i \cdot (w+1)] \cdot b[j \cdot w \rightarrow j \cdot (w+1)]$
        $v \leftarrow v \oplus u \ll (w \cdot (i+j))$
    end
end
return $v$
```

Algorithm 2.5: Binary field window-based multiplication algorithm.

If some hardware support is available, the WindowMultiply algorithm is usually the fastest one. Otherwise, the Comb method is the quickest one for fields with a small degree, while the KaratsubaOfman approach is best suited for the larger fields.

Squaring

Computing the square of a polynomial is a relatively common operation in ECC. Technically, this can be accomplished by a multiplicative procedure, however, an ad-hoc algorithm can be devised to achieve better performance. There exists two fundamental techniques to perform the squaring. The first one works well on all the platforms, while the second one requires some special hardware support, but has vastly superior performance.

The first algorithm has a simple and elegant bit-wise explanation. The square of a bit vector a can be constructed by inserting a zero between all the bits of a. For instance, the square of $(1,1,0,1,0)_2$ is $(1,0,1,0,0,0,1,0,0)_2$. To increase the computation speed, a static look-up table can be used to produce the result using a window-based approach [25]. Its code is shown in Algorithm 2.6, where w represents the window width.

algorithm PrecomputedSquare(a field elements $a(t)$) **return** $a^2(t)$

```
$table \leftarrow (\bigoplus_{i=0}^{w-1} ((j \gg i) \bmod 2) \ll (2i))_j, \forall j \in [0, 2^w - 1]$
$u \leftarrow 0$
for $i \leftarrow 0$ to $\lceil n/w \rceil$ do
    $index \leftarrow$ convert $a[iw \rightarrow i(w+1)]$ into an integer
    $u[2wi \rightarrow 2w(i+1)] \leftarrow table[index]$
end
return $u$
```

Algorithm 2.6: Binary field precomputation-based squaring algorithm.

As expected, the window width w greatly affects the running time of the algorithm. Several implementations (e.g. OpenSSL) makes use of $w = 8$, since accessing and manipulating bytes is easy and well supported in several hardware architectures.

As in Algorithm 2.5, if the CPU implements some special cryptographic circuitry (Section 6.3.3), the Algorithm 2.7 can be used instead, achieving better performance.

Note that in order to obtain the final squarings, a reduction is still needed to be performed on the algorithm results.

```
algorithm WindowSquare(a field elements $a(t)$) return $a^2(t)$
$u \leftarrow 0$
for $i \leftarrow 0$ to $\lceil n/w \rceil$ do
    for $j \leftarrow 0$ to $\lceil n/w \rceil$ do
        $u \leftarrow a[i \cdot w \rightarrow i \cdot (w+1)] \cdot a[j \cdot w \rightarrow j \cdot (w+1)]$
        $v \leftarrow v \oplus u \ll (w \cdot (i+j))$
    end
end
return $v$
```

Algorithm 2.7: Binary field window-based squaring algorithm.

Inversion and division

The multiplicative inverse computation is the slowest elementary operation that can be achieve on a binary field, so its optimization is crucial in order to obtain better performance on higher level algebraic structures.

In literature, there exist various algorithms that can be used to perform such computations [88], most of them based of the Euclidean algorithm. The core concept of these approaches is that the multiplicative inverses can be efficiently computed by leveraging the fact that for all the binary polynomials $c(t)$ the following assumption holds:

$$\gcd(a(t), b(t)) = \gcd(b(t) + c(t)a(t), a(t)).$$

By iteratively applying the aforementioned equation, the inverse can be computed. All the algorithms proposed in this section return the already reduced multiplicative inverse, so that a modular reduction round is not needed.

All the following techniques make use of the $\mathrm{fls}(\cdot)$ operator that computes the *find last set* of a binary tuple, that is the index of the leftmost bit set to one. If a bit vector a is translated into an unsigned integer $\overline{a}$, computing the last set is the same as calculating $\lceil \log_2(\overline{a}) \rceil$. This is also equivalent to compute the degree of the polynomial $a(t)$, so that $\mathrm{fls}(a) \equiv \lceil \log_2(\overline{a}) \rceil \equiv \deg(a(t))$. Some processors (e.g. all the modern AMD64 families) allow to compute this values directly in hardware, speeding up significantly the execution time of the multiplicative inversion algorithms (Section 6.3.3).

The Binary Euclidean Algorithm (BEA) computes $a^{-1}(t)$ by finding a $g(t)$ such that:

$$a(t)g(t) + m(t)h(t) = 1.$$

At the end of the computations, the polynomial $g(t)$ is $a^{-1}(t)$ since $a(t)g(t) = 1 + m(t)h(t) = 1$. Its code is listed in Algorithm 2.8.

An interesting variation of the BEA method is the DCEA [18, 19]. From an algorithmic point of view, the main difference is that the two nested loops of the Binary Euclidean algorithm are merged together into a single, bigger one having a slightly restructured order of the instructions. This approach is discussed in detail in Chapter 3 and its code is listed in Algorithm 2.9.

Traditional compilers are quite sensitive to how the source is organized and they can produce radically different assembly code even with a slight variation on the input source code. Even if the BEA and DCEA apply the same operations, the final performance of their implementations can be different, potentially giving the developer a faster technique to compute the multiplicative inverse.

```
algorithm BEA(a field elements $a(t)$) return $a^{-1}(t)$
$u \leftarrow a$
$v \leftarrow m$
$g_1 \leftarrow 1$
$g_2 \leftarrow 0$
while true do
    while $u[0] = 0$ do
        $u \leftarrow u \gg 1$
        if $g_1[0] = 0$ then
            $g_1 \leftarrow g_1 \gg 1$
        else
            $g_1 \leftarrow (g_1 \oplus m) \gg 1$
        end
    end
    if $u = 1$ then
        return $g_1$
    end
    if fls$(u) <$ fls$(v)$ then
        $u \leftrightarrow v$
        $g_1 \leftrightarrow g_2$
    end
    $u \leftarrow u \oplus v$
    $g_1 \leftarrow g_1 \oplus g_2$
end
```

Algorithm 2.8: Binary field BEA inversion algorithm.

The Extended Euclidean Algorithm (EEA) makes use of a similar approach, but exploits the degree difference of the polynomials $\deg(u(t)) - \deg(v(t))$ to avoid some iterations. A drawback of this approach is that it requires to perform several shifts whose value cannot be known at compile time, thus being harder to optimize. Note that the BEA and DCEA only require to perform right shifts of 1 position.

The EEA pseudo-code is listed in Algorithm 2.10.

The inversion computation can be exploited to achieve the division by leveraging the fact that $b(t)/a(t) = b(t) \cdot a^{-1}(t)$. However the BEA, DCEA and EEA algorithms can also be easily modified to directly produce the dividend, avoiding a multiplication. In order to compute $b(t)/a(t)$, Algorithms 2.8, 2.9 and 2.10 can be changed by replacing $g_1 \leftarrow 1$ with $g_1 \leftarrow b$. On termination, the result will be $g_1 = b \cdot a^{-1}$.

2.2.2 Elliptic curve arithmetic

The arithmetic of elliptic curves consists of four operations: point addition, doubling, negation and scalar multiplication. Since these operations are mapped to the base field algebra, a fast field arithmetic is a necessary prerequisite to achieve good performance in ECC.

Base operations

The point addition, doubling and negation are performed on the coordinates of some points, that are elements of the underlying field. An elegant and simple way to represent the complexity of

```
algorithm DCEA(a field elements $a(t)$) return $a^{-1}(t)$

$u \leftarrow a$
$v \leftarrow m$
$g_1 \leftarrow 1$
$g_2 \leftarrow 0$
$direction \leftarrow backward$
while $true$ do
    if $direction = backward$ then
        if $u[0] = 0$ and $g_1[0] = 1$ then
            $g_1 \leftarrow g_1 \oplus m$
        end
        $direction \leftarrow forward$
    else
        if $u[0] = 0$ then
            $u \leftarrow u \gg 1$
            $g_1 \leftarrow g_1 \gg 1$
        else
            if $\mathrm{fls}(u) < \mathrm{fls}(v)$ then
                $u \leftrightarrow v$
                $g_1 \leftrightarrow g_2$
            end
            $u \leftarrow u \oplus v$
            $g_1 \leftarrow g_1 \oplus g_2$
        end
        $direction \leftarrow backward$
    end
    if $u = 1$ then
        return $g_1$
    end
end
```

Algorithm 2.9: Binary field DCEA inversion algorithm.

a point operation is to express it as an equation that specifies how many field operations are executed. The following symbols are used to denote the base field operation costs: $\mathcal{M}$ for the field multiplication, $\mathcal{S}$ for the field squaring and $\mathcal{I}$ for the field inversion. Since the binary field addition (a bitwise exclusive or) has a negligible cost, it is omitted for simplicity.

Computing the base field multiplicative inversion is by far the most expensive operation, however, by making use of some special coordinate systems, the number of divisions and inversions can be significantly reduced. Note that the complexity of a point operation depends not only on the chosen coordinate system, but also on how the equations are factorized (Section 6.2.6). In the following paragraphs, four coordinate systems will be analyzed:

- the *affine* coordinate system is the simplest (as well the slowest) one and make use of the formulas reported in Section 2.1.3;

- the *projective* coordinate system maps the point $(x : y : z)$ into the affine point $(x/z, y/z)$ if $z \neq 0$ and $P_\infty = (0 : 1 : 0)$ if $z = 0$;

```
**algorithm** EEA(a field elements $a(t)$) **return** $a^{-1}(t)$
$u \leftarrow a$
$v \leftarrow m$
$g_1 \leftarrow 1$
$g_2 \leftarrow 0$
while $u \neq 1$ do
    $j \leftarrow \text{fls}(u) - \text{fls}(v)$
    if $j < 0$ then
        $u \leftrightarrow v$
        $g_1 \leftrightarrow g_2$
        $j \leftarrow -j$
    end
    $u \leftarrow u \oplus (v \ll j)$
    $g_1 \leftarrow g_1 \oplus (g_2 \ll j)$
end
return $g_1$
```

Algorithm 2.10: Binary field EEA inversion algorithm.

- the *Jacobian* coordinate system maps the point (x, y, z) into the affine point $(x/z^2, y/z^3)$ if $z \neq 0$ and $P_\infty = (1, 1, 0)$ if $z = 0$;
- the *López-Dahab* coordinate system maps the point $(x; y; z)$ into the affine point $(x/z, y/z^2)$ if $z \neq 0$ and $P_\infty = (1; 0; 0)$ if $z = 0$.

Several cryptographic protocols require to use points in an affine space, so that, if needed, once all the computations are performed a *normalization* phase must be done to convert a point in non-affine space into the affine one. These operations are performed scarcely, hence their performance costs are usually negligible.

Table 2.1 lists the formulas for the base operations in the four coordinate systems, while Table 2.2 shows their respective optimal complexity costs [24], when correctly factorized.

Usually the López-Dahab coordinate system offers superior performance, as clear from the cost formulas.

Scalar multiplication

The scalar multiplication is the operation that dominates the computational time in the elliptic curve arithmetic. Several algorithms can be implemented to obtain the product in an efficient manner and in this section three of them are presented. All these approaches are completely independent from the choice of the coordinate systems.

The scalar multiplication algorithms' complexities can be written as a function of the point addition and doubling costs, respectively denoted by the $\mathcal{A}$ and $\mathcal{D}$ symbols. The negation is omitted since its performance are usually negligible.

One of the simplest algorithms is the *add-and-double* and its code is shown in Algorithm 2.11. It is based on the observation that any natural number $n = (\, n_{l-1}, \ldots, n_0 \,)_2$ can be written as:

$$n = n_0 2^0 + n_1 2^1 + n_2 2^2 + n_3 2^3 + \ldots = n_0 + 2(n_1 + 2(n_2 + 2(n_3 + \ldots))).$$

system	addition $R = P + Q$	doubling $R = 2P$	negation $R = -P$
affine	$\begin{cases} a = (y_P + y_Q)/(x_P + x_Q) \\ x_R = a^2 + a + x_P + x_Q + a_2 \\ y_R = a(x_P + x_R) + x_R + y_P \end{cases}$	$\begin{cases} a = x_P + y_P/x_P \\ x_R = a^2 + a + a_2 \\ b = a(x_P + x_R) \\ y_R = b + x_R + y_P \end{cases}$	$\begin{cases} x_R = x_P \\ y_R = x_P + y_P \end{cases}$
projective	$\begin{cases} a = y_P z_Q + z_P y_Q \\ b = x_P z_Q + z_P x_Q \\ c = (a^2 + ab + a_2 b^2) z_P z_Q + b^3 \\ d = b^2 (a x_P + y_P b) z_Q \\ x_R = bc \\ y_R = d + (a + b)c \\ z_R = b^3 z_P z_Q \end{cases}$	$\begin{cases} a = x_P^2 + y_P z_P \\ b = x_P z_P \\ c = a^2 + ab + a_2 b^2 \\ x_R = bc \\ y_R = (a + b)c + x_P^4 b \\ z_R = c^3 \end{cases}$	$\begin{cases} x_R = x_P \\ y_R = x_P + y_P \\ z_R = z_P \end{cases}$
Jacobian	$\begin{cases} a = x_P z_Q^2 \\ b = x_Q z_P^2 \\ c = y_P z_Q^3 \\ d = y_Q z_P^3 \\ e = (c + d) x_Q + (a + b) z_P y_Q \\ z_R = (a + b) z_P z_Q \\ f = c + d + z_R \\ x_R = a_2 z_R^2 + (c + d) f + (a + b)^3 \\ y_R = f x_R + ((a + b) z_P)^2 e \end{cases}$	$\begin{cases} a = x_P^4 \\ b = z_P^2 \\ c = x_P^2 + y_P z_P \\ x_R = a + a_6 b^4 \\ z_R = x_P b \\ y_R = a z_R + (c + z_R) x_R \end{cases}$	$\begin{cases} x_R = x_P \\ y_R = x_P z_P + y_P \\ z_R = z_P \end{cases}$
López-Dahab	$\begin{cases} a = (x_P z_Q)^2 \\ b = (x_Q z_P)^2 \\ c = x_P z_Q + x_Q z_P \\ d = y_P z_Q^2 \\ e = y_Q z_P^2 \\ f = d + e \\ z_R = (a + b) z_P z_Q \\ x_R = x_P z_Q (e + b) + x_Q z_P (a + d) \\ g = x_P z_Q f c + (a + b) d \\ y_R = i(a + b) + (f c + z_R) x_R \end{cases}$	$\begin{cases} a = a_6 z_P^4 \\ z_R = z_P^2 x_P^2 \\ x_R = x_P^4 + a \\ b = y_P^2 + a_2 z_R + a \\ y_R = b x_R + z_R a \end{cases}$	$\begin{cases} x_R = x_P \\ y_R = x_P z_P + y_P \\ z_R = x_P \end{cases}$

Table 2.1: Elliptic curve formulas for various coordinate systems.

If n is on l bits and it is randomly generated, its Hamming weight (the number of non-null items that it contains) will be $l/2$ in the average case. That means that the Algorithm 2.11 performs about $l/2$ point additions per multiplication. This method is quite fast on small integers, but its performance deteriorate quickly when l starts to increase. For most of the interesting cases in ECC, the add-and-double approach is unsuitable, but it can be modified to achieve faster speeds by making use of an alternative representation instead of the traditional binary one.

Definition (Signed-digit representation)**.** *The* signed-digit representation *of an integer n in radix*

system	addition	doubling	negation
affine	$1\mathcal{I} + 2\mathcal{M} + 1\mathcal{S}$	$1\mathcal{I} + 2\mathcal{M} + 1\mathcal{S}$	$0\mathcal{M}$
projective	$14\mathcal{M} + 1\mathcal{S}$	$7\mathcal{M} + 3\mathcal{S}$	$0\mathcal{M}$
Jacobian	$14\mathcal{M} + 5\mathcal{S}$	$4\mathcal{M} + 5\mathcal{S}$	$1\mathcal{M}$
López-Dahab	$13\mathcal{M} + 4\mathcal{S}$	$3\mathcal{M} + 5\mathcal{S}$	$1\mathcal{M}$

Table 2.2: Elliptic curve costs for various coordinate systems.

```
**algorithm** ADDANDDOUBLE($n \in \mathbb{N}$, $P \in \mathcal{E}(\mathbb{F})$) **return** $n \cdot P$
$Q \leftarrow P_\infty$
for $i \leftarrow 0$ to $l-1$ do
    if $n_i = 1$ then
        $Q \leftarrow Q + P$
    end
    $P \leftarrow 2P$
end
return $Q$
```

Algorithm 2.11: Elliptic curve add-and-double scalar multiplication algorithm.

b is given by the summation:

$$n = \sum_{i=0}^{l-1} n_i b^i, \quad |n_i| < b.$$

A signed-digit representation where the radix is 2 is called a *signed-binary digit representation*. In this case $n_i \in \{-1, 0, 1\}$.

Definition (NAF)**.** *A signed-digit representation is in* Non-Adjacent Form (NAF) *if* $n_i \cdot n_{i+1} = 0$*, for all* $i \geq 0$*. This form is denoted by* $(n_{l-1}, \dots, n_0)_{NAF}$.

EXAMPLE. A signed-binary digit representation of 478 is $(1,0,-1,1,1,0,0,-1,1,0)$ and its unique NAF is $(1,0,0,0,-1,0,0,0,-1,0)_{NAF} = 2^9 - 2^5 - 2^1 = 512 - 32 - 2 = 578$.

Computing the NAF representation of an integer can be performed by using the Algorithm 2.12. Its running time is linear with respect to the bit length of the input integer n.

```
**algorithm** COMPUTENAF($n = (0, 0, n_{l-2}, \dots, n_0)_2$) **return** the NAF of $n$
$c_0 \leftarrow 0$
for $i \leftarrow 0$ to $l-1$ do
    $c_{i+1} \leftarrow \lfloor (c_i + n_i + n_{i+1})/2 \rfloor$
    $n'_i \leftarrow c_i + n_i - 2c_{i+1}$
end
return $(n'_{l-1}, \dots, n'_0)_{NAF}$
```

Algorithm 2.12: NAF computation algorithm.

Algorithm 2.13 shows the add-and-double procedure modified to use the NAF representation.

algorithm NAFMULTIPLY($n \in \mathbb{N}$, $P \in \mathcal{E}(\mathbb{F})$) **return** $n \cdot P$

```
$(n_{l-1}, \dots, n_0)_{NAF}$ = COMPUTENAF($n$)
$Q \leftarrow P_\infty$
for $i \leftarrow l-1$ to $0$ do
    $Q \leftarrow 2Q$
    if $n_i = 1$ then
        $Q \leftarrow Q + P$
    else if $n_i = -1$ then
        $Q \leftarrow Q - P$
    end
end
return $Q$
```

Algorithm 2.13: Elliptic curve NAF-based scalar multiplication algorithm.

Remark. The NAF of an integer n on l bits is unique, its Hamming weight is minimal among all the signed-digit representations of n and its average value is $l/3$.

The Algorithm 2.13 in average computes $l/3$ additions/subtractions w.r.t. to the $l/2$ additions of the Algorithm 2.11. The idea behind the NAF can be extended using a radix greater than 2, allowing an ad-hoc method to be faster.

Definition (NAF$_w$)**.** *A* width-w NAF (NAF$_w$) *of a positive integer n is a signed-digit representation of n where each n_i is zero or odd, $|n_i| < 2^{w-1}$ and among any w consecutive coefficients at most one is zero. This form is denoted by $(n_{l-1}, \dots, n_0)_{NAF_w}$.*

Similarly to a value's NAF, the NAF$_w$ is also unique and the Algorithm 2.14 can compute it.

algorithm COMPUTENAFW($n = (0, 0, n_{l-2}, \dots, n_0)_2$) **return** the NAF$_w$ of n

```
$i \leftarrow 0$
while $k \neq 0$ do
    if $n[0] = 1$ then
        $n'_i \leftarrow n \operatorname{mods} 2^w$
        $n \leftarrow n - n'_i$
    else
        $n'_i \leftarrow 0$
    end
    $n \leftarrow n \gg 1$
    $i \leftarrow i + 1$
end
return $(n'_{l-1}, \dots, n'_0)_{NAF_w}$
```

Algorithm 2.14: NAF$_w$ computation algorithm.

In Algorithm 2.14, the mods operator performs a *signed modular reduction* defined as:

$$n \operatorname{mods} 2^w = \begin{cases} (n \bmod 2^w) - 2^w & \text{if } n \bmod 2^w \geq 2^{w-1} \\ n \bmod 2^w & \text{otherwise.} \end{cases}$$

The scalar multiplication method based on the NAF$_w$ representation is shown in Algorithm 2.15.

```
algorithm NAFwMultiply($n \in \mathbb{N}$, $P \in \mathcal{E}(\mathbb{F})$) return $n \cdot P$
$(n_{l-1}, \dots, n_0)_{NAF}$ = ComputeNAFw($n$)
compute $P_i = i \cdot P$ for all $i \in \{1, 3, 5, \dots, 2^{w-1} - 1\}$
$Q \leftarrow P_\infty$
for $i \leftarrow l - 1$ to $0$ do
    $Q \leftarrow 2Q$
    if $n_i > 0$ then
        $Q \leftarrow Q + P_{n_i}$
    else if $n_i < 0$ then
        $Q \leftarrow Q - P_{-n_i}$
    end
end
return $Q$
```

Algorithm 2.15: Elliptic curve NAF$_w$-based scalar multiplication algorithm.

Remark. The NAF$_w$ of an integer n on l bits is unique and its average Hamming weight is $l/(w+1)$.

The NAFwMultiply computes less and less additions and subtractions as the NAF$_w$ width w is increased. However, if w is too large, the initial pre-computation phase, where the table P_i is calculated, starts to dominate the running time and can significantly slow-down the whole procedure.

Table 2.3 lists the average complexity costs of the aforementioned multiplication algorithms.

algorithm	average cost with an l bit length integer
AddAndDouble	$\frac{l}{2}\mathcal{A} + l\mathcal{D}$
NAFMultiply	$\frac{l}{3}\mathcal{A} + l\mathcal{D}$
NAFwMultiply	$\left(2^{w-2} + \frac{l}{w+1} - 1\right)\mathcal{A} + (l+1)\mathcal{D}$

Table 2.3: Elliptic curve costs for various scalar multiplication algorithms.

2.2.3 ECDH

If a scalar multiplication algorithm is available, the ECDH implementation is trivial.

In order to establish a shared secret via the ECDH protocol (Figure 2.3), each party must perform two main phases:

1. a public key generation phase, where given a random integer n_A (or n_B), the value $R_A = n_A \cdot G$ (or $R_B = n_B \cdot G$) is computed;

2. derive the secret point by computing $K = n_A \cdot R_B$ (or $K = n_B \cdot R_A$).

If we exclude the random data generation, the whole protocol reduces itself to calculating only two scalar multiplications, so that the algorithms presented in Section 2.2.2 can be directly applied.

2.2.4 Summary of decisions

In order to achieve the fastest scalar multiplication a series of decisions must be taken. They are briefly summarized in Figure 2.5

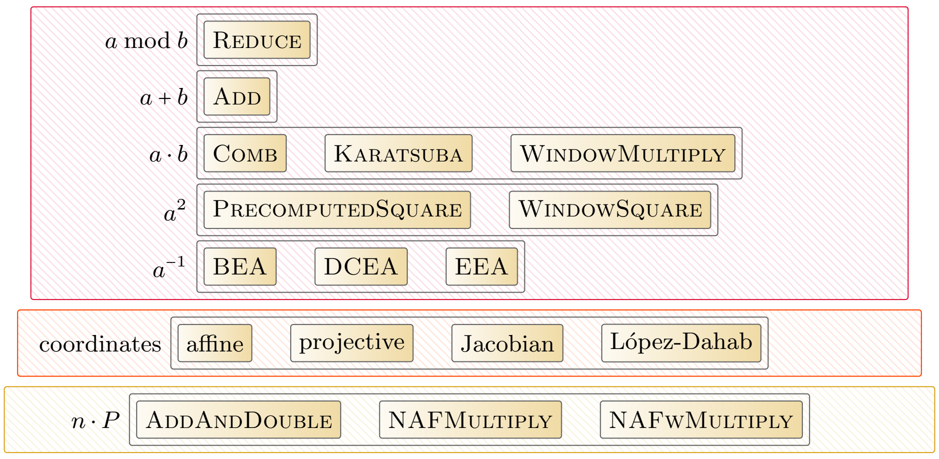


Figure 2.5: Algorithm choices for the scalar multiplication implementation.

First, the field operation algorithms must be selected, one for each of the five base operations (modular reduction, addition, multiplication, squaring and inversion/division). Some algorithms such as the Comb are window-based, so a proper window width must be also picked. Second, the best coordinate system for the curve base operations (point addition, doubling and negation) should be chosen. It is usually the López-Dahab system. Finally, a proper algorithm for the scalar multiplication must be selected.

Note that all these choices depend by the field structure, the curve parameters, the hardware platform features and the algorithm implementations. Changing even only one of these factors might require a new search for the optimal suite of algorithms from scratch.

Chapter 3

Implicit computational complexity

The mathematics is not there till we put it there.

The Philosophy of Physical Science
Sir Arthur Eddington

Mathematical logic families usually focus on formal proofs, that is their goal is to prove that some property holds or not. Algorithms, on the other hand, focus on computational mechanisms, by showing the processes needed to compute a value. In the 1930s Haskell Brooks Curry and later William Alvin Howard in the 1960s, started to note a number of equivalences between some kind of logical families and programming languages. This led to the so-called *Curry-Howard correspondence* [100], stating that some proof systems and programming languages are structurally the same. In other words, proving something in a certain logic family is equivalent to write an algorithm with a particular programming language and vice-versa.

Several interesting properties can be proved by applying the Curry-Howard correspondence to a program. Remarkably, inferring the computational complexity class of an algorithm is one of these. The branch of Implicit Computational Complexity (ICC) is interested in proving the complexity of some algorithm without an explicit reference to a machine model and to cost bounds. In this context, there is a logic family, known as Dual Light Affine Logic (DLAL), that offers a peculiar property: any algorithm translatable in DLAL has FPTime complexity. This is due to the fact that this logic system does not have the expressivity to represent non-FPTime algorithms.

All the ECC algorithms have PTime complexity (except the ECDLP-solving ones), so they can be rewritten in DLAL form [101, 18, 19]. The DLAL version of an algorithm might suggest how to better reorganize its operations and data structures, thus potentially leading to a restructured algorithm with better performance.

This chapter is split into two sections. The first one gives a brief, not too formal background behind DLAL. The second one is the core of this chapter and describes how DCEA (Algorithm 2.9) was developed by introducing the DLAL version of BEA (Algorithm 2.8), since DCEA is basically the functional BEA translated into an imperative language.

The content of this chapter is based on the following sources:

- Torben Braüner. «Introduction to Linear Logic». 1996. URL: `http://www.brics.dk/LS/96/6/BRICS-LS-96-6.pdf` (visited on 03/26/2016);

- Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*. Cambridge University Press, 1989. ISBN: 0-521-37181-3;

- Patrick Baillot and Kazushige Teruib. «Light types for polynomial time computation in lambda calculus». In: *Information and Computation* 207 (1 2008), pp. 41–62. DOI: `10.1016/j.ic.2008.08.005`;
- Emanuele Cesena, Marco Pedicini, and Luca Roversi. «Typing a Core Binary Field Arithmetic in a Light Logic». In: *FOPARA 2011, proceedings of the 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis*. Madrid (Spain), 2011. DOI: `10.1007/978-3-642-32495-6_2`;
- Daniele Canavese et al. «Can a Light Typing Discipline Be Compatible with an Efficient Implementation of Finite Fields Inversion?» In: *FOPARA 2013, proceedings of the 3rd International Workshop on Foundational and Practical Aspects of Resource Analysis*. Bertinoro (Italy), 2014. DOI: `10.1007/978-3-319-12466-7_3`;
- Daniele Canavese et al. «Light combinators for finite fields arithmetic». In: *Science of Computer Programming* 111 (3 2015), pp. 365–394. DOI: `10.1016/j.scico.2015.04.001`.

3.1 Logic background

This section gradually builds the mathematical background needed to comprehend DLAL starting from the basic notion of the classical logic. For the sake of brevity, all the inference and proof rules are omitted in the following paragraphs, but they are available in Appendix A.

From now on, the meta-variables consisting of an upper Latin letter (e.g. X and Y) range over formulas, while the meta-variables denoted with an upper Greek letter (e.g. Γ and Δ) range over list of formulas.

The notation $\Gamma \vdash \Delta$ indicates a *sequent*, that is a generalization of an implication. Its meaning is that if all the formulas in Γ are true, then at least one formula in Δ hold. The parameters of the sequent operator $\vdash$ are known as *contexts* (Γ and Δ in this case). Every term in the left-hand context is known as an *antecedent*, while the terms in the right-hand side context are called *succedents* or *consequents*. For instance, $\vdash B$ means that B is true and $A_1, A_2 \vdash B$ means that $A_1 \wedge A_2 \Rightarrow B$.

3.1.1 Classical logic

The formulas of the classical logic can be defined via the BNF rule:

$$s ::= \text{“}\texttt{0}\text{”} \mid \text{“}\texttt{1}\text{”} \mid s \text{ “}\wedge\text{” } s \mid s \text{ “}\vee\text{” } s \mid s \text{ “}\Rightarrow\text{” } s.$$

Where $\wedge$, $\vee$ and $\Rightarrow$ are the usual logical *and*, *or* and *implication* operators. The negation $\neg X$ does not appear in the list since it can be defined as $X \Rightarrow 0$.

The proof rules for the CL are listed in Appendix A.1.

3.1.2 Intuitionistic logic

The Intuitionistic Logic (IL) is very close the classical one. The main difference is that the IL is based on the notion of *constructive proof*. CL is concerned about the existence of a particular mathematical object, while IL is interested in determining how to build it. For instance, the expression $\exists x : x \bmod 2 = 0$ is true in CL if it can be inferred somehow that at least one even number exists, while it is true in IL only if a method for finding at least one even number is found.

The formulas of the IL can be constructed exactly with the same BNF rule of the CL. The proof rules are however slightly different and are shown in Appendix A.2.

An interesting interpretation of the IL is the so-called Brouwer-Heyting-Kolmogorov functional interpretation [102], where the formulas are interpreted by means of their proofs. That is:

- $X \vee Y$ is considered true if at least one of the terms X or Y is true and there is a method by which it is possible to find out which holds;
- $X \wedge Y$ is considered true if both X and Y are true;
- $\neg X$ is considered true if the proof of X leads to absurdity;
- $X \Rightarrow Y$ is considered true if there exists a method by which a proof of Y can be inferred from the proof of X.

Since IL poses very strict constraints, several classical rules are abolished or modified. For instance, in the general case, $X \vee (\neg X)$ holds in CL, but not in IL since there is no general method of finding, for any given X, whether X or $\neg X$ is true.

3.1.3 Linear logic

The Linear Logic (LL) is a refinement of both CL and IL, which takes ideas from both the families.

In CL (and also in IL, but with a slightly different semantic), $X \Rightarrow Y$ means that whenever X is true, then also Y holds, but X still holds. In real worlds scenarios, this might not be always the case since the implication is 'causal'. A causal implication can not be executed an indefinitely number of times, since the premises are modified after their use. In physics, the modification of a premise is known as *reaction*. For instance, the phrase 'X sells an apple to Y $\Rightarrow$ Y gets an apple' can be executed only if X has still some apples. This can be modeled in LL, but not in CL or IL.

The LL formulas can be described by means of the syntactical rule:

$$s ::= \text{“}\mathtt{0}\text{”} \mid \text{“}\mathtt{1}\text{”} \mid \text{“}\top\text{”} \mid \text{“}\bot\text{”} \mid s \text{ “}\otimes\text{” } s \mid s \text{ “}⅋\text{” } s \mid s \text{ “}\&\text{” } s \mid s \text{ “}\oplus\text{” } s \mid s \text{ “}\multimap\text{” } s \mid \text{“}!\text{”}s \mid \text{“}?\text{”}s.$$

This logic family introduces several new unique operators that can be interpreted as working on resources instead on truth values, that is:

- the $X \otimes Y$ and $X \& Y$ expressions respectively denote the *multiplicative conjunction* and *additive conjunction*. In both the cases it means that X and Y are available, but in the first case both X and Y will be consumed, while in the latter only one will be used.
- the $X \oplus Y$ and $X ⅋ Y$ instead denote the *multiplicative disjunction* and *additive disjunction*. Their meaning is that at least one resource between X and Y is available, but in the first case both the resources are used, while only one is consumed in the latter.
- the $X \multimap Y$ is the *linear implication*. It has a causal interpretation and it expresses that when X is available, it is consumed and Y is produced.
- the $!X$ expresses the *of course* operator (also known as *bang* operator). Intuitively it means that the resource X is available an infinite number of times.
- the $?X$ formula uses the *why not* operator. It specifies that the resource X is available a finite number of times.

The values 0, 1, $\top$ and $\bot$ are the identity elements respectively for $\oplus$, $\otimes$, $\&$ and ⅋.

The LL allows to express several intuitionistic rules. For instance, $X \Rightarrow Y$ becomes $!X \multimap Y$, that is X implies Y when Y is caused by some interaction of X.

The inference rules for the linear logic are reported in Appendix A.3.

3.1.4 λ-calculus

The vast majority of the modern programming languages follow the *imperative programming* paradigm (e.g., C, Java, but also HRL and aXiom). These languages specify what must be done, step by step, essentially modeling how to change the program's states. In contrast, *declarative programming* languages focus on what the program should accomplish, without explicitly specifying how to do it. *Functional programming* is a particular kind of declarative programming where the computation of a value is treated as the evaluation of a mathematical function. Some modern languages implements this idea (e.g. Haskell and ML).

The λ-calculus was introduced by Alonzo Church and it is the abstract mathematical representation of the functional programming languages. It was shown that it is Turing-complete [105], so that theoretically any algorithm can be written with it.

Each term in a λ-calculus expression is known as a λ-*term*. There exists three kind of λ-terms:

- variables;
- λ-*abstractions* denoted as $\lambda x.Y$, where x is a variable and Y a λ-term;
- λ-*applications* denoted by XY, where X and Y are λ-terms.

A λ-abstraction models an anonymous function with a single input parameter. For instance, $\lambda x.(2x)$ is the λ-calculus equivalent of the function $f(x) = 2x$ and $\lambda x.(\lambda y.(x+y))$ is the equivalent of $f(x,y) = x + y$.

A λ-application instead represents the instantiation of a λ-abstraction with a particular input. For instance, $(\lambda x.(2x))3 = 6$ since it means $f(3) = 2 \cdot 3 = 6$.

The λ-calculus might be typed or untyped. In a *typed λ-calculus*, each λ-term is tagged with a specific *type* and, from now on, only typed systems will be described. Typed systems are much complex than their untyped counterparts, but they offer several remarkable relationships with the programming languages world.

A relatively simple, yet not trivial, type system is the Simply Typed λ-Calculus (STLC), denoted by the symbol $\lambda^{\rightarrow}$. More complex typed systems can be created by extending it.

If τ indicates any basic type, then the following BNF rule can be used to describe the types of a $\lambda^{\rightarrow}$-term:

$$t ::= \tau \mid t \text{ ``}\Rightarrow\text{''} t.$$

If c indicate any constant and x any variable, a $\lambda^{\rightarrow}$-term can be defined by the rule:

$$s ::= c \mid x \mid s\ s \mid \text{``}\lambda\text{''} x \text{ ``:'' } t \text{ ``.'' } s.$$

With respect to the untyped λ-calculus, the $\lambda^{\rightarrow}$-applications are tagged with a type. For instance, $\lambda x : t.(2x)$ can be used only when the variable x has type t.

In a sequent, the notation $x : X$ indicates a *typed context*, that is a variable x paired with a type X. For instance the sequent $x_1 : X_1 \vdash x_2 : X_2$ means that if x_1 has type X_1 then x_2 has type X_2. This allows to express inference rules for the types.

The following rules can be used to infer the types of a term:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\,var \qquad \frac{\Gamma \vdash X : (T_1 \Rightarrow T_2) \quad \Gamma \vdash Y : T_1}{\Gamma \vdash XY : T_2}\,app \qquad \frac{\Gamma, x : T_1 \vdash Y : T_2}{\Gamma \vdash (\lambda x : T_1.Y) : (T_1 \Rightarrow T_2)}\,abs.$$

In other words:

- the rule *var* states that if we know that the variable x has type T in the context Γ, then we deduce that x has always type T;
- the rule *app* states that if X has type $T_1 \Rightarrow T_2$ and Y has type T_1, then the application XY has type T_2;
- the rule *abs* states that if the variable x has type T_1 and Y has type T_2, then $\lambda x : T_1.Y$ has type $T_1 \Rightarrow T_2$.

Basically, if the proof of a $\lambda^{\rightarrow}$ program can be demonstrated, then the program is type-correct. For instance, let suppose to have a $\lambda^{\rightarrow}$ program where:

- there are only two basic types: *int* (the integers) and *bool* (the booleans);
- a boolean variable can assume the values $true$ and $false$;
- an integer variable can assume any natural number $\mathbb{N}$.

Let consider the program:

$$((\lambda x : int.(\lambda y : bool.x))3)false.$$

By performing the substitution in x and then in y, it becomes $(\lambda y : bool.3)false$ and finally only 3. The result type is obviously an *int*, since this program map the function $f(x,y) = x$ where x is an integer and y a boolean. Formally, the goal is to prove that $\vdash ((\lambda x : int.(\lambda y : bool.x))3)false : int$. This can be accomplished via the following proof tree:

$$\mathit{app}\,\dfrac{\mathit{app}\,\dfrac{\mathit{abs}\,\dfrac{\mathit{abs}\,\dfrac{\mathit{var}\,\dfrac{\dfrac{}{\vdash x : int} \quad \dfrac{}{\vdash y : bool}}{x : int, y : bool \vdash x : int}}{x : int \vdash (\lambda y : bool.x) : bool \Rightarrow int}}{\vdash \lambda x : int.(\lambda y : bool.x) : int \Rightarrow bool \Rightarrow int} \quad \dfrac{}{\vdash 3 : int}}{\vdash (\lambda x : int.(\lambda y : bool.x))3 : bool \Rightarrow int} \quad \dfrac{}{\vdash false : bool}}{\vdash ((\lambda x : int.(\lambda y : bool.x))3)false : int}.$$

An interesting fact about the traditional functional programming in general is that there are no side effects[1] involved. This allows to represent in a very elegant and simple way proofs, but it has a number of drawbacks:

- interactions with the real world need side effects (e.g. I/O from files, streams and sockets);
- directly accessing an array cell also needs side effects;
- iteration cannot be done in the classical λ-calculus since requires side effects, but it can be simulated through recursion.

To mitigate these issues, most of the modern functional programming languages supports some form of side effects (e.g. Haskell and Scheme).

There are a number of Curry-Howard correspondences between several typed λ-calculi and some logic families. For instance, the $\lambda^{\rightarrow}$ corresponds to a fragment of IL known as *minimal logic*.

[1] In this context, a *side effect* is a change in a program state that does not depend on a function input parameters. A function without side effects called twice with the same parameters will perform the same tasks twice, while a function with some side effects can perform different jobs even if executed with the same input arguments.

3.1.5 Dual light affine logic

Dual Light Affine Logic (DLAL) is a typed λ-calculus system that is a simplification of LL proposed by Baillot and Teruib [104] in 2008. The types in DLAL can be expressed by means of the BNF rule:

$$t ::= \tau \mid t \text{ “}\multimap\text{” } t \mid t \text{ “}\Rightarrow\text{” } t \mid \text{“}\forall\text{”} \tau \text{ “.” } t \mid \text{“§”} t.$$

Where τ represent any basic type. The linear implication $\multimap$ has the same semantic as in LL, while the logical implication $\Rightarrow$ is intuitionistic. The of course ! and why not ? operators are replaced by the *neutral* operator § (also known as the *paragraph* operator). Intuitively, a §-rule may use multiple occurrences of its argument without detailing if the number of occurrences is finite or not.

The proof rules for DLAL are presented in Appendix A.4.

By the Curry-Howard correspondence, the typeability proofs in DLAL represents a program. In addition, it can be shown that any program in DLAL has FPTime complexity. Non-FPTime algorithms cannot be expressed in DLAL since it lacks the semantic power to do so. That means that all the ECC algorithms listed in Chapter 2 can, theoretically, be represented as λ-terms in DLAL since they model a function $f : \mathbb{F}_2^* \mapsto \mathbb{F}_2^*$.

3.1.6 Typeable functional assembly

The Typeable Functional Assembly (TFA) logic family was introduced by Cesena, Pedicini, and Roversi [101] in 2011. TFA is a slight variant of DLAL, introduced to simplify the use of several programming patterns that are quite common in the binary field algorithms. In general, λ-terms in TFA that work on bits and array of bits recall a style similar to programming a Turing machine.

The types in TFA can be expressed by means of the same formula of DLAL. The only difference is in the proof rules, available in Appendix A.5, which simplifies the typing in this logic family w.r.t. DLAL.

3.2 Multiplicative inversion in TFA

One of the multiplicative inversion that are implemented in this dissertation's companion project is DCEA. The basic ideas on which it is built are derived from the translation of BEA (Algorithm 2.8) in TFA. This section starts by introducing how the Binary Euclidean algorithm can be rewritten as a purely functional λ-term and concludes with the description of DCEA.

Only the most relevant parts of the λ-terms are presented. A more complete description is available in «Can a Light Typing Discipline Be Compatible with an Efficient Implementation of Finite Fields Inversion?» [18] and «Light combinators for finite fields arithmetic» [19].

3.2.1 Data representations

Booleans (i.e. bits) in λ-calculus can be represented via the so-called *Church booleans*, that is:

$$False = 0 = \lambda x.\lambda y.x$$
$$True = 1 = \lambda x.\lambda y.y$$

In the following sections, for the sake of brevity, the usual Arabic number notation will be used instead of writing the full λ-term of a Church boolean.

Tuples (or lists) can be modeled recursively by:

$$
\begin{aligned}
\varnothing &= \lambda f.\lambda x.x \\
(\, a_0\,) &= \lambda f.\lambda x.(f\ a_0\ x) \\
(\, a_1, a_0\,) &= \lambda f.\lambda x.(f\ a_1\ (f\ a_0\ x)) \\
&\vdots \\
(\, a_{n-1}, \dots, a_0\,) &= \lambda f.\lambda x.(\underbrace{f\ a_{n-1}\ (f\ a_{n-2}\ (f\ a_{n-3}\ (\dots}_{n \text{ times } f}\ x)))).
\end{aligned}
$$

As before, by abuse of notation, the representation $(\, a_{n-1}, \dots, a_0\,)$ will be used to indicate the λ-term derived from the normal tuple definition.

A *Church word* is a tuple of bits (i.e. a bit vector) written as a λ-term. For instance, the tuple $(\,1,0,1,1\,)_2$ can be modeled as:

$$\lambda f.\lambda x.(f\ 1\ (f\ 0\ (f\ 1\ (f\ 1\ x)))).$$

In this context, each element of a Church word (a bit) can have three values: 0, 1 and $\perp$. The bottom value $\perp$ simplifies the programming of some functions. By convention, the leftmost bit in a Church word is the MSB and the rightmost one is the LSB.

A bit matrix is a tuple of Church words, hence its content can be specified row by row. Similarly, the notation $\langle a_{n-1}, \dots, 0\rangle$ will be used do denote a set of *threaded words*, that is a bit matrix where the columns contains the values specified in the tuples $a_{n-1}, \dots, 0$.

3.2.2 Basic operations

The BEA essentially need to perform two basic bit operations: the exclusive or and the right shift of 1 position.

The exclusive or is the simplest one and the bit-to-bit version can be described by the λ-term Xor defined as:

$$
\begin{aligned}
Not &= \lambda x.\lambda y.\lambda z.(x\ z\ y) \\
Xor &= \lambda x.\lambda y.(x\ (Not\ y)\ y).
\end{aligned}
$$

For example:

$$
\begin{aligned}
Xor\ 0\ 1 &= (\lambda x.\lambda y.(x\ (Not\ y)\ y))\ 0\ 1 = \\
&= 0\ (Not\ 1)\ 1 = \\
&= 0\ ((\lambda x.\lambda y.\lambda z.(x\ z\ y))\ 1)\ 1 = \\
&= 0\ ((\lambda y.\lambda z.(1\ z\ y))\ 1)\ 1 = \\
&= 0\ ((\lambda y.\lambda z.y)\ 1)\ 1 = \\
&= 0\ 1\ 1 = \\
&= 1.
\end{aligned}
$$

In λ-calculus, a tuple can be recursively defined as empty or consisting of its first element (called the *head*) and another list (called the *tail*). By recursively splitting a list into heads and tails, it can be preprocessed element by element. For instance, to extract the head and the tail of a tuple, the following functions can be used:

$$\begin{aligned} Head &= \lambda x.(x\ True) & &\text{retrieves the head of the tuple } x \\ Tail &= \lambda x.(x\ False) & &\text{retrieves the tail of the list } x. \end{aligned}$$

The bitwise exclusive or can be obtained by means of the $Map[F]$ meta-function that applies a function F to all the elements of a list. Its code is not listed here since it is well known in the current literature [103].

The right shift of 1 position is more complex and in this case is performed on an entire bit matrix of threaded words. One requirement is that only some columns must be shifted, while some others must remain unaffected. The core idea is to append a new row of zeros at the end, reverse vertically the matrix via to the *wRev* function [18, 103], delete the last row and reverse the matrix again. This has the effect to right shift all the columns, so in order to avoid the modification of a certain threaded word a slightly different technique must be used. An example of the actual approach is sketched in Figure 3.1. Note that performing a right shift, in this context, means shifting a column's content upward.

$$\begin{array}{ccc} a & a' & b \\ \hline 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

(a) Initial setup.

$$\begin{array}{cccc} a & & a' & b \\ \hline 0 & & 0 & 1 \\ & \swarrow & & \\ 0 & & 1 & 1 \\ & \swarrow & & \\ 1 & & 0 & 0 \\ & \swarrow & & \\ 0 & & 1 & 1 \\ & \swarrow & & \\ 1 & & 0 & 0 \end{array}$$

(b) Extended matrix.

$$\begin{array}{ccc} a & a' & b \\ \hline 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ \cancel{0} & \cancel{0} & \cancel{1} \end{array}$$

(c) Reversed matrix.

$$\begin{array}{ccc} a & a' & b \\ \hline 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{array}$$

(d) Shifted matrix.

Figure 3.1: Example of functional right shift.

Let suppose that the matrix has three columns and four rows (Figure 3.1a). The first column a must not be affected by the right shift, the second column a' is used as a support for 'canceling' the effect of the matrix right shift on a and the third column b must be shifted. It is important to observe that, due to the λ-calculus constraints, the matrix can be iterated only in a particular direction. Each step effectively receives an input matrix and creates an output matrix. The notations a_{in} and a_{out} respectively denote the column a extracted from the input or output matrix. To perform the right shift, the following steps are then executed:

1$^{\gg}$. the output matrix is extended by adding one new row at the end, then $a'_{out} \leftarrow (\, a_{in}, 0\,)$, $a_{out} \leftarrow (\, 0, \text{the first four rows of } a'_{out}\,)$ and $b_{out} \leftarrow (\, b_{in}, 0\,)$ (see Figure 3.1b);

2$^{\gg}$. the matrix is reversed vertically and the last row is deleted (see Figure 3.1c);

3$^{\gg}$. the matrix is reversed vertically again, thus obtaining the final result (see Figure 3.1d). Note that the a column is unchanged while b is correctly shifted (at the beginning it was $(\,1,0,1,1\,)_2$ and at the end is $(\,0,1,0,1\,)_2$).

3.2.3 Functional BEA

If the modulus of the binary field is $m = (\, m_{n-1}, \ldots, m_0\,)$, then the main λ-term of BEA in TFA can be written as:

$$\begin{aligned}wInv =\,&\lambda u.(wProj(d\\ &\quad(\lambda w.wRevInit(BkwVst(wRev(FwdVst\,tw)))))\\ &\quad\quad(MapThread[\lambda u.\lambda v.\lambda g_1.\lambda g_2.\langle u, v, g_1, g_2, m, stop, b_0, b_1, v', g_2', m'\rangle]\\ &\quad\quad\quad u\\ &\quad\quad\quad m\\ &\quad\quad\quad (\,0, \ldots, 0, 1\,)\\ &\quad\quad\quad (\,0, \ldots, 0\,)\\ &\quad\quad\quad m\\ &\quad\quad\quad (\,0, \ldots, 0\,)\\ &\quad\quad\quad (\,\bot, \ldots, \bot\,)\\ &\quad\quad\quad (\,\bot, \ldots, \bot\,)\\ &\quad\quad\quad (\,0, \ldots, 0\,)\\ &\quad\quad\quad (\,0, \ldots, 0\,)\\ &\quad\quad\quad (\,0, \ldots, 0\,))).\end{aligned} \tag{3.1}$$

The λ-term $wInv$ make use of the same variables u, v, g_1 and g_2 used in Algorithm 2.8. It is a function that works on a polynomial $a \in \mathbb{F}_{2^n}$ and reduces to its multiplicative inverse, so that $wInv\ a = a^{-1}$.

The imperative version of BEA works by using four bit vectors u, v, g_1 and g_2. Its functional counterpart $wInv$ is more complex and works on a bit matrix consisting of eleven columns whose initial setup is:

$$\begin{array}{lllllllllllll}\lambda f.\lambda x.(f & (u_0, & m_0, & 1, & 0, & m_0, & 0, & \bot, & \bot, & 0, & 0, & 0) & \\ \quad\quad\quad (f & (u_1, & m_1, & 0, & 0, & m_1, & 0, & \bot, & \bot, & 0, & 0, & 0) & \\ \quad\quad\quad (f & (u_2, & m_2, & 0, & 0, & m_2, & 0, & \bot, & \bot, & 0, & 0, & 0) & \\ \quad\quad\quad (f & (u_3, & m_3, & 0, & 0, & m_3, & 0, & \bot, & \bot, & 0, & 0, & 0) & \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ & & & & & & & & & & & & x\ldots)))).\end{array}$$

This matrix is represented by the set of threaded words:

$$\langle u, v, g_1, g_2, m, stop, b_0, b_1, v', g_2', m'\rangle.$$

From left to right, the columns are:

- u, v, g_1 and g_2, representing the same named variable in the classical BEA;
- m, that is the field modulus;
- $stop$, used to indicate that the multiplicative inverse has been found (or not);
- b_0 and b_1, specify a sort of state, essentially containing the result of the checks if $u[0] \overset{?}{=} 0$ and $g_1[0] \overset{?}{=} 0$;

- v', g'_2 and m', that are three support variables for 'invalidating' the global right shift on v, g_2 and m, so that only u and g_1 are actually changed.

The $MapThread$ meta-function is used to build the initial matrix and this setup phase is quite clear in Equation 3.1.

In an imperative language exiting from a loop as soon as required is often trivial (e.g. via the **break** instruction in C), but in TFA this can not be done so easily. The value d in Equation 3.1 amounts to n^2, that is the square of the field $\mathbb{F}_{2^n}$'s degree. It represents the maximum number of passes that the functional term must perform to achieve its job. In other words, $wInv$ always executes $d = n^2$ iterations, even if fewer steps can be used to compute the inverse.

The basic idea of $wInv$ is to work on the threaded words simulating the iteration through recursion. At each pass the following tasks are executed:

1. the $FwdVst$ function updates the threaded words performing a forward visit, if needed;
2. the $wRev$ function reverses the matrix from top to bottom (a sort of 'vertical' transposition);
3. the $BkwVst$ function updates the threaded words performing a backward visit (since the matrix is now reversed), if needed;
4. the $wRevInit$ function works as the $wRev$, but in addition reinitializes some bits in the matrix.

During a *forward visit*, $wInv$ accesses the columns in the matrix iterating from the LSBs bit to the MSBs, and vice-versa during a *backward visit*.

In a forward visit, the $FwdVst$ function essentially performs one of the following mutually exclusive steps:

$1^{\downarrow}$. if $stop[0] = 1$ then $FwdVst$ does nothing since the algorithm has already computed the inverse;

$2^{\downarrow}$. if $u[0] = 0 \wedge g_1[0] = 0$, the step $1^{\gg}$ is performed in order to avoid changes to the columns u, g_1 and m;

$3^{\downarrow}$. if $u[0] = 0 \wedge g_1[0] = 1$, the column m is added to the column g_1 (via an exclusive or) and the step $1^{\gg}$ is executed as before;

$4^{\downarrow}$. if $u = 1$ the inverse has been found and $stop[n-1]$ becomes 1.

After a forward visit, the matrix is reversed by $wRev$ and a backward visit begins by executing one of the following mutually exclusive steps:

$1^{\uparrow}$. if $stop[0] = 1$ then $BkwVst$ does nothing since the algorithm has already computed the inverse;

$2^{\uparrow}$. if the steps $2^{\downarrow}$ or $3^{\downarrow}$ were executed (this information is store in the columns b_0 and b_1), then the last row of the matrix is erased, effectively performing the step $2^{\gg}$;

$3^{\uparrow}$. if $u \neq 1 \wedge u[0] = 1$, it sets $u[i]$ with $Xor\ u[i]\ v[i]$ and $g_1[i]$ with $Xor\ g_1[i]\ g_2[i]$ until it finds the least $j \geq 0$ such that $v[j] = 1$ and $u[j] = 0$, thus performing the find last set operations in the traditional BEA. If such j exists it sets $v[i]$ with $Xor\ v[i]\ u[i]$ and $g_2[i]$ with $Xor\ g_2[i]\ g_1[i]$.

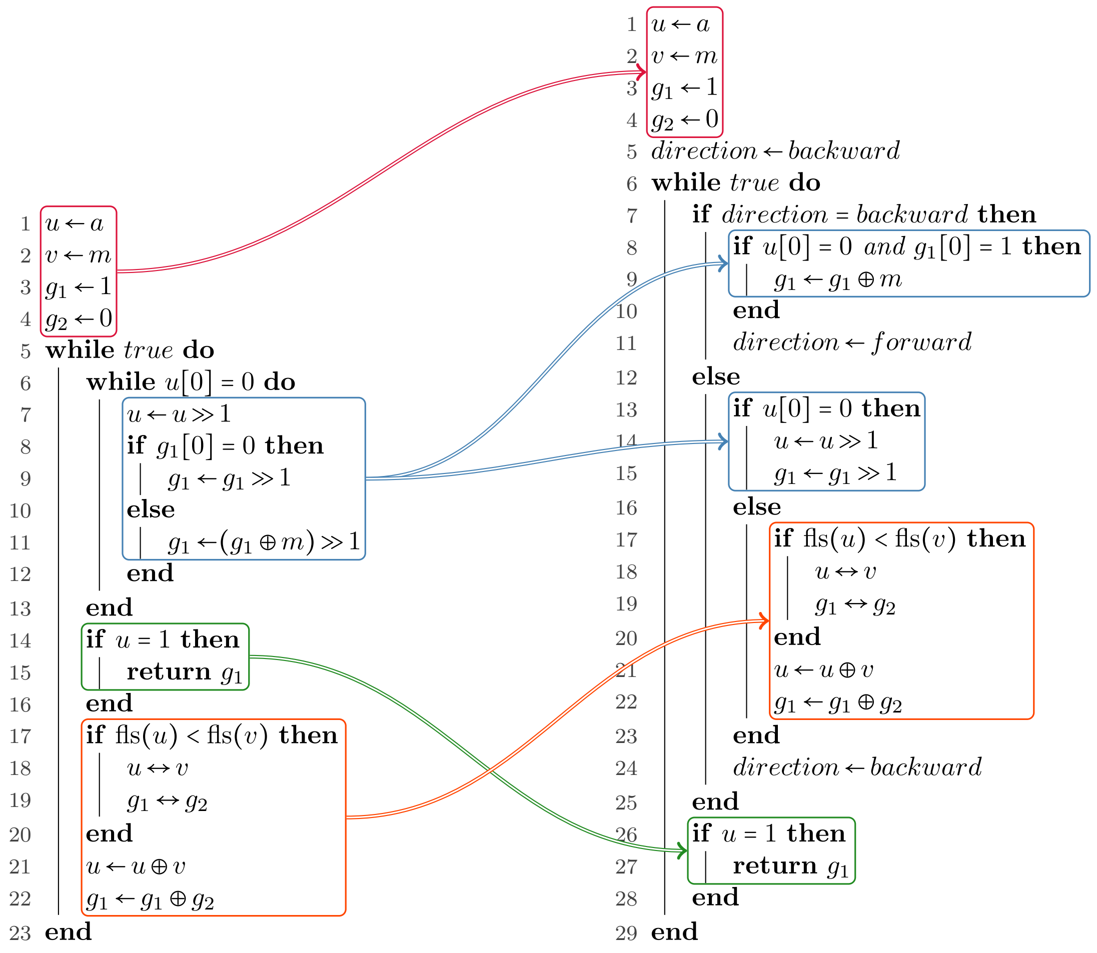
After that the matrix is reversed via the *wRevInit* and another pass might begin.

At the end of all the d^2 iterations, the *wProj* function is used to extract the column g_1 from the threaded words, since it will contain the multiplicative inverse.

Note that writing *wInv* is not enough to prove that it is in TFA. Its type must also deducible from the proof rules of TFA. The discussion of the *wInv* typeability is omitted, but can be found in «Light combinators for finite fields arithmetic» [19].

3.2.4 DCEA

The DLAL Certified Euclidean Algorithm is an imperative version of the λ-term *wInv* (Equation 3.1) that leverages its core ideas. Figure 3.2 shows the BEA and the DCEA side by side emphasizing the similarities between the two algorithms.



```
$u \leftarrow a$
$v \leftarrow m$
$g_1 \leftarrow 1$
$g_2 \leftarrow 0$
while $true$ do
    while $u[0] = 0$ do
        $u \leftarrow u \gg 1$
        if $g_1[0] = 0$ then
            $g_1 \leftarrow g_1 \gg 1$
        else
            $g_1 \leftarrow (g_1 \oplus m) \gg 1$
        end
    end
    if $u = 1$ then
        return $g_1$
    end
    if $\text{fls}(u) < \text{fls}(v)$ then
        $u \leftrightarrow v$
        $g_1 \leftrightarrow g_2$
    end
    $u \leftarrow u \oplus v$
    $g_1 \leftarrow g_1 \oplus g_2$
end
```

(a) BEA.

```
$u \leftarrow a$
$v \leftarrow m$
$g_1 \leftarrow 1$
$g_2 \leftarrow 0$
$direction \leftarrow backward$
while $true$ do
    if $direction = backward$ then
        if $u[0] = 0$ and $g_1[0] = 1$ then
            $g_1 \leftarrow g_1 \oplus m$
        end
        $direction \leftarrow forward$
    else
        if $u[0] = 0$ then
            $u \leftarrow u \gg 1$
            $g_1 \leftarrow g_1 \gg 1$
        else
            if $\text{fls}(u) < \text{fls}(v)$ then
                $u \leftrightarrow v$
                $g_1 \leftrightarrow g_2$
            end
            $u \leftarrow u \oplus v$
            $g_1 \leftarrow g_1 \oplus g_2$
        end
        $direction \leftarrow backward$
    end
    if $u = 1$ then
        return $g_1$
    end
end
```

(b) DCEA.

Figure 3.2: BEA and DCEA in comparison.

The main differences are that the DCEA includes only one loop, while the BEA has two nested iterations, and a slightly different order of operations. In the pseudo-code of Figure 3.2b the *direction* variable is used to emulate the forward and backward visits in *wInv*. The right shift is much easier to do in an imperative environment, so that it is completely performed during the forward visit. Note that the check on the *direction* variable can be safely removed, since each loop iteration executes a forward and a backward visit.

It is easy to prove that the two algorithms are equivalent and have the same complexity. Nevertheless, a different ordering and categorization of the instructions can allow a compiler to perform different types of code improvements, potentially generating a faster implementation. Chapter 7 details the performance of all the multiplication inversion algorithms described in Chapter 2, including DCEA.

Chapter 4

Architecture

Some people think technology has the answers.

Kevin Mitnick

The framework (named CryptoStudio) is an extensible software system that can be used to produce a wide array of optimized cryptographic functions. In spite of its flexibility, this dissertation focuses solely on generating high speed ECC code.

We can distinguish three main types of users that may want to use it to achieve different goals:

- the *end-user* is interested only in producing a final library for some platform — he is not required to know the framework internal structure or to be a cryptography expert;
- the *cryptography developer* that wants to add new cryptographic algorithms to the system;
- the *framework developer* whose goal is to add new features to the framework, such as a new optimization pass to the compiler.

The simplified architecture of the framework is sketched in Figure 4.1.

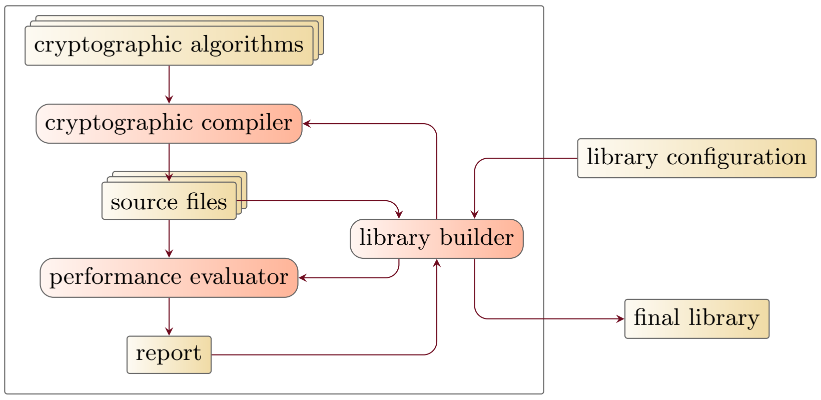


Figure 4.1: General architecture of the framework.

From the end-user point-of-view, the whole system can be viewed as a black box that produces the final optimized library starting from a configuration file, describing its content. The only manual action that the user has to do is write the configuration file, while the rest is handled by the framework. Since these files usually have the same basic structure, some ready-to-use templates are provided as a commodity. Under the hood, however, the framework internal mechanisms are handled by three main components: the cryptographic compiler, the performance evaluator and the library builder.

The *cryptographic compiler* is by far the most important and complex module of the whole infrastructure. It is a fully fledged optimizing compiler used to produce a set of source files implementing the chosen cryptographic primitives. The cryptographic compiler translates an algorithm written in a hardware independent language into an ad-hoc optimized version for a specific mathematical structure and processor. Actually the only supported output language is C, but its structure is flexible enough to easily accommodate other programming languages. It comes with a stock of already written, and tested, cryptographic algorithms supporting the most common operations for the binary fields and elliptic curves.

Once the source files have been generated, their are execution times are measured by the means of a *performance evaluator*. Its job is to assess the running time of the produced cryptographic functions with the greatest accuracy possible. This is performed by actually compiling the code, running it several times and extracting an estimation via a statistical analysis of the results. The output of this software component is a report stating the function execution times with various compilation flags, so that also some suitable compilation options can be chosen when producing the library.

Finally, the *library builder* is in charge of collecting the outputs of the other two tools and effectively creating the final cryptographic library. It achieves this goal with a two phase approach. First, it generates the makefiles and scripts needed to compile the library from scratch. Second, it uses the evaluation report for picking the best (fastest) implementations from the generated source files pool. This last step is needed since the same cryptographic primitive can be (frequently) implemented by a number of algorithms and/or with different execution parameters (e.g. the size of a sliding window). The performance evaluation step can be skipped if there is only one possible implementation for each cryptographic primitive.

4.1 Code compilation

The cryptographic compiler tool (named CryptoGen) represents the core component of the framework and its task is to produce a set of highly optimized and specialized source files starting from an algorithm written in a very high level hardware independent language.

The diagram in Figure 4.2 sketches the main building blocks of the cryptographic compiler.

This component was designed by leveraging several paradigms behind the compiler theory. From an architectural point-of-view, it can be split in three sub-components that are always executed sequentially:

1. the *front-end stage* is in charge of translating the input algorithm into an intermediate representation known as HR, more suitable for the optimization process. In addition, this phase also performs various lexical, syntactical and semantic checks to ensure that only correct algorithms are compilable.

2. the *middle-end stage* is where most of the optimizations take place. Note that since the HR is hardware independent, these optimizations cannot include some machine specific idioms. If producing optimized code is not required this phase can be skipped.

3. the *back-end stage* takes in input the (unoptimized or optimized) HR and produces the final source files. A back-end implementation can optionally include some additional optimization

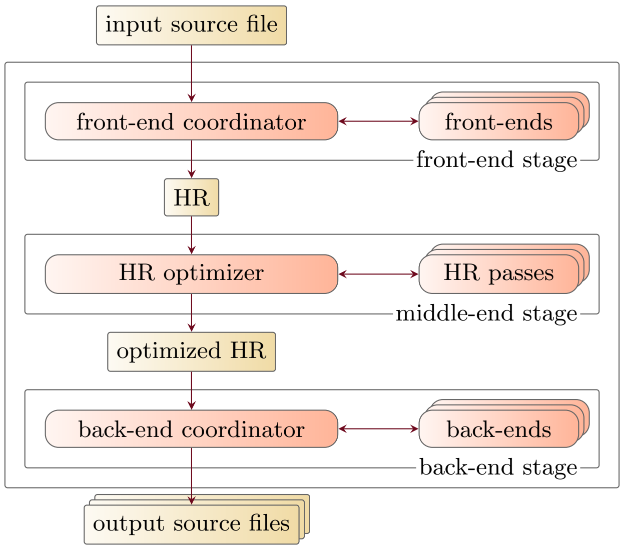


Figure 4.2: Architecture of the cryptographic compiler.

passes that may exploit some hardware specific features (e.g. by using special assembler instructions of the target machine).

This architecture allows a very flexible and modular approach to the compilation process, allowing to easily switch between front-ends, back-ends and adding new optimization passes.

4.1.1 Front-end stage

The front-end stage main job is to translate an input algorithm into an equivalent HR model, eventually performing some sanity checks on the input source file.

A *front-end coordinator* is in charge of selecting the right *front-end* module in order to perform the checks and the HR translation. The selection can be actually performed in two different ways:, by looking at the input source file extension or by forcing the compiler to use a specific front-end via a special API call.

The output of the front-end stage consists of:

- the list of errors found in the input source file — it is empty if the input code is lexically, syntactically and semantically correct.
- the list of warnings found in the input source file;
- the HR translation of the input source file. It is blank only if the error list is non-empty.

4.1.2 Middle-end stage

The middle-end stage is in charge of the optimizations. The optimizations are enabled and configured directly in the code by a series of special annotations (Appendix C). This phase is optional, and, if skipped, the HR will be left untouched and unoptimized.

All the optimization passes can be categorized in two main types:

- the *analysis pass*es are used to perform some kind of inspection on the code. They fill some data structure in the memory and do not alter the code.

- the *transformation passes* are instead used to actually modify the code in order to boost its performance. A transformation pass can also not modify the code is some constrain is not matched (e.g. a dead code elimination pass will not remove anything if all the code is live).

All the available passes are described in detail in Chapter 6.

The middle-end stage establishes a partial ordering between the optimizations since a pass may require that some other passes must be completed before its execution. This is the typical case of a transformation optimization that relies on the results previously gathered by some analysis pass. The framework developer only needs to specify these precedence relationships and the compiler will take care of choosing an appropriate order when performing the middle-end stage optimizations.

Note that, since the HR is machine independent, some code-improving techniques (e.g. use of assembler instructions) cannot be applied and are hence delegated to the back-end stage.

4.1.3 Back-end stage

The back-end stage is the last step in the code generation chain. It takes the HR, optionally performs some hardware dependent optimizations and produces the final code. In spite of supporting a multiple back-end system, only the C language one is actually implemented. The Figure 4.3 shows its internal architecture.

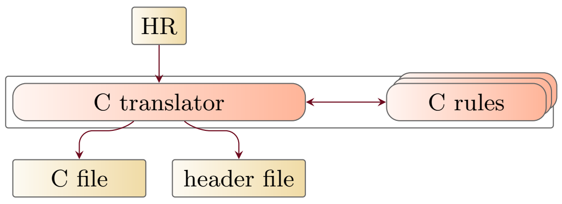


Figure 4.3: Architecture of the C back-end.

The C back-end core component is a C translator that iteratively executes a set of translation rules to produce the final output files, that are a C implementation and header files. The translation rules can be added via the Eclipse extension paradigm and can be classified in two main categories:

- data type rules are used to translate HR data types into C data types;
- instruction rules are instead used to translate a single HR into one or more C instructions.

Note that each back-end is free to implement any kind of work-flow and the aforementioned one is only one possible scenario.

4.2 Performance evaluation

In this thesis context, measuring the running time of a code region is important for a least two reasons. First, as stated before, the time measures are used to discriminate the best algorithm implementations from the slowest ones. Second, this information can provide the user an interesting report in order to check how fast is a cryptographic library (e.g. for comparing its performance against another implementation).

Measuring with great accuracy the execution time of some code is not as easy as it seems, especially on high-end modern hardware platforms. Several factors can effectively influence the quality of a measurement, such as:

- implicit delays induced by the measurement function calls (e.g. due to the call assembler instruction itself or stack pushes/pops);
- the measurement functions may be called too early or too late (e.g. due to an out-of-order execution of a CPU core);
- the measured function is temporarily stopped by the operating system (e.g. due to a kernel preemption or due to an interrupt service routine execution).

Most of these issues can be mitigated, but not completely removed, so that a statistical approach computed over a set of repeated measures is mandatory to achieve a stable result.

The performance evaluator (named Chronon) is implemented through a plug-in system that is able to choose the best measurement strategy according to the target hardware platform. Actually, all the implemented measurement functions make use of the generic template listed in Algorithm 4.1.

```
algorithm MEASURE(a function $f$) return the execution time of $f$

$T \leftarrow \varnothing$
for $i \leftarrow 1$ to $p$ do
    $start \leftarrow$ current time
    for $s$ times do
        call $f$ with some random parameters
    end
    $end \leftarrow$ current time
    $T[i] \leftarrow (end - start)/p$
end
$q_1 \leftarrow$ compute the lower quartile of $T$
$q_3 \leftarrow$ compute the upper quartile of $T$
foreach $t \in T$ do
    if $t \notin [q_1 - \theta(q_3 - q_1), q_3 + \theta(q_3 - q_1)]$ then
        remove $t$ from $T$
    end
end
return the mean of $T$
```

Algorithm 4.1: Generic measurement algorithm.

The algorithm executes the measurement in four consecutive steps:

1. the arithmetic means for a set of p populations containing s samples is computed;
2. the lower and upper quartiles are computer over the p means;
3. the outliers are discarded by using the interquartile range [106] with a *sensitivity* constant $0 \leq \theta \leq 1$ (the lower the constant the stricter the analysis);
4. the final value is computed as the grand mean of the surviving means.

Currently, the tool supports two functions to acquire the current time:

- the `clock_gettime()` function for computing the execution time in nanoseconds. This is a POSIX compliant function available on most of the Linux and BSD modern systems.

- the *rdtsc* and *rdtscp* assembler instructions for estimating the execution time in clock ticks [107]. These instructions are available on most of the modern AMD64 processors. This strategy was partially inspired by the SUPERCOP toolkit [108]. If available, this approach is much more reliable and accurate than the aforementioned one.

4.3 Library building

The *library builder* (named CodePack) is the component in charge of creating the final library. It can be though both as a high level front-end of the whole framework for the end-user, but also as a system orchestrator whose task is executing the other framework components in the right order with the right parameters.

The builder internal structure is relative simple and straightforward. Its only input is a configuration file specifying the cryptographic algorithms that must be included in the final library and their parameters, if any. Once launched, the tool will parse the configuration file and, by means of the cryptographic compiler and the performance evaluator tools, will produce the final library. Its structure was developed with extensibility in mind and is able to handle several programming languages, compilers and compilation tool-chains via a modular approach, although actually it handles only the C programming language.

The Figure 4.4 shows a very simple example configuration file (in XML) that will create a library containing the modular reduction and multiplication for $\mathbb{F}_{2^{163}}$. Note that the final multiplication implementation will be selected as the fastest one between two combs (Algorithm 2.3) with a window width of 3 bit and 4 bit.

The library builder can be used in two different modalities:

- in the *on-line* mode, it immediately generates and evaluates the code, effectively allowing the end-user to build a library on-the-fly;

- in the *off-line* mode, CodePack works in two stages. First, it generates the code and a proper makefile. Second, the user manually runs the makefile, potentially on a different machine than the one containing CodePack. This will produce the final library consisting of the fastest implementations.

In spite of requiring some manual effort, the off-line mode is particularly interesting in two scenarios. On one hand, when the user needs very accurate measurements, by running the performance evaluation in a more stable environment (e.g. with no graphical interface and no services running in background). On the other hand, it decouples the code generation from the code evaluation phase, thus allowing to produce a library for an hardware platform different from the one running the framework. For instance, the code generation can be performed on a PC while the final target of the library is a DSP or a micro-controller.

4.4 UI

The framework comes bundled with a complete UI for the Eclipse 4 platform. It consists of:

- some wizards able to guide the user to quickly create a new CryptoStudio project;

- a number of source code editors, one for each supported language (Chapter 5), which fully support keyword highlighting, auto-completion and on-the-fly validation of the code. The compiler is automatically invoked when a source file is saved.

```
<library name="sampleLibrary" xmlns="http://security.polito.it/cryptostudio/codepack/library">
    <operation>
        <implementation name="reduce" main="reduce">
            <unit name="bfReduction">
            <parameter name="BF_DEGREE_">163</parameter>
            <parameter name="BF_EXPONENTS_">(7, 6, 3, 0)</parameter>
            <parameter name="BF_REDUCTION_">reduce</parameter>
            </unit>
        </implementation>
    </operation>
    <operation>
        <implementation name="comb3" main="multiply">
            <unit name="bfMultiplicationComb">
            <parameter name="BF_DEGREE_">163</parameter>
            <parameter name="BF_WINDOW_">3</parameter>
            <parameter name="BF_MULTIPLICATION_">multiply</parameter>
            </unit>
        </implementation>
        <implementation name="comb4" main="multiply">
            <unit name="bfMultiplicationComb">
            <parameter name="BF_DEGREE_">163</parameter>
            <parameter name="BF_WINDOW_">4</parameter>
            <parameter name="BF_MULTIPLICATION_">multiply</parameter>
            </unit>
        </implementation>
    </operation>
</library>
```

Figure 4.4: A sample library configuration file.

- various menus allowing the user to call both the performance evaluator and the library builder;
- several views that can be used to debug the compiler passes (e.g. showing the control flow graph of a function).

The system was developed by leveraging the MVC paradigm, so that the developer can also make use of the framework programmatically via the API instead of the UI.

Chapter 5

Languages and representations

Language is the source of
misunderstandings.

ANTOINE DE SAINT-EXUPERY

Being able to model an algorithm with ease is a vital feature of any compilation tool-chain. In this chapter the supported input languages and their internal representations are discussed.

5.1 Domain specific languages

The current scientific world offers a great variety of languages that can be adopted to specify with clarity and unambiguity how an algorithm works. Most of the modern general purpose programming languages can be chosen to describe a cryptographic primitive, however, by making use of a DSL the developer's job can be greatly simplified, allowing him to write the code with less effort and major clarity.

Even if there exists some (few) cryptography oriented DSLs, for instance CAO [17] and Cryptol [16], two new languages were designed in order to better capture the features offered by the framework, namely:

- HRL, an assembler-like DSL;
- aXiom, a high level DSL with a built-in syntactic preprocessor.

5.1.1 HRL

HRL is, in a nutshell, the textual representation of the HR used internally by the cryptographic compiler (Sections 4.1 and 5.2).

It was primarily developed for testing purposes, but, nonetheless, it is a fully fledged programming language and can be used to write very complex algorithms. In spite of lacking several high level constructs (e.g. `for` loops), HRL and aXiom offer nearly the same expressive power. If an algorithm can be implemented in aXiom it can also be written in HRL. Its grammar in BNF form is given in Appendix B.1.

It is an imperative, procedural, strongly typed and hardware independent programming language. It has virtually no syntactic sugar, providing a very restricted instruction set. It has two main restrictions:

- it follows the *three-address code* paradigm [109], that is each HRL instruction can have at most three operands: one for the output and two for the inputs. This affects the code writing since the more complex instructions must be break down into simpler statements. For instance, the expression $x \leftarrow y + z + w$ can be rewritten as $t \leftarrow y + z$ and $x \leftarrow t + w$.

- Jumps and branches can only be expressed by **`goto`**s.

Several modern optimizing compilers make use of similar representations in order to significantly simplify the code analysis and transformation algorithms, such as GIMPLE [110], one of GCC's internal representations.

HRL supports only two basic data types:

- the *logical* type, that is the boolean type, identified by the **`logical`** keyword;

- the *relative* type, specifying a relative number in $\mathbb{Z}$ (a signed integer). They are declared specifying a range via the syntax `[x, y]`, allowing a variable to hold any integer number in the range from x to y (bounds included).

Knowing the integer range of a variable is useful for various reasons. First, the compiler can easily compute how many bits are needed to store a value. Second, the DSL validation process can perform stricter checks on the input source file, thus reducing the probability of an error. Third, the optimization process can take advantage of this additional knowledge in order to produce a faster code. For instance, if a variable x can contain only integers in the $[0, 10]$ interval, the expression **if** $x \leq 20$ **then** $y \leftarrow 1$ **else** $y \leftarrow -1$ can be simplified into $y \leftarrow 1$.

The only available type constructor is the *tuple* constructor that allow to create a vector of *`size`* elements of the same type *`type`* via the syntax *`size`* **`of`** *`type`*. The tuple constructor can be used recursively to produce multi-dimensional matrices (e.g. *`size1`* **`of`** *`size2`* **`of`** *`type`*).

Note that a *bit* has the type `[0, 1]` and a *bit vector* has the type *`size`* **`of`** `[0, 1]`. These types are processed in a special way by the C back-end in order to enhance their performance due to their wide use in cryptography (Section 6.3.1).

Apart the name, the data type and an optional initial constant value, every variable (and function parameter) must also specify an I/O *direction*, indicating the allowed and forbidden operations on the data. The available directions are **`ro`**, **`wo`** and **`rw`**, used respectively to specify read-only, write-only and read-and-write variables. Read-only variables are effectively constants since their value cannot change during the code execution. This information is used both to enforce several validation checks and for optimization purposes.

Functions can be declared with any number of parameters and can contain an arbitrary list of operations. The Table 5.1 summarizes all the available instructions, that are:

- the **`nop`** instruction that does nothing. It was introduced primarily as a placeholder for labels, since it simplifies the HR translation phase in the front-ends.

- The **`goto`** and the **`if`**-**`goto`** jump instructions;

- the **`push`** and the **`call`** instructions used to perform function calls. The **`push`** instruction insert a value into a fictitious call stack and the **`call`** operation executes a function by passing all the data stored in the stack, thus allowing to overcome the three-address code constraint.

- The `<-` copy instruction. It works on every data type and allows also to clone mono and multi-dimensional tuples.

- Two special copy instructions that can respectively extract a single element from a tuple and overwrite a tuple element;

- two other copy instructions used to shrink or expand bit vectors, respectively the **reduce** and **extend** instructions;
- a conversion operation, the **relative** instruction, which converts a bit vector into an unsigned relative;
- the usual operators for manipulating bits (and logicals), which are and, inclusive/exclusive or, not, left/right shifts and the find last set;
- various operators for working with integers, allowing to perform additions, subtractions, multiplications, integer divisions, round-up divisions[1], modulo reductions and sign negations;
- the traditional six comparison operators able to evaluate equalities, inequalities and less/-greater than (or equal to) comparisons.

HRL supports also a form of code annotations. Similarly to Java, the *annotations* are meta-data that do not alter the semantic of the code, but provide the compiler additional information so that it can produce better code. All the annotations starts with the : character (semicolon) and can be applied to the whole source file or to a function. They can be used to achieve different effects such as to enable an optimization pass or to change some internal compiler parameter. Two particularly important annotations are the `:export` and `:import` ones. The first is used to declare that a function should be callable from outside of the current compilation unit, while the latter is used to access an exported function defined in another source file (thus allowing the developer to split the implementations in several files). The complete list of annotations can be found in Appendix C.

The Figure 5.1 shows two semantically equivalent code snippets in C and HRL. The implemented algorithm works on a tuple of 79 bits, performs a not and then a right shift of 2 positions. It clearly shows that the HRL code is much more concise and readable (and thus maintainable) than its C counterpart, despite being severely limited in its syntactic sugar.

```c
void f(uint8_t x[10])
{
    int i;

    /* The "x <- not x". */
    for (int i = 0; i < 10; ++i)
        x[i] = ~x[i];
    x[9] &= 127;
    /* The "x <- x >> 2". */
    for (int i = 0; i < 9; ++i)
        x[i] = (x[i] >> 2) ^
            (x[i + 1] << 6);
    x[9] >>= 2;
}
```

(a) Snippet in C.

```
function f
    x in rw 79 of [0, 1]
{
    x <- not x
    x <- x >> 2
}
```

(b) Snippet in HRL.

Figure 5.1: Two equivalent snippets, in C and HRL.

[1]Given two unsigned integers x and y, the *round-up division* computes the value $\lceil x/y \rceil$, while the *integer division* returns $\lfloor x/y \rfloor$.

syntax	name
nop	no operation
goto *x*	unconditional jump
if *x* **goto** *y*	conditional jump
push *x*	call stack push
call *x*	function call
x <- *y*	copy
x[*y*] <- *z*	copy to tuple
x <- *y*[*z*]	copy from tuple
x <- **reduce** *y*	tuple reduction
x <- **extend** *y*	tuple extension
x <- **relative** *y*	relative conversion
x <- *y* **and** *z*	and
x <- *y* **or** *z*	inclusive or
x <- *y* **xor** *z*	exclusive or
x <- **not** *y*	not
x <- *y* << *z*	left shift
x <- *y* >> *z*	right shift
x <- **findlastset** *y*	find last set
x <- *y* + *z*	addition
x <- *y* - *z*	subtraction
x <- *y* * *z*	multiplication
x <- *y* / *z*	integer division
x <- *y* *z*	round-up division
x <- *y* **mod** *z*	modulo reduction
x <- - *y*	negation
x <- *y* = *z*	equal to
x <- *y* /= *z*	not equal to
x <- *y* < *z*	less than
x <- *y* <= *z*	less than or equal to
x <- *y* > *z*	greater than
x <- *y* >= *z*	greater than or equal to

Table 5.1: The instructions available in the HRL language.

5.1.2 aXiom

aXiom is the primary input language of the cryptographic compiler. It was designed to be as close as possible to the mathematical language typically used to describe algorithms via pseudo-code.

Despite bearing some similarities to HRL, it has far less grammatical constraints and boasts a number of new powerful constructs. It is an imperative, procedural and strongly typed programming language that supports multi-operand expressions, various types of loop instructions and a built-in syntactic preprocessor. Its grammar is available in Appendix B.2.

It supports the same data types of HRL (logicals, relatives and tuples), but it offers a number of useful data type aliases that includes:

- **int8**, **int16**, **int32**, **int64** **uint8**, **uint16**, **uint32** and **uint64** for declaring signed or unsigned integers on 8, 16, 32 and 64 bits;

- **int** and **uint** respectively specifying a signed or unsigned integer that can be stored exactly on one machine word;

- **`bit`** referring to a simple bit.

In addition a special pseudo-constant named **`wordsize`** is available and returns the size of a machine word in bits (usually 8, 16, 32 or 64).

aXiom relaxes the constrains imposed by HRL by removing the three-address code limitation, so that complex statements can be easier written, such as `x <- (y or (0, 1, 1)) << (1 + 2)`. This introduces however the necessity to establish a precise order of computation amongst the various operators. Their precedence in aXiom is the same as in most of the programming languages such as C, C++ and Java. As usual in the imperative programming languages, the round parenthesis `(` and `)` can be used to change the calculations' priorities.

In addition, some special operators are available to further simplify the code development. They include:

- the **`size`** operator returns the number of elements in a tuple or one if applied to a non-tuple (scalar) type;
- the **`zeros`**, **`ones`**, **`one`** operators create a tuple respectively containing all zeros, all ones and all zeros except the first element that is one;
- the **`lowones`** and **`highones`** operators create a tuple respectively containing all zeros except the lower or higher elements that are ones;
- the **`tuple`** operator converts a relative value into a tuple of bits;
- the **`min`** and **`max`** operators compute the minimum and maximum value between two relative values;
- the `x[y, z]` operator is used to perform the *tuple slicing*, that is it returns the sub-tuple consisting of the bits extracted from the bit vector `x` from the `y`-th position to the `z`-th position.

In addition, all the binary operators can be transformed into *n-ary variadic operators* by using the **`do`** keyword. For instance, the snippet `do + (1, 2, 3, 4, 5)` is equivalent to `1 + 2 + 3 + 4 + 5`.

aXiom supports also a powerful form of *operator overloading*, allowing the user to redefine the semantic of the traditional symbolic operators (e.g. `+` and `/`), but also to define new named operators. For instance, the developer can declare the named operator `log` and use it via the snippet `x <- log(y)`. A binary operator must be declared as a special function having three parameters (one write-only for the output and two read-only for the inputs), while a unary operator must have two arguments (one write-only for the output and one read-only for the input). For instance, the `log` operator can be defined as `operator log(y in wo uint, x in ro uint)`[2].

The language does not support directly the **`goto`** instruction, which has been replaced by more developer friendly constructs, that are:

- the usual **`if`** and **`if-else`** constructs to perform the branches in a structured way;
- the **`while`** and **`do-while`** loop constructors, working exactly as in C, C++ and Java;

[2]This behavior is due to the fact that in aXiom (and HRL) the functions do not have an explicit return value such as in C or Java.

- the **for** instruction, which can be used to easily create loops with an integer counter ranging over a user-specified interval or to iterate over tuples.

When writing a cryptographic algorithm, the need to create hard-coded look-up tables arises quite frequently. In order to concisely describe this kind of tuples, aXiom supports a powerful feature known as *tuple comprehension* that allows the developer to create arrays using a compact math-like representation without explicitly using a loop. For instance, `(2 * i | i in [0, 99])` creates the tuple containing the first 100 even numbers since it has the same meaning as the mathematical expression $(\, 2i : \forall i \in [0, 99] \,)$.

Figure 5.2 shows two equivalent snippets, in C and aXiom, showing this language's enhanced compactness and clarity.

```c
void f(uint8_t x[10])
{
    int i;
    int j;

    /* The "for 10". */
    for (i = 0; i < 10; ++i)
    {
        /* The "not x". */
        for (int j = 0; j < 10; ++j)
            x[j] = ~x[j];
        x[9] &= 127;
        /* The "(not x) >> 2". */
        for (int j = 0; j < 9; ++j)
            x[j] = (x[j] >> 2) ^
                (x[j + 1] << 6);
        x[9] >>= 2;
    }
}
```

(a) Snippet in C.

```
function f(x in rw 79 of bit)
{
    for 10
        x <- (not x) >> 2
}
```

(b) Snippet in aXiom.

Figure 5.2: Two equivalent snippets, in C and aXiom.

Some (a few, actually) programming languages perform a *preprocessing* phase before the actual compilation, modifying the source code accordingly to some transformation rules. aXiom implements a syntactical preprocessor that allows the developer to perform some code manipulation in a safe way. Note that the C and C++ programming languages include a lexical preprocessor acting quite differently from the aXiom one. A lexical preprocessor works on tokens and strings, while a syntactical one is grammar-aware and manipulates abstract syntax trees.

By working at a grammatical level, the aXiom preprocessor is able to perform additional sanity and type checks that cannot be done relying only on the lexical data. For instance, the expression `4 * M + 1` will evaluate to 12 if `M` is defined as `#define M 2 + 3` in C, while in aXiom will correctly evaluate to 21 since the preprocessor computes `M` beforehand. The aXiom preprocessor is invoked when a construct is prefixed by the special **static** keyword. Actually only three static constructs are supported:

- *static constants* are read-only variables whose value is computed on-the-fly by the preprocessor and substituted in the output HR. They are similar to the C symbolic constants declared through the **#define** directive.

- *Static ifs* are **if** instructions whose condition is computed on-the-fly during the compilation. If it is true only the 'then' branch is included in the HR, otherwise the 'else' branch is included (if present). Their behavior is similar to the C directives **#if** and **#ifdef**.

- *Static fors* are **for** loops whose bounds are computable at compile time. During the translation into HR, they are completely unrolled, allowing a sort of pre-optimization of the code.

The Figure 5.3 shows an aXiom snippet with some static constructs and the output in HRL. Note that the *SIZE* constant has been replaced by its value and that the **static if** and **static for** have disappeared.

```
static SIZE in ro uint8 <- 10 * 7 + 9

function f(x in rw SIZE of bit)
{
    i in ro uint

    static if SIZE mod 2 = 0
    {
        for i in [1, 3]
            x <- x << i
    }
    else
    {
        for i in [2, 4]
            x <- x >> i
    }
}
```

(a) Snippet in aXiom.

```
function f
    x in rw 79 of bit
{
    i in ro uint

    i <- 2
    x <- x >> i
    i <- 3
    x <- x >> i
    i <- 4
    x <- x >> i
}
```

(b) HRL output.

Figure 5.3: An aXiom snippet and its output HRL code.

To further emphasize the expressivity of aXiom, the Figure 5.4 compares the pseudo-code of the modular reduction algorithm (Algorithm 2.1) with its aXiom counterpart.

```
$u \leftarrow a[0 \rightarrow n-1]$
foreach exponent $e_i$ of $r(t)$ do
    $u \leftarrow u \oplus (a \gg n) \ll e_i$
end
$v \leftarrow u[0 \rightarrow n-1]$
foreach exponent $e_i$ of $r(t)$ do
    $v \leftarrow v \oplus (u \gg n) \ll e_i$
end
return $v[0 \rightarrow n-1]$
```

(a) Reduction pseudo-code.

```
u <- a and lowones(N, 2 * N)
static for e in E
    u <- u xor ((a >> N) << e)

v <- u and lowones(N, 2 * N)
static for e in E
    v <- v xor ((u >> N) << e)

out <- v[0, N]
```

(b) Reduction aXiom code.

Figure 5.4: The modular reduction algorithm in aXiom.

5.1.3 Test cases

When developing a new cryptographic algorithm, it is important to make use of an adequate number of tests to validate its correctness. CryptoGen allows a developer to write a test case directly in one of its input languages (HRL or aXiom) via the `:test` function annotation. This annotation accepts a single parameter that can be `success` or `failure` if the function is expected respectively to succeed or to fail. A test function must declare exactly one write-only `logical` parameter that will contain the test result.

This approach allows the developer to quickly build a series of platform-independent test cases in the same language as the cryptographic algorithm itself.

For instance, Listing 5.1 shows a code snippet with two tests in aXiom.

```
:export
function f(out in wo int, in1 in ro int, in2 in ro int)
{
    out <- in1 / in2 + in1 mod in2
}

:test(success)
function test1(result in wo logical)
{
    x in rw int

    f(x, 10, 4)
    result <- x = 4
}

:test(faiilure)
function test2(result in wo logical)
{
    x1 in rw int
    x2 in rw int

    f(x1, 10, 4)
    f(x2, 4, 10)
    result <- x1 = x2
}
```

Listing 5.1: Code snippet containing two test cases.

Once a test case is written, it is the CryptoGen back-end's job to create an output file that will perform the tests and assess their results. For instance, the C back-end will produce a file containing a `main()` function calling all the declared tests and printing the results on the standard output via a number of `printf()` calls.

5.2 HR

During the various compilation stages, a compiler frequently constructs a sequence of semantically equivalent intermediate representations of the source input code.

CryptoGen's intermediate representation is known as High level intermediate Representation (HR). It was designed to be simple and very close to the mathematical language. It is the internal abstract meta-model of HRL (Section 5.1.1) and hence hardware independent. Its UML class diagram is sketched in Figure 5.5.

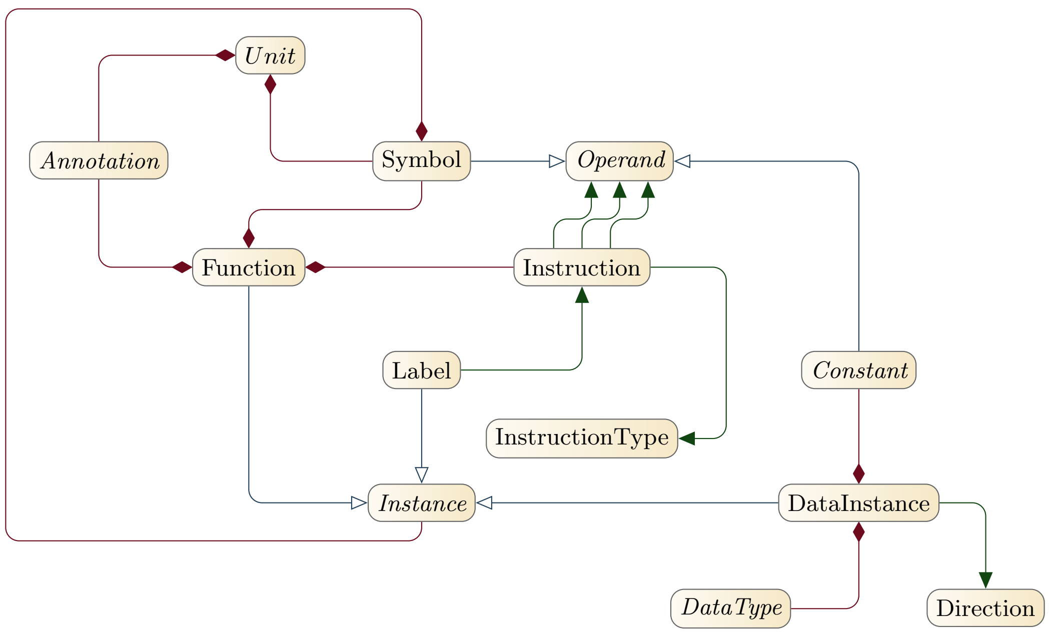


Figure 5.5: UML class diagram of HR.

The root of the meta-model is represented the *unit*, that is the source file, which contains all the data needed to describe the full source code. It acts as a container of symbols and it can be complemented by a number of annotations.

A *symbol* is a named instance of some kind. The *instances* can be distinguished in three categories:

- *data instances*, which represent global/local variables and function parameters;
- *functions* themselves;
- *labels*, which are simply references to a single instruction.

A data instance contains a direction (read-only, write-only or read-and-write) that specifies what I/O is allowed on its content, a proper data type (e.g. relative number) and an optional initialization constant value.

A function contains a list of symbols, instructions and annotations. The symbol list specifies both the function parameters and its local variables. Each instruction instead contains four values: three operands and a value stating the type of instruction, since HR follows the three-address code paradigm. An operand can be a symbol or a constant value. The supported instruction types are listed in Table 5.1.

Chapter 6

Optimizations

If you can't make it good, at least make it look good.

BILL GATES

One of the many advantages that a compiler offers is the ability to automatically produce faster (and/or smaller) code w.r.t. to a manually written one. This feature is particularly tempting when the source code is vast and complex, since a manual optimization process is time consuming and error prone. CryptoGen leverages several optimization techniques taken from the compiler theory field for aiding the developer to produce better code in a simpler way.

In this context, the term *end-compiler* denotes a generic compiler used to compile the code produced by CryptoGen (e.g. GCC). In spite of the modern end-compilers' complexity and smartness, CryptoGen is able to perform a number of optimizations than cannot be performed otherwise. This is mainly due to a lack of semantics in lower level languages such as C, C++ or assembler. When translating some source code written in a high level programming language into a lower level one, often some meta-data is lost, hindering the ability of an optimizer to perform its job. For instance, all the variables in the C programming language are typed, but their assembler counterpart is not. That means that all the data type information is lost, thus denying the ability to perform some data type specific optimizations. CryptoGen, by operating at a higher level, where more semantics is involved, is able to complement the end-compiler's optimizer performing some code transformations that cannot be easily done at the lowest levels.

As introduced in Section 5.1, the input languages managed by CryptoGen support the notion of annotation. The annotations represent the main method through the developer can specify additional meta-data to aid the optimization process and enable/disable certain kind of code-improving techniques.

The algorithms described in the following paragraphs are mostly inspired and adapted from the techniques presented in the following sources:

- Alfred Vaino Aho et al. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 2006. ISBN: 978-0321486813;

- Ken Kennedy and John Randal Allen. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach.* Morgan Kaufmann Publishers Inc., 2001. ISBN: 1-55860-286-0.

This chapter is split into four sections. The first one is devoted to present the analysis algorithms used to correctly apply the code transformations described in the second section. The third section introduces some smart translation techniques that can be used to produce a better C code. Finally, the last section introduces a minimal, but interesting example of code optimization.

6.1 Analysis algorithms

The analysis passes perform some kind of reasoning on the code, without altering it. They are used to understand the code structure in order to execute some transformation by keeping its functional behavior intact.

6.1.1 Control flow graph analysis

Many interesting optimization techniques need to know in advance how the control flow interacts with the instructions of a program. A preliminary code analysis usually starts with locating the *basic block*s, that are sequences of consecutive instruction with:

- exactly one entry point for the control flow, represented by its first instruction;
- exactly one exit point for the control flow, represented by its last instruction.

In other words, the instructions in a basic block are always executed in a linear manner without intermediate jumps or branches. The first instruction in a basic block is often known as the *leader instruction*. In order to split a function into its constituting basic blocks it is sufficient to identify the leaders, for instance, by means of the Algorithm 6.1.

```
algorithm ComputeBasicBlocks(a function $F$) return the basic blocks of $F$

$\overline{B} \leftarrow (\ll\text{ENTRY}\gg)$
foreach instruction $i \in F$ do
    if $i$ is the first instruction of $F$ or
          $i$ has a label or
          the instruction preceding $i$ is a jump then
        $B \leftarrow$ new empty basic block
        $\overline{B} \leftarrow \overline{B} \cup (B)$
    end
    $B \leftarrow B \cup (i)$
end
$\overline{B} \leftarrow \overline{B} \cup (\ll\text{EXIT}\gg)$
return $\overline{B}$
```

Algorithm 6.1: Basic block analysis algorithm.

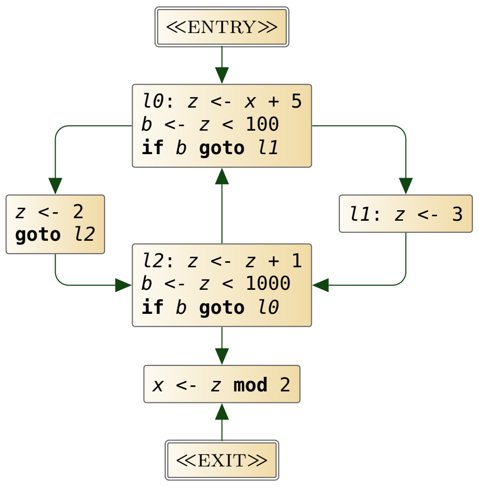
Note that the algorithm adds two fictitious blocks named ≪ENTRY≫ and ≪EXIT≫ representing the control flow entry and exit point of a function. Due to the properties of the HR, in this context, a function has exactly one entry and one exit point of the control flow. From this, it follows that the smallest sequence of basic blocks (i.e. for a function without instructions) consists of $\{\ll\text{ENTRY}\gg, \ll\text{EXIT}\gg\}$.

Basic blocks are quite important due to their flow execution simplicity. A great number of relatively simple basic block local optimizations can frequently increase the speed of the code, without the need to resort to much complex inter-basic blocks transformation techniques.

A number of code-improving algorithms, directly or indirectly, need to understand the relationships amongst the various basic blocks. The *control flow graph* is a graph where the nodes are the basic blocks and the edges indicate which blocks can follow which other blocks in the control flow of the program. Figure 6.1 shows a function and its control flow graph.

```
function f
    x in wo [0, 2000]
    y in ro [0, 2000]
{
    z in wo [0, 1]

    l0: z <- y + 5
    b <- z < 100
    if b goto l1
    z <- 2
    goto l2
    l1: z <- 3
    l2: z <- z + 1
    b <- z < 1000
    if b goto l0
    x <- z mod 2
}
```

(a) Source code in HRL.



(b) Control flow graph.

Figure 6.1: Example of a control flow graph.

The notations SUCC$[B]$ and PREC$[B]$ are used to respectively denote the set of all the basic blocks that follow or precede B in the control flow graph. The Algorithm 6.2 can be used to compute this information [109].

In CryptoGen, the basic block and control flow graph analysis are activated by specifying the `:controlFlowGraphAnalysis` annotation.

6.1.2 Live variable analysis

Variables and function parameters play an important role in a program, so that understanding when their content is used and modified is vital for several code-improving transformations.

With the notation USE[B] it is denoted the set of variables *used* by the basic block B, that is the variables that are read by the instructions in B. Analogously, DEF[B] indicates the set of variables *defined* by the basic block B, that is the variables whose content is modified by the instructions in B. In a three-address code representation, such as HR, computing these sets is trivial. As a rule of thumb, if a variable is on the left hand side of the `<-` in HRL it is defined, otherwise it is used.

Let v be a variable and i an instruction in a program. It is said that the variable v is *live* at i if its value is used along some path in the control flow graph starting from the instruction `i`. If a variable is not live at i, it is called *dead* at i. That means that if v is live at i, its content can be used by some instruction following i in the control flow. Given a basic block B, the *set of variables live on entry* IN[B] contains all the variables that are live at the first instruction of B. Similarly, the *set of variables live on exit* of B, denoted by OUT[B], contains all the variables that are live at the last instruction of B. The Algorithm 6.3 can be used to compute these two sets for each basic block of a function [109].

To enable the live variable analysis, the unit must be annotated with the `:liveVariableAnalysis` annotation. This pass automatically activates the control flow graph analysis, if needed.

```
algorithm ComputeCFG(a function $F$) return the control flow graph of $F$

$\overline{B} \leftarrow$ ComputeBasicBlocks($F$)
foreach basic block $B \in \overline{B}$ do
    succ[$B$] $\leftarrow \varnothing$
    prec[$B$] $\leftarrow \varnothing$
end
succ[0] $\leftarrow \{ \overline{B}[1] \}$
foreach basic block $B \in \overline{B} \setminus \{ \ll$ENTRY$\gg, \ll$EXIT$\gg \}$ do
    if the last instruction of $B$ is an unconditional jump then
        $B' \leftarrow$ basic block containing the target label of the jump
        succ[$B$] $\leftarrow$ succ[$B$] $\cup \{ B' \}$
        prec[$B'$] $\leftarrow$ prec[$B'$] $\cup \{ B \}$
    else
        if the last instruction of $B$ is a conditional jump then
            $B' \leftarrow$ basic block containing the target label of the jump
            succ[$B$] $\leftarrow$ succ[$B$] $\cup \{ B' \}$
            prec[$B'$] $\leftarrow$ prec[$B'$] $\cup \{ B \}$
        end
        $B' \leftarrow$ basic block following $B$
        succ[$B$] $\leftarrow$ succ[$B$] $\cup \{ B' \}$
        prec[$B'$] $\leftarrow$ prec[$B'$] $\cup \{ B \}$
    end
end
return $\left( \overline{B}, \{ \text{succ}[B] \}_{B \in \overline{B}}, \{ \text{prec}[B] \}_{B \in \overline{B}} \right)$
```

Algorithm 6.2: Control flow graph analysis algorithm.

```
algorithm ComputeLiveliness(a function $F$) return the set of live variables

$\left( \overline{B}, \{ \text{succ}[B] \}_{B \in \overline{B}}, \{ \text{prec}[B] \}_{B \in \overline{B}} \right) \leftarrow$ ComputeCFG($F$)
in[$\ll$EXIT$\gg$] $\leftarrow \varnothing$
foreach basic block $B \in \overline{B} \setminus \{ \ll$EXIT$\gg \}$ do
    while any in[$\cdot$] is changed do
        out[B] $\leftarrow \bigcup_{S \in \text{succ}[B]}$ in[S]
        in[B] $\leftarrow$ use[B] $\cup$ (out[B] $\setminus$ def[B])
    end
end
return $\left( \{ \text{in}[B] \}_{B \in \overline{B}}, \{ \text{out}[B] \}_{B \in \overline{B}} \right)$
```

Algorithm 6.3: Live variable analysis algorithm.

6.1.3 DAG analysis

Several important techniques for optimizing a basic block require a more structured representation of the instructions than a simple list. A DAG model is particularly interesting for the mathematical expressions and hence for the cryptography realm. For instance, the expression $(2 \cdot x + x)/y$ can be represented with the DAG shown in Figure 6.2.

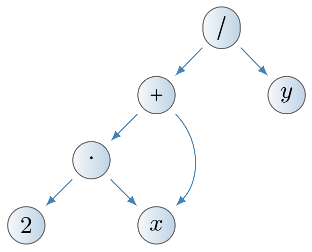


Figure 6.2: DAG representation for the $(2 \cdot x + x)/y$ expression.

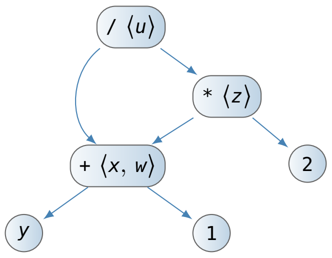
This approach can be extended to a whole basic block with some proper modifications, that is:

- each node has up to two children representing the two input operands of a generic instruction (since HR is a three-address code representation);
- the leaves are labeled with a constant, a variable name or with the **`nop`** instruction;
- the intermediate nodes are labeled with an instruction type and a list of variables where the value of the computation is written;
- the custom binary/unary operators are represented with a single node with one/two children.

Figure 6.3 shows an example DAG constructed for a basic block. Note that the expressions `x <- y + 1` and `w <- y + 1` are merged together in the graph since the analysis detected that they compute the same value.

```
y + 1
x * 2
y + 1
w / z
```

(a) Source code in HRL.



(b) DAG.

Figure 6.3: Example of the DAG for a basic block.

Care must be taken for the correct translation of the copy from/to tuple instructions. For instance, the code shown in Figure 6.4 contains the instructions `x <- y[i]` and `v <- y[i]` that can be apparently merged during the DAG construction. However, if `i = j` this cannot be done since the instruction `y[j] <- u` will alter the content of `y[i]`.

A way to circumvent these troublesome situations, as shown in Figure 6.4, is to alter the representation for the tuple instructions by:

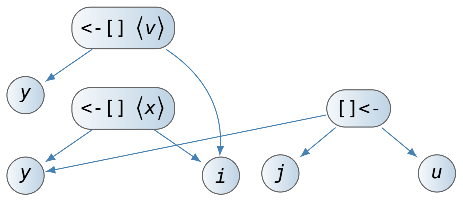
- modeling a copy from tuple instruction in the usual way, that is `x <- y[z]` is represented by means of a node with two children `y` and `z` and labeled with `x`;
- representing a copy to tuple instruction `x[y] <- z` by means of a node with three children `x`, `y` and `z`. These special nodes deny the subsequent merging for an instruction involving `y`.

The pseudo-code for building the DAG of a basic block is given in Algorithm 6.4.

The `:dagAnalysis` annotation activates this pass and also the control flow graph analysis, if needed.

```
y[i]
<- u
y[i]
```

(a) Source code in HRL.



(b) DAG.

Figure 6.4: Example of the DAG for some tuple instructions.

6.1.4 Operator chain analysis

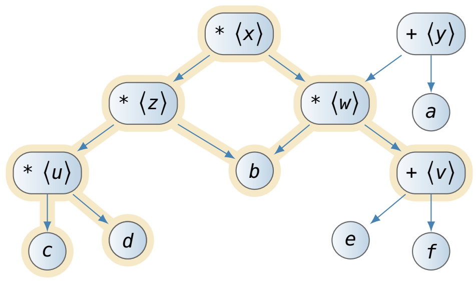
When performing certain types of optimizations, it is useful to detect successive applications of the same operator such as in $x + y + z + w$ or $x \cdot y \cdot z \cdot w$. These sequences are known as *operator chains* and they are particularly important when performing a number of simplifications for the algebraic expressions.

An effective way of locating them is through the DAG of a basic block, where an operator chain can be viewed as a particular sub-graph with all the intermediate nodes have the same instruction type.

For instance, the example sketched in Figure 6.5 depicts an operator chain rooted in the node `* ⟨x⟩`. By analyzing the DAG, it is simple to observe that it corresponds to the mathematical expression $x = ((c \cdot d) \cdot b) \cdot (b \cdot v)$.

```
v <- e + f
w <- b * v
u <- c * d
z <- u * b
x <- z * w
y <- w + a
```

(a) Source code in HRL.



(b) DAG.

Figure 6.5: Example of an operator chain.

Given a particular node n of a DAG, finding the biggest operator chain rooted in n is a relatively simple task, since it reverts to perform a depth first search extended until a node with an instruction incompatible with the node n's instruction is found.

The Algorithm 6.5 implements this idea.

This analysis is enabled by specifying the `:operatorChainAnalysis` annotation and it automatically activates the DAG analysis, if needed.

6.2 Transformation algorithms

The following paragraphs describe a number of annotation-aware code-improving techniques that alter the code,but preserve its semantic. These passes can be classified in two main categories:

- the *peephole* optimizations make use of a window-based approach, by analyzing and modifying a list of consecutive instructions;

algorithm ComputeDAG(a basic block B) **return** the DAG of B

```
$V[\cdot] \leftarrow \varnothing$, $O[\cdot] \leftarrow \varnothing$, $N \leftarrow \varnothing$
foreach instruction $i \in B$ do
    $out \leftarrow$ the output operand of $i$ or NIL if does not exist
    $in_1 \leftarrow$ the first input operand of $i$ or NIL if does not exist
    $in_2 \leftarrow$ the second input operand of $i$ or NIL if does not exist
    if $in_1 \neq$ NIL then
        $n_1 \leftarrow V[in_1]$
        if $n_1 =$ NIL then
            $n_1 \leftarrow$ new node labeled with $in_1$ and add it to $N$
            $V[in_1] \leftarrow n_1$
        end
    end
    if $n_2 \neq$ NIL then
        $n_2 \leftarrow V[in_2]$
        if $n_2 =$ NIL then
            $n_2 \leftarrow$ new node labeled with $in_2$ and add it to $N$
            $V[in_2] \leftarrow n_2$
        end
    end
    if $i$ is a copy to tuple then
        $n_0 \leftarrow V[out]$
        if $n_0 =$ NIL then
            $n_0 \leftarrow$ new node labeled with $out$ and add it to $N$
        end
    else
        $n_0 \leftarrow$ NIL
    end
    $n \leftarrow O[\{t, n_0, n_1, n_2\}]$
    if $n =$ NIL then
        $n \leftarrow$ new node with the type of $i$
        if $i$ is a copy to tuple then
            set $n_0$, $n_1$ and $n_2$ as children of $n$
            $V[out] \leftarrow$ NIL
        else
            set $out$ as the label of $n$ and $n_1$, $n_2$ as children of $n$
            $V[out] \leftarrow n$
        end
        $O[\{t, n_0, n_1, n_2\}] \leftarrow n$ and add it to $N$
    else
        add $out$ to the labels of $n$
    end
end
return $N$
```

Algorithm 6.4: DAG analysis algorithm.

```
algorithm ComputeChain(a DAG node $n$) return the biggest operator chain rooted in $n$
$C \leftarrow \varnothing$
$N \leftarrow (n)$
$t_1 \leftarrow$ instruction type of $n$ or NIL if it is a constant or variable
while $N \neq \varnothing$ do
    $n \leftarrow N[0]$
    $t_2 \leftarrow$ instruction type of $n$ or NIL if it is a constant or variable
    $C \leftarrow C \cup \{n\}$
    $N \leftarrow N \setminus N[0]$
    if $t_2 \neq$ NIL and $t_2$ is compatible with $t_1$ then
        $N \leftarrow$ the children of $n \cup N$
    end
end
return $C$
```

Algorithm 6.5: Operator chain analysis algorithm.

- the DAG-based optimizations make use of the graph representation explained in Section 6.1.3.

6.2.1 Constant folding

The *constant folding* is a simple peephole optimization whose goal is to replace an operation with its result, if its input operands are constants known at compile time. Figure 6.6 shows a code snippet before and after constant folding.

```
x <- (1, 2, 3) << 2
y <- 3 * 0
z <- 2 * y
```

(a) Original code.

```
x <- (3, 0, 0)
y <- 0
z <- 2 * y
```

(b) Transformed code.

Figure 6.6: Constant folding example.

Traditional end-compilers can perform constant folding, but this trivial optimization is important in CryptoGen because it can bring out several opportunities that can be exploited by other code transformation techniques not available to the end-compiler.

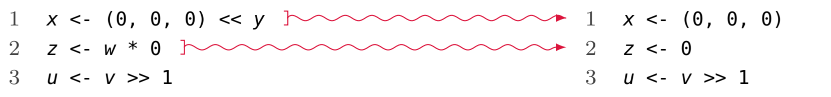
This code-improving technique is enabled when the `:constantFolding` annotation is specified.

6.2.2 Algebraic simplification

A relatively easy but effective optimization technique is the code simplification by means of some algebraic identities. This method reduces several instructions to a faster version even if some of its inputs are not known at compile time, as shown in Figure 6.7.

CryptoGen, can perform the simplification of integers and booleans, as most of the end-compilers, but can also take into account the relative's ranged values and several mathematical properties of the functions, specified via annotations. The function annotations that directly affect this transformation pass are:

- the `:hasIdentityElement(e)` annotation, which indicates that `e` is an identity element for a custom binary operator;



```
x <- (0, 0, 0) << y
z <- w * 0
u <- v >> 1
```

(a) Original code.

```
x <- (0, 0, 0)
z <- 0
u <- v >> 1
```

(b) Transformed code.

Figure 6.7: Algebraic simplification example.

- the `:hasAbsorbingElement(a)` annotation, which specifies that `a` is an absorbing element for a custom binary operator;
- the `:hasSpecialCase1(x, y)` annotation, which states that if the first input parameter of a custom binary operator is `x`, then the result of the instruction is `y`;
- the `:hasSpecialCase2(x, y)` annotation, which, similarly, indicates that if the second input parameter of a custom binary operator is `x`, then the result of the instruction is `y`.

Table 6.1 lists the simplification rules that the end-compiler cannot perform, but that CryptoGen can do.

original code	constraints	transformed code
`x <- y /= z`	y, z are relative and $\max(y) < \min(z)$	`x <- true`
`x <- y /= z`	y, z are relative and $\max(y) > \min(z)$	`x <- true`
`x <- y < z`	y, z are relative and $\max(y) < \min(z)$	`x <- true`
`x <- y < z`	y, z are relative and $\min(y) \geq \max(z)$	`x <- false`
`x <- y <= z`	y, z are relative and $\max(y) \leq \min(z)$	`x <- true`
`x <- y <= z`	y, z are relative and $\min(y) > \max(z)$	`x <- false`
`x <- y > z`	y, z are relative and $\max(y) > \min(z)$	`x <- true`
`x <- y > z`	y, z are relative and $\min(y) \leq \max(z)$	`x <- false`
`x <- y >= z`	y, z are relative and $\max(y) \geq \min(z)$	`x <- true`
`x <- y >= z`	y, z are relative and $\min(y) < \max(z)$	`x <- false`
push `x`, **push** `y`, **push** `z`, **call** `f`	`:hasIdentityElement(y)` holds for `f`	`x <- z`
push `x`, **push** `y`, **push** `z`, **call** `f`	`:hasIdentityElement(z)` holds for `f`	`x <- y`
push `x`, **push** `y`, **push** `z`, **call** `f`	`:hasAbsorbingElement(y)` holds for `f`	`x <- a`
push `x`, **push** `y`, **push** `z`, **call** `f`	`:hasAbsorbingElement(z)` holds for `f`	`x <- a`
push `x`, **push** `y`, **push** `z`, **call** `f`	`:hasSpecialCase1(y, w)` holds for `f`	`x <- w`
push `x`, **push** `y`, **push** `z`, **call** `f`	`:hasSpecialCase2(z, w)` holds for `f`	`x <- w`

Table 6.1: CryptoGen algebraic simplification rules.

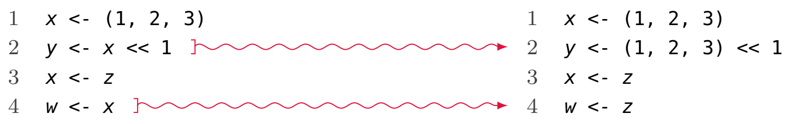
By means of this optimization technique, for instance, some identities that CryptoGen can exploit are:

- if $x \in \mathbb{F}_{2^n}$, then $x + 0 = 0 + x = x$, $x \cdot 0 = 0 \cdot x = 0$ and $x \cdot 1 = 1 \cdot x = x$;
- if $P \in \mathcal{E}$, then $P + P_\infty = P_\infty + P = P$, $0 \cdot P_\infty = P_\infty$ and $n \cdot P_\infty = P_\infty$.

The annotation `:algebraicSimplification` is used to enable this optimization pass.

6.2.3 Copy propagation

The *copy propagation* is the process of replacing the variables used in an instruction with their latest known value, that can be a constant or another variable name. Figure 6.8 shows the effects of the copy propagation on a simple code fragment.



```
1 x <- (1, 2, 3)
2 y <- x << 1
3 x <- z
4 w <- x
```

(a) Original code.

```
1 x <- (1, 2, 3)
2 y <- (1, 2, 3) << 1
3 x <- z
4 w <- z
```

(b) Transformed code.

Figure 6.8: Copy propagation example.

The copy propagation does not affect the performance of a program, but it is a preliminary code transformation that can unveil new opportunities for further optimizations. For instance, in Figure 6.8, the Line 2 can be further simplified via constant folding.

Performing a copy propagation in a basic block is simple due to its linear nature. This can be accomplished by keeping a map that store the last definition of a variable. The Algorithm 6.6 implements this peephole approach.

```
algorithm PropagateCopies(a basic block $B$) return the transformed basic block $B$
$V[\cdot] \leftarrow \varnothing$
foreach instruction $i \in B$ do
    $out \leftarrow$ the output operand of $i$ or NIL if does not exist
    $in_1 \leftarrow$ the first input operand of $i$ or NIL if does not exist
    $in_2 \leftarrow$ the second input operand of $i$ or NIL if does not exist
    if $in_1$ is a variable and $V[in_1] \neq$ NIL then
        replace $in_1$ with $V[in_1]$ in $i$
    end
    if $in_2$ is a variable and $V[in_2] \neq$ NIL then
        replace $in_2$ with $V[in_2]$ in $i$
    end
    if $i$ is a copy then
        $V[out] \leftarrow in_1$
    end
end
return $B$
```

Algorithm 6.6: Copy propagation algorithm.

In literature there exists a number of other techniques that can execute a global copy propagation spanning across the basic block boundaries using a much more complex algorithm [111].

In CryptoGen the `:copyPropagation` annotation is used to enable this code transformation and also the control flow graph analysis, if needed.

6.2.4 Dead code elimination

An instruction is considered *dead* if its execution is meaningless for the program. The exact definition of what is useful depends from the context. In the case of CryptoGen, an instruction is considered useless if it is not directly or indirectly used to compute a function output value or a global value.

In literature there exists several algorithms to eliminate the dead code on a whole function [109, 111]. CryptoGen implements a simplified version that works on each basic block.

The core idea is to recursively eliminate all the DAG root nodes with no live variables attached. The Algorithm 6.7 implements this concept.

```
algorithm EliminateDeadCode(a basic block $B$) return the transformed basic block $B$
$D \leftarrow$ ComputeDAG($B$)
$(\{\text{IN}[B]\}_{B\in\overline{B}}, \{\text{OUT}[B]\}_{B\in\overline{B}}) \leftarrow$ ComputeLiveliness(the function containing $B$)
foreach root node $n \in D$ do
    remove all the variables not in OUT[B] attached to $n$
    if $n$ has no variable attached then
        $D \leftarrow D \setminus (n)$
    end
end
$B \leftarrow$ reassemble $D$
return $B$
```

Algorithm 6.7: Dead code elimination algorithm.

Note that once the DAG has been modified, the basic block must be reassembled accordingly. This operation is trivial and can be done with a bottom-up visit of the DAG in order to keep the right instruction order. For instance, Figure 6.9 shows the effects of a dead code elimination pass over a small code fragment.

```
y <- x + 1
b <- y < 10
z <- true and w // z is dead.
w <- z // w is dead.
if b goto l1
```

(a) Original code.

```
y <- x + 1
b <- y < 10
if b goto l1
```

(b) Transformed code.

Figure 6.9: Dead code elimination example.

This optimization can be efficiently performed also by the end-compilers, but it has the interesting side effect that can reduce the set of the live variables on exit of a basic block. This in turn can decrease several constraints that other optimizations must adhere to, thus allowing CryptoGen to perform a better code improvement job.

This pass is enabled by means of the `:deadCodeElimination` annotation and enables the DAG and live variable analysis, if needed.

6.2.5 Strength reduction

The *strength reduction* is an optimization that replaces some instructions with some other one that run faster. For instance, a classic C strength reduction is to replace `x = 2 * y;` with `x = y + y;` since the hardware multiplier is usually much slower than the adder.

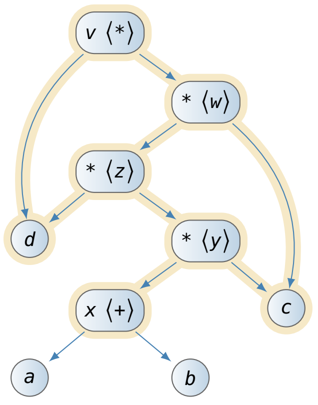
Apart the usual strength reductions on integers, that can be easily performed by the end-compilers, CryptoGen leverages the `:hasSpecialization(f)` annotation to improve the code speed.

This annotation indicates that when a custom binary operator is called with the same input operands it can be reduced to the unary operator `f`. By means of this indications, for example, CryptoGen can simplify a field multiplication into a much faster squaring or a point addition into a point doubling.

Implementing this technique analyzing each instruction independently from the others can lead to miss several code-improving opportunities. For instance, Figure 6.10 shows a code that contains two ‘hidden’ strength reductions relative to the expression $v = d \cdot (d \cdot ((a + b) \cdot c) \cdot c)$. By strictly following the order imposed by the parenthesis it is impossible to simplify this equation.

```
x <- a + b
y <- x * c
z <- d * y
w <- z * c
v <- d * w
```

(a) Source code in HRL.



(b) DAG.

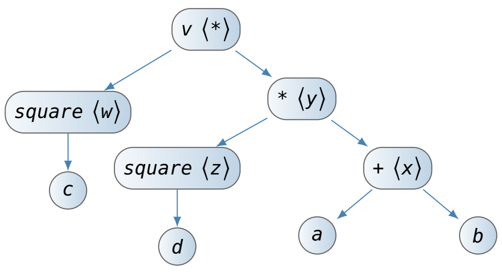
Figure 6.10: Example of hidden strength reduction possibilities.

To bring out the optimization potentials, CryptoGen first identifies the operator chains (Section 6.1.4). Then, if the binary operator has been annotated with the `:isCommutativeAndAssociative` annotation, CryptoGen is authorized to reorder its parameters (since, for the operator, both the commutative and associative properties hold). Care must be taken, however, since reordering the parameters can affect the values of the intermediate variables (`w`, `z` and `y` in the example). As a rule of thumb, the content of a variable cannot be altered if it is still used later in the basic block or in another basic block. The Algorithm 6.8 reorders the variables and simplifying the binary operators whenever possible.

After the rewriting, the DAG in Figure 6.10 becomes as depicted in Figure 6.11. It represents the equivalent equation $v = c^2 \cdot (d^2 \cdot (a + b))$.

```
x <- a + b
push z
push d
call square
y <- z * x
push w
push c
call square
v <- w * y
```

(a) Source code in HRL.



(b) DAG.

Figure 6.11: Strength reduction example.

This pass is activated by means of the `:strengthReduction` annotation. It automatically enable

```
algorithm ReduceStrengths(a basic block $B$) return the transformed basic block $B$
$D \leftarrow$ ComputeDAG($B$)
$(\{\text{IN}[B]\}_{B\in\overline{B}}, \{\text{OUT}[B]\}_{B\in\overline{B}}) \leftarrow$ ComputeLiveliness(the function containing $B$)
foreach root node $n \in D$ do
    $N \leftarrow (n)$
    while $N \neq \emptyset$ do
        $n \leftarrow N[0]$
        $N \leftarrow N \setminus N[0]$
        $N \leftarrow$the children of $n \cup N$
        if n is related to a binary operator that is commutative and associative then
            $C \leftarrow$ ComputeChain($n$)
            remove all the non-frontier nodes from $C$
            remove all the nodes with more than 1 parent from $C$
            remove all the nodes with at least one variable in OUT[B] from $C$
            repeat
                pick two nodes $n_1 \neq n_2$ with a common children $c$
                if $n_1$ is a successor of $n_2$ then
                    $d \leftarrow$ the children of $n_2$ different from $c$
                    set $d$ as the child of $n_2$ equal to $c$
                    set $c$ as the child of $n_2$ equal to $d$
                else
                    $d \leftarrow$ the children of $n_1$ different from $c$
                    set $c$ as the child of $n_1$ equal to $d$
                    set $d$ as the child of $n_1$ equal to $c$
                end
            until any change in $C$ occurs
        end
    end
end
foreach node $n \in D$ do
    if $n$ has the same two children and its type has a specialization $G$ then
        change the type of $n$ to $G$ and remove one of its children
    end
end
$B \leftarrow$ reassemble $D$
return $B$
```

Algorithm 6.8: Strength reduction algorithm.

the DAG, operator chain and live variable analysis.

6.2.6 Common sub-expression elimination

When translating from a high level language to a lower level one (e.g. from aXiom to HRL or from C to assembler), the developer can involuntarily create several *common sub-expressions*, that are expressions that evaluate the same value twice or more. By applying the *common sub-expression elimination* optimization, a compiler tries to reduce these redundant computations, usually by replacing them with simpler and faster copy instructions.

CryptoGen performs the common sub-expression elimination for each basic block by analyzing its DAG representation. Note that the algorithm presented in Section 6.1.3 is already able to identify the simplest common sub-expressions by merging together similar nodes. Once the graph has been built, CryptoGen analyzes it in order to discover some hidden sub-expressions by exploiting the functions' mathematical properties specified via annotations.

If a binary operator has been labeled with the `:isCommutativeAndAssociative` annotation, then CryptoGen will perform a deeper search ignoring the order of parameters of the operator chains. The Algorithm 6.9 implements this idea.

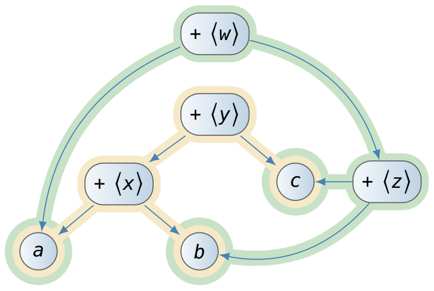
```
algorithm EliminateCS(a basic block $B$) return the transformed basic block $B$

$D \leftarrow$ ComputeDAG($B$)
$C[\cdot] \leftarrow \varnothing$
foreach node $n \in D$ do
    $C[n] \leftarrow$ ComputeChain($n$)
end
foreach node $n \in D$ do
    if the type of $n$ is associative and commutative then
        $C_1 \leftarrow C[n]$
        foreach chain $C_2 \in C[\cdot]$ such that $C_1 \neq C_2$ and $|C_1| = |C_2|$ do
            if $C_1$ and $C_2$ have the same type and the same leaf nodes in any order then
                merge $n$ and the root of $C_2$
            end
        end
    end
end
$B \leftarrow$ reassemble $D$
return $B$
```

Algorithm 6.9: Common sub-expression elimination algorithm.

Figure 6.12 shows an example containing a common sub-expression. The algorithm infers that the value $y = a + b + c$ is the same as $w = a + c + b$.

```
a + b
x + c
c + b
a + z
```

(a) Source code in HRL.



(b) DAG.

Figure 6.12: Common sub-expression elimination example.

This pass is activated by specifying the `:commonSubexpressionElimination` annotation. It automatically enables the DAG and operator chain analysis, if needed.

6.3 Translation algorithms

This section introduces the optimization techniques that are used during the translation process of HR into the final C code.

6.3.1 Bit vector packing

A theoretical simple but powerful optimization performed during the C code generation is the *bit vector packing*. In the C programming language, the most naive way to implement a bit vector (or a bit matrix in general) is to use an array of integers where each element hosts one bit. By activating the bit vector packing, CryptoGen will implement the bit vectors by making use of all the bits in an array of machine words. For instance, a vector `x` containing 79 bits will be translated as `uint8_t x[79]` is no optimization is active, while it will be implemented as `uint8_t x[10]` is the bit vector packing is active (on a 8 bit processor).

Apart from reducing the memory consumption, the major benefit of this optimization is that the speed of the operations can be greatly improved by leveraging the ALU ability to work on an entire machine word at a time. This boost is particularly evident for the smaller bit vectors (that can be stored on very few words).

For instance, the Figure 6.13 shows two code regions performing the left shift of one position on a vector containing 79 bits without and with the bit vector packing. The unoptimized code contains a loop with 79 iterations, while the improved version has a loop with 10 iterations, being approximatively about 7.9 times faster than its counterpart. As a rule of thumb, if the machine word size is w bits, this technique allows to achieve a speed gain of about w times (if instruction and memory caching is not taken into account).

```c
uint8_t x[79];
uint8_t y[79];
int i;

for (i = 78; i > 0; --i)
    x[i] = y[i - 1];
x[0] = 0;
```

(a) Code without bit vector packing.

```c
uint8_t x[10];
uint8_t y[10];
int i;

for (i = 9; i > 0; --i)
    x[i] = (y[i]  << 1) ^
        (y[i - 1] >> 7);
x[0] = y[0]  << 1;
```

(b) Code with bit vector packing.

Figure 6.13: Bit vector packing example.

This optimization cannot be performed by the end-compiler because it cannot distinguish an array `uint8_t x[79]` used to store 79 8 bit integers from one used to represent a sequence of 79 bits.

This translation technique is enabled via the `:bitVectorPacking` annotation.

6.3.2 Loop unrolling

The *loop unrolling*, also known as *loop unwinding*, is a simple optimization by which a loop with n iterations is replaced by a loop with n/m iterations. If $m = n$ the loop is completely removed by generating for n times its body. This technique offers several advantages since the branch penalties are minimized and it can bring out several opportunities for other optimization passes. The disadvantages are an increased program size and, if performed too aggressively, an execution time slow down due to an increased register spilling and filling[1].

When the end-compiler optimizes the code it has to choose 'how' and 'when' performs the loop unrolling. Several compilers follow a quite conservative approach, by performing the unwinding only on very small and simple loops. However a more aggressive loop unrolling can potentially improve the code performance.

The loop unrolling works particularly well if the loop has few iterations and contains several parallelizable instructions. A number of the HR instruction needs to work on arrays and follows the aforementioned pattern (e.g. the exclusive or instruction **`xor`** on bit vectors). CryptoGen can enable the loop unwinding of these instructions to produce a completely loop-free C translation of an instruction.

For instance, the Figure 6.14 shows the effect of the loop unrolling when translating the and between two bit vectors.

```c
uint8_t x[5];
uint8_t y[5];
uint8_t z[5];
int i;

for (i = 0; i < 5; ++i)
    x[i] = y[i] & z[i];
```

(a) Code without loop unrolling.

```c
uint8_t x[5];
uint8_t y[5];
uint8_t z[5];
int i;

i = 0;
x[i] = y[i] & z[i];
++i;
x[i] = y[i] & z[i];
++i;
x[i] = y[i] & z[i];
++i;
x[i] = y[i] & z[i];
++i;
x[i] = y[i] & z[i];
++i;
```

(b) Code with loop unrolling.

Figure 6.14: Loop unrolling example.

The effects of this optimization is usually very beneficial on high-end multi-scalar platforms, where multiple instructions can be executed in parallel or partially overlapped through the processor pipeline.

This pass is enabled via the `:loopUnrolling` annotation.

6.3.3 Use of target idioms

Several compilers and platforms allow the use of ad-hoc instructions to perform some specific computation completely in hardware. By using these custom instructions instead of the standard C constructs, CryptoGen can increase the performance of the code, sacrificing its portability.

Currently CryptoGen supports only two built-ins for the GCC compiler: the `__builtin_clzl()` and the `__builtin_ia32_pclmulqdq128()` functions.

The `__builtin_clzl()` is used to compute the number of leading zero bits in a **`long`** variable. This instruction is automatically used when translating the **`findlastset`** HR construct. GCC can

[1]The *register spilling* is the operation of moving a variable from a register to the memory. The opposite procedure is known as *register filling*.

implement this built-in differently w.r.t. the hardware platform, for instance:

- on the AMD64 family it uses the *bsr* (bit scan reverse) assembler instruction, which implements the find last set on a word;
- on the ARMv7 family it uses the *clz* (count leading zeros) assembler instruction, which counts the leading zeros in a word.

The *__builtin_ia32_pclmulqdq128()* is available only on the AMD64 processors and it implements the *pclmulqdq* assembler instruction [112]. This construct was introduced in 2010 with the Intel Westmere micro-architecture and it allows to compute the carry-less multiplication of two 64 bit words. In other words, it computes the binary field reduction-less multiplication between two words, that is $\cdot : \mathbb{F}_{2^{64}} \times \mathbb{F}_{2^{64}} \to \mathbb{F}_{2^{128}}$. In order to be effectively used by CryptoGen, a function computing the field multiplication must be annotated with the *:carryLessMultiplication* annotation. When performing the translation of the annotated function, CryptoGen will replace its body with a call to the *__builtin_ia32_pclmulqdq128()* built-in if its parameters are respectively a 128 and two 64 bit vectors. In all the other cases, CryptoGen will perform the usual function translation. A similar annotation, *:carryLessSquaring*, is available for the field squaring function.

This optimization is enabled when the *:targetIdiomUsage* annotation is present.

6.4 Optimization example

The strength of the optimization algorithms is bring out when they are chained together to form a complete code-improving toolbox. In the following paragraphs, a full example is given on an elliptic curve algorithm.

The equations shown in Table 2.1 are correct, but their naive implementation can be greatly improved by the optimization algorithms listed in Section 6.2. For instance, the expressions for computing the point doubling within the López-Dahab coordinate system are:

$$\begin{cases} a &= a_6 z_P^4 \\ z_R &= z_P^2 x_P^2 \\ x_R &= x_P^4 + a \\ b &= y_P^2 + a_2 z_R + a \\ y_R &= b x_R + z_R a. \end{cases}$$

A possible three-address code translation over $\mathbb{F}_{2^{13}}$ when $z_P = 1$ is given in Algorithm 6.10. Note that the code was poorly written on purpose, with no squarings and with a bad order of the products. The implementation costs $14\mathcal{M} + 0\mathcal{S}$.

After some rounds of constant folding, algebraic simplifications and copy propagations, some dead code can be eliminated, leading to the Algorithm 6.11. Its cost drops to $8\mathcal{M} + 0\mathcal{S}$.

The strength reduction and common sub-expression elimination passes further decreases the code cost to $4\mathcal{M} + 2\mathcal{S}$, as shown in Algorithm 6.12.

Finally, another round of copy propagation, strength reduction and dead code elimination leads to the final code in Algorithm 6.13 with a cost of $3\mathcal{M}+3\mathcal{S}$. This is the same optimal formula known in literature [113].

```
$t_0 \leftarrow (0,0,0,0,0,0,0,0,0,0,0,1) \cdot a_6$
$t_1 \leftarrow t_0 \cdot (0,0,0,0,0,0,0,0,0,0,0,1)$
$t_2 \leftarrow t_1 \cdot (0,0,0,0,0,0,0,0,0,0,0,1)$
$a \leftarrow t_2 \cdot (0,0,0,0,0,0,0,0,0,0,0,1)$
$t_3 \leftarrow (0,0,0,0,0,0,0,0,0,0,0,1) \cdot (0,0,0,0,0,0,0,0,0,0,0,1)$
$t_4 \leftarrow t_3 \cdot x_P$
$z_R \leftarrow t_4 \cdot x_P$
$t_5 \leftarrow x_P \cdot x_P$
$t_6 \leftarrow t_5 \cdot x_P$
$t_7 \leftarrow t_6 \cdot x_P$
$x_R \leftarrow t_7 + a$
$t_8 \leftarrow y_P \cdot y_P$
$t_9 \leftarrow a_2 \cdot z_R$
$t_{10} \leftarrow t_8 + t_9$
$b \leftarrow t_{10} + a$
$t_{11} \leftarrow b \cdot x_R$
$t_{12} \leftarrow z_R \cdot a$
$y_R \leftarrow t_{11} + t_{12}$
```
Algorithm 6.10: Naive implementation of the point doubling with $z_R = 1$.

```
$z_R \leftarrow x_P \cdot x_P$
$t_5 \leftarrow x_P \cdot x_P$
$t_6 \leftarrow t_5 \cdot x_P$
$t_7 \leftarrow t_6 \cdot x_P$
$x_R \leftarrow t_7 + a_6$
$t_8 \leftarrow y_P \cdot y_P$
$t_9 \leftarrow a_2 \cdot z_R$
$t_{10} \leftarrow t_8 + t_9$
$b \leftarrow t_{10} + a_6$
$t_{11} \leftarrow b \cdot x_R$
$t_{12} \leftarrow z_R \cdot a_6$
$y_R \leftarrow t_{11} + t_{12}$
```
Algorithm 6.11: Point doubling code after some initial simplifications.

```
$z_R \leftarrow x_P^2$
$t_5 \leftarrow z_R$
$t_6 \leftarrow z_R$
$t_7 \leftarrow t_6 \cdot t_5$
$x_R \leftarrow t_7 + a_6$
$t_8 \leftarrow y_P^2$
$t_9 \leftarrow a_2 \cdot z_R$
$t_{10} \leftarrow t_8 + t_9$
$b \leftarrow t_{10} + a_6$
$t_{11} \leftarrow b \cdot x_R$
$t_{12} \leftarrow z_R \cdot a_6$
$y_R \leftarrow t_{11} + t_{12}$
```
Algorithm 6.12: Point doubling code after some DAG-based optimizations.

```
$z_R \leftarrow x_P^2$
$t_7 \leftarrow z_R^2$
$x_R \leftarrow t_7 + a_6$
$t_8 \leftarrow y_P^2$
$t_9 \leftarrow a_2 \cdot z_R$
$t_{10} \leftarrow t_8 + t_9$
$b \leftarrow t_{10} + a_6$
$t_{11} \leftarrow b \cdot x_R$
$t_{12} \leftarrow z_R \cdot a_6$
$y_R \leftarrow t_{11} + t_{12}$
```

Algorithm 6.13: Optimal point doubling code.

Chapter 7

Experimental results

If your experiment needs statistics, you ought to have done a better experiment.

Ernest Rutherford

In this chapter the implementation results are presented and discussed. The data gathered in the following sections represent the performance obtained by compiling and optimizing the algorithms and equations[1] described in Section 2.2.

This dissertation's framework was implemented in Java 8 as a set of plug-ins for the Eclipse Mars IDE (`http://www.eclipse.org`). The following tools were also used during the development:

- Xtend (`http://www.eclipse.org/xtend/`), a high level programming language offering additional features w.r.t. Java as type inferencing, operator overloading and extension methods;
- Xcore (`https://wiki.eclipse.org/Xcore`), a tool for describing and developing EMF[2] models and data types;
- Xtext (`https://eclipse.org/Xtext/`), a powerful framework for developing DSLs, their parsers and their graphical editors.

In addition, Mathematica (`https://www.wolfram.com/mathematica/`) was also used to automatically generate the binary field and elliptic curve test cases.

The complete toolbox consists of 30 plug-ins, containing 564 files (292 hand-written and 272 automatically generated), for a total of 212 648 lines of code (38 640 hand-written and 174 008 automatically generated).

Table 7.1 shows the specifications of the platforms where the optimized code was compiled, executed and measured. It consists of a traditional laptop processor (an Intel i7) and a mobile device CPU (an ARMv7).

All the tests were focused on the five binary fields and the corresponding B elliptic curves listed in the standard *FIPS PUB 186-4 – Digital Signature Standard (DSS)* [13]. All the curves are described by the equation $y^2 + xy = x^3 + x^2 + a'_6$ and their parameters are reported in Table 7.2.

[1]Recall that the formulas displayed in Section 2.2 are not in an optimal form (e.g. they include several common sub-expressions that can be removed).

[2]EMF (`http://www.eclipse.org/emf/`) is a framework for creating models in the context of the MVC paradigm.

manufacturer	Apple	Samsung
model	MacBook Pro	S5
word size	64 bit	32 bit
CPU	Intel i7-4980HQ	ARMv7 rev 3 (v7l)
CPU base frequency	2.8 GHz	600 MHz
RAM	16 GiB	1.4 GiB
OS	Debian Linux 4.3.5	Android 4.4.2
compiler	GCC 5.3.1	GCC 4.9
optimization flags	`-O3 -mpclmul`	`-O3`

Table 7.1: Specifications of the tested hardware platforms.

curve name	field	a'_6
B-163	$t^{163} + t^7 + t^6 + t^3 + 1$	$20a601907b8c953ca1481eb10512f78744a3205fd_{16}$
B-233	$t^{233} + t^{74} + 1$	$066647ede6c332c7f8c0923bb58213b333b20e9ce4281fe115f7d8f90ad_{16}$
B-283	$t^{283} + t^{12} + t^7 + t^5 + 1$	$27b680ac8b8596da5a4af8a19a0303fca97fd7645309fa2a581485af6263e313b79a2f5_{16}$
B-409	$t^{409} + t^{87} + 1$	$021a5c2c8ee9feb5c4b9a753b7b476b7fd6422ef1f3dd674761fa99d6ac27c8a9a197b272822f6cd57a55aa4f50ae317b13545f_{16}$
B-571	$t^{571} + t^{10} + t^5 + t^2 + 1$	$2f40e7e2221f295de297117b7f3d62f5c6a97ffcb8ceff1cd6ba8ce4a9a18ad84ffabbd8efa59332be7ad6756a66e2939baca0c7ffeff7f2955727a_{16}$

Table 7.2: Specifications of the tested elliptic curves and fields.

All the measurements were computed using the Algorithm 4.1 with 1000 populations of 100 samples each (so that each test was executed 100 000 times) with a sensitivity of 5 %.

This chapter reports only the most interesting data. Additional results are available in Appendix D.

7.1 Impact of optimizations

This section experimentally investigates the effects that the optimizations have on the produced code. In order to simplify the discussion, the following tests concern the scalar multiplication of only one curve: the B-233, one of the most used ones.

Table 7.3 shows the data gathered on the i7 processor for the aforementioned curve.

The column 'optimizations' lists the code-improving techniques that were enabled, that is:

- 'none' indicates that no optimization is active;
- '+ bit vector packing' denotes that only the bit vector packing (Section 6.3.1) is active;
- '+ target idiom usage' specifies that the target idiom usage (Section 6.3.3) is active in addition to the previous optimizations;
- '+ loop unrolling' indicates that the loop unrolling (Section 6.3.2) is active in addition to the previous optimizations;

optimizations	time		code size	
	value [ns]	ratio	value [B]	ratio
none	448 520 206	–	64 301	–
+ bit vector packing	4 570 143	98.1	46 015	0.7
+ target idiom usage	2 053 109	2.2	37 099	0.8
+ loop unrolling	59 481	34.5	11 345	0.3
+ middle-end optimizations	40 036	1.5	11 280	1.0

Table 7.3: Effects of the optimizations on the B-233 curve's scalar multiplication code.

- '+ middle-end optimizations' denotes that all the middle-end stage optimizations (Section 6.2) are active in addition to the previous optimizations (that is a full optimization).

The 'time' and 'code size' columns respectively indicate the time to execute a single multiplication and the static size of the generated code. The two 'ratio' columns contain the ratio between the previous time/code size value and the current one.

The bit vector packing, as expected, decreases both the time and the code size of the scalar multiplication. Naively, on a 64 bit CPU such as the i7, this option allows to perform the bit vector operations about 64 times faster. In practice, the achieved gain is much greater since the bit vector size is also reduced, promoting a faster caching of the intermediate results, further boosting the speed-up.

Enabling the target idiom usage optimization allows the use of some special GCC built-ins (such as the `__builtin_ia32_pclmulqdq128()` function). This allows the code to better leverage the underlying hardware and also to decrease the code size since some algorithm parts are replaced by faster and fewer instructions[3].

The aggressive loop unrolling further boosts the speed since it removes branches and simplifies the control flow graph, reducing the number of basic blocks, allowing GCC to better exploit the instruction parallelism. Oddly enough, this optimization is also beneficial for the code size. Global optimizations between basic blocks are notoriously harder than the intra-basic block ones, however by reducing the number of branches in the control flow, GCC can identify better the common sub-expressions and the dead code, thus producing a smaller binary.

When all the middle-end optimizations are enabled, the performance gains an additional 50 % of speed. This is mainly due to the fact that the equations shown in Table 2.1 are factorized in a better way, removing unnecessary sub-expression and function calls.

7.2 Performance

The following paragraphs report the performance of the generated code on the Intel i7 and ARMv7 processors. All the algorithms were chosen in order to maximize the speed of the scalar multiplication.

Table 7.4 shows the symbols and abbreviations used in this section.

[3]In this case the low level 64 bit × 64 bit WindowMultiply function implements the Comb algorithm with a window width of 4 bit if the target idiom usage is not enabled, otherwise it is translated as a single `__builtin_ia32_pclmulqdq128()` function call.

symbol	operation	symbol	algorithm/space
$a \mod b$	field modular reduction	R	Reduce
$a + b$	field addition	A	Add
		Cn	Comb with width n
$a \cdot b$	field multiplication	Kn	Karatsuba+Comb, window on n bit
		M	WindowMultiply, window on 64 bit
a^2	field squaring	Pn	PrecomputeSquare, window on n bit
		S	WindowSquare, window on 64 bit
a^{-1}	field multiplicative inversion	B	BEA
		D	DCEA
$P + Q$	point addition		
$2P$	point doubling	LD	López-Dahab space
$-P$	point negation		
$n \cdot P$	scalar multiplication	Wn	NAFwMultiply, NAF$_w$ on n bit

Table 7.4: Legend of the symbols.

7.2.1 Results on the i7 processor

Table 7.5 shows the performance achieved on the Intel i7 processor.

operation	time [ns]				
	B-163	B-233	B-283	B-409	B-571
$a \mod b$	$^{\text{R}}8$	$^{\text{R}}6$	$^{\text{R}}8$	$^{\text{R}}7$	$^{\text{R}}17$
$a + b$	$^{\text{A}}1$	$^{\text{A}}1$	$^{\text{A}}1$	$^{\text{A}}2$	$^{\text{A}}3$
$a \cdot b$	$^{\text{M}}7$	$^{\text{M}}11$	$^{\text{M}}14$	$^{\text{M}}35$	$^{\text{M}}217$
a^2	$^{\text{S}}5$	$^{\text{S}}7$	$^{\text{S}}9$	$^{\text{S}}25$	$^{\text{S}}190$
a^{-1}	$^{\text{B}}1254$	$^{\text{B}}2206$	$^{\text{B}}3005$	$^{\text{B}}6001$	$^{\text{B}}11\,507$
$P + Q$	$^{\text{LD}}238$	$^{\text{LD}}256$	$^{\text{LD}}354$	$^{\text{LD}}601$	$^{\text{LD}}3301$
$2P$	$^{\text{LD}}177$	$^{\text{LD}}180$	$^{\text{LD}}243$	$^{\text{LD}}410$	$^{\text{LD}}2261$
$-P$	$^{\text{LD}}6$	$^{\text{LD}}8$	$^{\text{LD}}10$	$^{\text{LD}}14$	$^{\text{LD}}19$
$n \cdot P$	$^{\text{W3}}27\,522$	$^{\text{W3}}40\,036$	$^{\text{W3}}65\,832$	$^{\text{W3}}163\,318$	$^{\text{W4}}1\,240\,570$

Table 7.5: Performances on the i7 processor.

With a proper hardware support of the carry-less multiplication through the `pclmulqdq` assembler instruction, the field multiplication and squaring achieve an impressive speed-up, such that their temporal cost tend to become negligible with respect to the other functions. The only non-negligible field operation is the multiplicative inversion, and hence the division.

In this case, the coordinate system of choice for maximizing the performance of the elliptic curve scalar multiplication is the López-Dahab one. Selecting the proper coordinate system for achieving the fastest scalar multiplication can be (theoretically) performed in advance by means of some cost computations. For instance, the cost estimation for the add-and-double method (Algorithm 2.11) is $\frac{l}{2}\mathcal{A} + l\mathcal{D}$. Let's look at B-233, so that $l = 233$, $\mathcal{I}/\mathcal{M} \approx 200.5$ and $\mathcal{M}/\mathcal{S} \approx 1.6$. By using the formulas listed in Table 2.2, the following costs can be estimated[4]:

- in affine space, $\mathcal{A} = \mathcal{D} = 325\mathcal{S}$ so that the add-and-double cost is $113587.5\mathcal{S}$;
- in projective space, $\mathcal{A} = 23.4\mathcal{S}$ and $\mathcal{D} = 14.2\mathcal{S}$ so that the add-and-double cost is $6034.7\mathcal{S}$;
- in Jacobian space, $\mathcal{A} = 27.4\mathcal{S}$ and $\mathcal{D} = 11.4\mathcal{S}$ so that the scalar multiplication cost is $5848.3\mathcal{S}$;
- in López-Dahab space, $\mathcal{A} = 24.8\mathcal{S}$ and $\mathcal{D} = 9.8\mathcal{S}$ so that the add-and-double cost is $5172.6\mathcal{S}$.

An analogous approach can be used for the other scalar multiplication techniques.

7.2.2 Results on the ARMv7 processor

Table 7.6 reports the results on the ARMv7 CPU.

operation	time [ns]				
	B-163	B-233	B-283	B-409	B-571
$a \bmod b$	$^{\mathrm{R}}104$	$^{\mathrm{R}}120$	$^{\mathrm{R}}281$	$^{\mathrm{R}}246$	$^{\mathrm{R}}565$
$a+b$	$^{\mathrm{A}}45$	$^{\mathrm{A}}69$	$^{\mathrm{A}}88$	$^{\mathrm{A}}153$	$^{\mathrm{A}}239$
$a \cdot b$	$^{\mathrm{K4}}3743$	$^{\mathrm{K4}}7266$	$^{\mathrm{K4}}12\,714$	$^{\mathrm{K5}}25\,094$	$^{\mathrm{K5}}77\,572$
a^2	$^{\mathrm{P8}}1621$	$^{\mathrm{P8}}3282$	$^{\mathrm{P8}}5520$	$^{\mathrm{P8}}9280$	$^{\mathrm{P10}}19\,431$
a^{-1}	$^{\mathrm{D}}29\,484$	$^{\mathrm{B}}56\,426$	$^{\mathrm{B}}78\,188$	$^{\mathrm{B}}166\,485$	$^{\mathrm{B}}314\,153$
$P+Q$	$^{\mathrm{LD}}58\,365$	$^{\mathrm{LD}}112\,388$	$^{\mathrm{LD}}157\,315$	$^{\mathrm{LD}}426\,454$	$^{\mathrm{LD}}1\,155\,117$
$2P$	$^{\mathrm{LD}}27\,795$	$^{\mathrm{LD}}53\,370$	$^{\mathrm{LD}}83\,751$	$^{\mathrm{LD}}206\,658$	$^{\mathrm{LD}}428\,432$
$-P$	$^{\mathrm{LD}}5744$	$^{\mathrm{LD}}7756$	$^{\mathrm{LD}}10\,601$	$^{\mathrm{LD}}28\,083$	$^{\mathrm{LD}}80\,703$
$n \cdot P$	$^{\mathrm{W2}}4\,649\,553$	$^{\mathrm{W2}}12\,443\,090$	$^{\mathrm{W2}}21\,601\,999$	$^{\mathrm{W3}}94\,854\,413$	$^{\mathrm{W5}}287\,818\,708$

Table 7.6: Performances on the ARMv7 processor.

The ARMv7 belongs to a 32 bit family of processors, widely diffused in the mobile world. For these reason, the performance of the field and curve operations are much slower than their counterpart on the i7 processor.

These CPU families have no support for a carry-less multiplication in hardware, so that the comb (Algorithm 2.3) and Karatsuba-Ofman algorithms (Algorithm 2.4) must be used instead.

An interesting result pertains the multiplicative inversion. In this context, the DCEA algorithm is the fastest one in the smallest field ($\mathbb{F}_{2^{163}}$), while in the other ones the BEA is the optimal choice. This is consistent with the literature results [18] where the DCEA offers superior performance on smaller fields on the ARM architectures.

Also in this case, the point arithmetic is faster if the López-Dahab coordinate system is chosen.

7.3 Comparison against standard libraries

There exist a number of cryptographic libraries written in various programming languages and several of them contain some elliptic curve related algorithms and primitives. Most of them, however, implements only elliptic curves on prime fields, most likely due to the fact that the arithmetic

[4]These values are lower bounds, since they do not take into account several kind of overheads such as the time spent to perform a function call, the register spilling/filling, etcetera.

on prime fields is often (but not always) faster than its binary counterpart (see Section 7.4 for a discussion about this subject).

In this context, two widely used and open-source cryptographic libraries that support binary elliptic curves are:

- OpenSSL 1.0.2 (`https://www.openssl.org`), a C library that is the de facto standard implementation for the SSL and TLS protocols in the open-source world;

- Crypto++ 5.6.1 (`http://www.cryptopp.com`), a C++ library extensively used in academic and open-source projects, well known for its multitude of cryptographic primitives and, unfortunately, the lack of high speed algorithms.

Table 7.7 lists the performance of the scalar multiplications on the i7 processor, comparing together the code produced by CryptoGen, the OpenSSL and the Crypto++ libraries. The ‘ratio’ row contains the ratio between the CryptoGen time and the OpenSSL/Crypto++ one, while the circle values emphasize the fastest results.

		B-163	B-233	B-283	B-409	B-571
CryptoGen	time [ns]	$\boxed{27\,522}$	$\boxed{40\,036}$	$\boxed{65\,832}$	$\boxed{163\,318}$	$\boxed{1\,240\,570}$
OpenSSL	time [ns]	180 007	232 475	466 992	735 405	1 848 002
	ratio	6.5	5.8	7.1	4.5	1.5
Crypto++	time [ns]	15 732 963	36 693 675	62 668 174	183 510 370	565 496 848
	ratio	571.7	916.5	952.0	1123.6	455.8

Table 7.7: Scalar multiplication performance on different implementations.

The Figure 7.1 graphically depicts the performance of the scalar multiplications to better visualize the speed gain. Note that the all the Crypto++ bars are cropped in order to make the CryptoGen/OpenSSL comparison more readable.

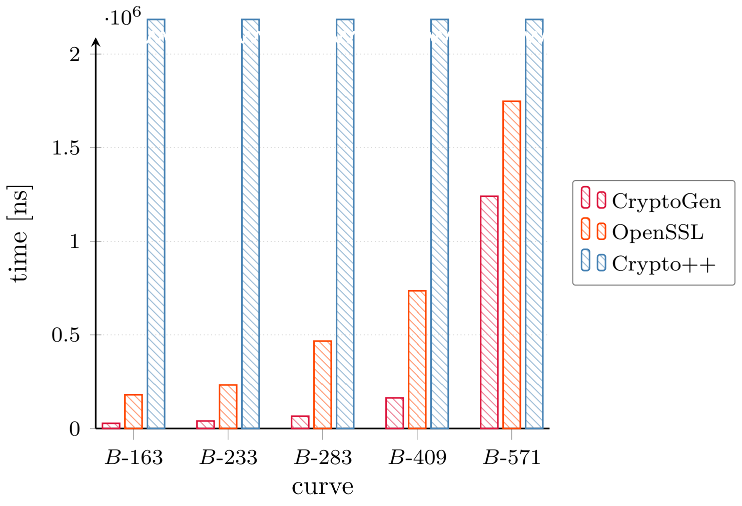


Figure 7.1: Comparison of various scalar multiplication implementations.

The CryptoGen code always outperforms the other libraries. With respect to the OpenSSL, it is from 1.5 to 7.1 times faster, while with respect to the Crypto++ the speed gain ranges from 455.8 to 1123.6.

7.4 Comparison against other curves

Several ECC implementations eschew binary field arithmetic, relying only on elliptic curves defined over prime fields for two main reasons, one practical and one legal. The first reason is related to the performance. An efficient implementation of the scalar multiplication needs a fast field multiplication. In the case of prime fields this can be easily achieved by making use of the hardware integer multiplier, while in the binary field realm the situation is much more complex. Unless the underlying processor supports some special hardware facility (such as the Intel `pclmulqdq` assembler instruction), the binary field multiplication is significantly slower than its prime counterpart. The second reason is due to the fact that, starting from the nineties, several companies, mainly the Certicom and the NSA, began to register several patents on a number of elliptic curve techniques, with several focuses on binary fields and curves[5]. This significantly slowed down the adoption of the ECC and in particular the use of binary curves.

If some good hardware support is available and the right algorithms are selected, a binary curve can be as fast as a prime one, or better. This is showed in Tables 7.8 and 7.9, which list the results of the comparisons for the NIST curves B-233 and B-283, chosen since they are the most diffused ones.

	CryptoGen	OpenSSL			
	B-233	B-233	K-233	P-224	secp224k1
time [ns]	$\boxed{40\,036}$	232 475	222 004	67 475	246 176
ratio	–	5.8	5.5	1.7	6.1

Table 7.8: Comparison of the B-233 curve performance against similar curves.

	CryptoGen	OpenSSL			
	B-283	B-283	K-283	P-256	secp256k1
time [ns]	65 832	466 992	389 727	$\boxed{62\,603}$	276 614
ratio	–	1.1	5.9	1.0	4.2

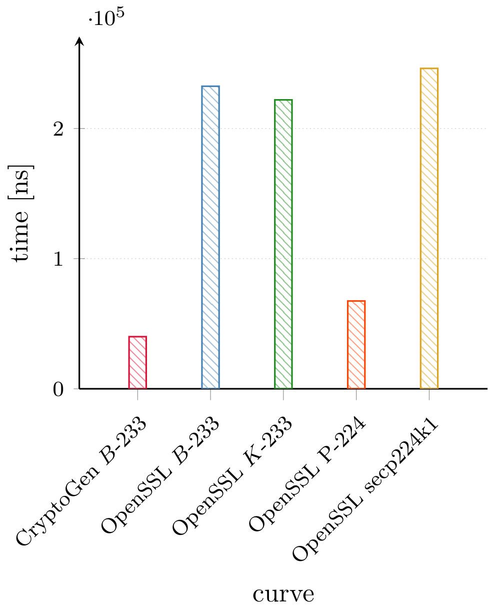
Table 7.9: Comparison of the B-283 curve performance against similar curves.

These tables report the scalar multiplication performance on the i7 processor against several curves with a similar security level [13]. The curves B-233, B-283, K-233 and B-283 are binary curves suggested in the standard *FIPS PUB 186-4 – Digital Signature Standard (DSS)*. On the other hand, the curves P-224, P-256, secp224k1 and secp256k1 are prime curves and the first two are recommended in the *FIPS PUB 186-4 – Digital Signature Standard (DSS)*, while the other two in the *SEC 2: Recommended Elliptic Curve Domain Parameters*. The 'ratio' row indicates the

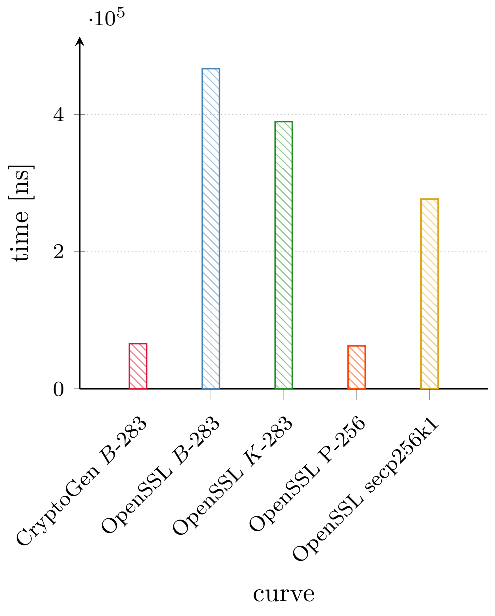
[5]See https://www.certicom.com/pdfs/FAQ-TheNSAECCLicenseAgreement.pdf.

ratio between a specific scalar multiplication time on a curve and the CryptoGen one. As usual the circled values highlight the fastest timings.

Figure 7.2 graphically depicts the performance of the curves.



(a) Comparison of the B-233 curve.



(b) Comparison of the B-283 curve.

Figure 7.2: Comparison of the B-233 and B-283 curve performance against similar curves.

The CryptoGen generated code is always the fastest one except in the B-283 vs P-256 comparison, where it is nearly as fast as its prime counterpart.

Chapter 8

Conclusions and future work

There is no real ending. It's just the place where you stop the story.

FRANK HERBERT

In this dissertation a system for automatically producing high speed elliptic curve cryptography code was presented. It consists of an optimization framework coupled with a number of ECC algorithms written in a high level and hardware independent language. The framework is able to automatically optimize and specialize the cryptographic algorithms into high speed C code and select the fastest implementations for a cryptographic primitive (if more than one implementation is available). Even if the system itself is flexible enough to handle a wide array of number theory algorithms, this thesis focused only on implementing a set of state-of-the-art cryptographic primitives for executing operations on binary fields and binary elliptic curves. These algorithms include also the DCEA, a novel method for computing the multiplicative inversion in binary fields [18, 19].

Two DSLs (HRL and aXiom) for describing cryptographic algorithms in a natural way, close to the traditional mathematical language were described (Chapter 5). These languages natively support various high level data types such as the bit vector, often encountered in number theory implementations. In addition, they permit to specify via annotations several mathematical properties of the functions, increasing the semantic of the code. This allows the compiler to perform a better optimization job by leveraging various algebraic simplifications and identities.

The DCEA (Chapter 3) method, a new algorithm for performing the multiplicative inversion (or division) in binary fields was presented. Based on ideas inspired by λ-calculus, its performance is on par or superior to other state-of-the-art inversion primitives, as experimentally proved.

Finally, a full optimizing compiler (Chapters 4 and 6) was described. It is able to translate and perform a number of hardware independent and dependent optimizations to produce high speed C code with no assistance from the user. These optimization techniques are inspired by the compiler theory field and are able to leverage several mathematical properties of the code and other information that are not available to the traditional C compilers such as GCC.

The experimental results were conducted on five different elliptic curves, on two processors (an i7 and an ARMv7 CPUs) and w.r.t. to two cryptographic libraries (OpenSSL and Crypto++) and various other curves with similar security levels. The speed of the automatically generated code always outperform the tested libraries in all the cases. With respect to the OpenSSL and Crypto++ libraries, the custom scalar multiplication implementation is respectively up to 7.1 times faster and up to 1123.6 times faster. The DCEA algorithm offers superior performance w.r.t. all the other state-of-the-art inversion algorithms on the smallest curve on the ARMv7 processor. In addition, with respect to prime elliptic curves with a similar security level, the generated implementations are faster or have nearly the same performance.

In spite of achieving excellent results in comparison w.r.t. other cryptographic libraries and curves, there is still room for improvements. As a future work, I am planning to expand the current code-improving algorithms for performing inter-basic block and inter-procedural optimizations [109, 111]. Also the support for a particular form of three-address code known as SSA [114] is foreseen. Its use should boost the code-improving capabilities and simplify several type of optimizations. Furthermore, the support for more annotations specifying new and better mathematical properties (e.g. the distributivity) is also a scheduled addition. Another future work is to better leverage the instruction parallelism at low and high levels. For the low level parallelism, the idea is to exploit the SIMD capabilities of some processors (e.g. the SSE instruction set for the AMD64 family) in order to further decrease the bit vector processing time. For the high level parallelism, it is foreseen the support for a multi-threaded programming library (e.g. the OpenMP library[1]), allowing to perform multiple point operations at the same time on concurrent threads. Finally, another idea that will be investigated in the future is to have a special back-end able to produce C code without branches that depend on some secret values (e.g. a key or a nonce). This will make timing attacks much harder to do by sacrificing some computational speed.

This PhD thesis has (hopefully) proved that automatically generating highly optimized cryptographic code is feasible, thus guaranteeing not only less programming errors and development times w.r.t. to a completely manual approach, but also superior performance.

[1]See `http://openmp.org/`.

Appendix A

Proof rules

This appendix lists all the proof rules for the logic families introduced in Chapter 3.

In the following sections, these notations are used:

- the meta-variables denoted with a lower Latin letter (e.g. x and y) range over simple variables;
- the meta-variables consisting of an upper Latin letter (e.g. X and Y) range over formulas;
- the meta-variables denoted with an upper Greek letter (e.g. Γ and Δ) range over list of formulas;
- the symbol τ represent a basic type in a logic system;
- the function $\mathrm{fv}(\Gamma)$ retrieves the set of free variables[1] in Γ;
- the notation $X[x/y]$ represents the *clash free substitution*, where each free occurrence of y in X is replaced by x;
- a sequent in the form $\Gamma|\Delta \vdash X : T$ means that if the variables in Γ are linear (they appear only once) and the variables in Δ are non-linear, then it can be implied that $X : T$.

A.1 CL rules

The proof rules for the Classical Logic (CL) are:

$$\frac{}{X \vdash X}\,\textit{axiom}$$

$$\frac{\Gamma \vdash X, \Delta \quad \Gamma', X \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\,\textit{cut}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, X \vdash \Delta}\,\textit{weakening}_l \qquad \frac{\Gamma \vdash X, \Delta}{\Gamma \vdash \Delta}\,\textit{weakening}_r$$

[1]In λ-calculus, a variable is free in a λ-term when it does not appear as an input parameter in a λ-abstraction. For instance, in $\lambda x.(x + y)$ the variable x is not free (it is *bound*), while y is free.

$$\frac{\Gamma, X, X \vdash \Delta}{\Gamma, X \vdash \Delta}\,contraction_l \qquad \frac{\Gamma \vdash X, X, \Delta}{\Gamma \vdash X, \Delta}\,contraction_r$$

$$\frac{\Gamma, X \vdash \Delta}{\Gamma, X \wedge Y \vdash \Delta}\wedge_{L_1} \qquad \frac{\Gamma, Y \vdash \Delta}{\Gamma, X \wedge Y \vdash \Delta}\wedge_{L_2} \qquad \frac{\Gamma \vdash X, \Delta \quad \Gamma' \vdash Y, \Delta'}{\Gamma, \Gamma' \vdash X \wedge Y, \Delta, \Delta'}\wedge_r$$

$$\frac{\Gamma, X \vdash \Delta \quad \Gamma', Y \vdash \Delta'}{\Gamma, \Gamma', X \vee Y \vdash \Delta, \Delta'}\vee_l \qquad \frac{\Gamma \vdash X, \Delta}{\Gamma \vdash X \vee Y, \Delta}\vee_{R_1} \qquad \frac{\Gamma \vdash Y, \Delta}{\Gamma \vdash X \vee Y, \Delta}\vee_{R_2}$$

$$\frac{\Gamma \vdash X, \Delta \quad \Gamma', Y \vdash \Delta'}{\Gamma, \Gamma', X \Rightarrow Y \vdash \Delta, \Delta'}\Rightarrow_l \qquad \frac{\Gamma, X \vdash Y, \Delta}{\Gamma \vdash X \Rightarrow Y, \Delta}\Rightarrow_r$$

$$\frac{\Gamma, 1 \vdash \Delta}{\Gamma \vdash \Delta}\,1_l \qquad \frac{}{\vdash 1}\,1_r$$

$$\frac{}{0 \vdash}\,0_l \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash 0, \Delta}\,0_r.$$

A.2 IL rules

The proof rules for the Intuitionistic Logic (IL) are:

$$\frac{}{X_1, \ldots, X_n \vdash X_q}\,axiom$$

$$\frac{\Gamma \vdash X \quad \Gamma \vdash Y}{\Gamma \vdash X \wedge Y}\wedge_i \qquad \frac{\Gamma \vdash X \wedge Y}{\Gamma \vdash X}\wedge_{e_1} \qquad \frac{\Gamma \vdash X \wedge Y}{\Gamma \vdash Y}\wedge_{e_2}$$

$$\frac{\Gamma \vdash X}{\Gamma \vdash X \vee Y}\vee_{i_1} \qquad \frac{\Gamma \vdash Y}{\Gamma \vdash X \vee Y}\vee_{i_2} \qquad \frac{\Gamma \vdash X \vee Y \quad \Gamma, X \vdash X \quad \Gamma, Y \vdash X}{\Gamma \vdash X}\vee_e$$

$$\frac{\Gamma, X \vdash Y}{\Gamma \vdash X \Rightarrow Y}\Rightarrow_i \qquad \frac{\Gamma \vdash X \Rightarrow Y \quad \Gamma \vdash X}{\Gamma \vdash Y}\Rightarrow_e.$$

$$\frac{}{\Gamma \vdash 1}\,1_i$$

$$\frac{\Gamma \vdash 0}{\Gamma \vdash X}\,0_e.$$

A.3 LL rules

The proof rules for the Linear Logic (LL) are:

$$\frac{}{X \vdash X}\,axiom$$

$$\frac{\Gamma \vdash X, \Delta \quad \Gamma', X \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\,cut$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, !X \vdash \Delta}\,weakening_l \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?X, \Delta}\,weakening_r$$

$$\frac{\Gamma, !X, !X \vdash \Delta}{\Gamma, !X \vdash \Delta}\,contraction_l \qquad \frac{\Gamma \vdash ?X, ?X, \Delta}{\Gamma \vdash ?X, \Delta}\,contraction_r$$

$$\frac{\Gamma, X, Y \vdash \Delta}{\Gamma, X \otimes Y \vdash \Delta}\otimes_l \qquad \frac{\Gamma \vdash X, \Delta \quad \Gamma' \vdash Y, \Delta'}{\Gamma, \Gamma' \vdash X \otimes Y, \Delta, \Delta'}\otimes_r$$

$$\frac{\Gamma, X \vdash \Delta \quad \Gamma', Y \vdash \Delta'}{\Gamma, \Gamma', X \parr Y \vdash \Delta, \Delta'}\parr_l \qquad \frac{\Gamma \vdash X, Y, \Delta}{\Gamma \vdash X \parr Y, \Delta}\parr_r$$

$$\frac{\Gamma, X \vdash \Delta}{\Gamma, X \,\&\, Y \vdash \Delta}\&_{l_1} \qquad \frac{\Gamma, Y \vdash \Delta}{\Gamma, X \,\&\, Y \vdash \Delta}\&_{l_2} \qquad \frac{\Gamma \vdash X, \Delta \quad \Gamma \vdash Y, \Delta}{\Gamma \vdash X \,\&\, Y, \Delta}\&_r$$

$$\frac{\Gamma, X \vdash \Delta \quad \Gamma, Y \vdash \Delta}{\Gamma, X \oplus Y \vdash \Delta}\oplus_l \qquad \frac{\Gamma \vdash X, \Delta}{\Gamma \vdash X \oplus Y, \Delta}\oplus_{r_1} \qquad \frac{\Gamma \vdash Y, \Delta}{\Gamma \vdash X \oplus Y, \Delta}\oplus_{r_2}$$

$$\frac{\Gamma \vdash X, \Delta \quad \Gamma', Y \vdash \Delta'}{\Gamma, \Gamma', X \multimap Y \vdash \Delta, \Delta'}\multimap_l \qquad \frac{\Gamma, X \vdash Y, \Delta}{\Gamma \vdash X \multimap Y, \Delta}\multimap_r$$

$$\frac{\Gamma, X \vdash \Delta}{\Gamma, !X \vdash \Delta}!_l \qquad \frac{!\Gamma \vdash X, ?\Delta}{!\Gamma \vdash !X, ?\Delta}!_r$$

$$\frac{?\Gamma, X \vdash ?\Delta}{!\Gamma, ?X \vdash ?\Delta}?_l \qquad \frac{\Gamma \vdash X, \Delta}{\Gamma \vdash ?X, \Delta}?_r.$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \top \vdash \Delta}\top_l \qquad \frac{}{\vdash \top}\top_r$$

$$\frac{}{\vdash \bot}\bot_l \qquad \frac{\Gamma \vdash X}{\Gamma \vdash \bot, \Delta}\bot_r$$

$$\frac{}{\Gamma \vdash 1, \Delta}1_r$$

$$\frac{}{\Gamma, 0 \vdash \Delta}0_l.$$

A.4 DLAL rules

The proof rules for the Dual Light Affine Logic (DLAL) are:

$$\frac{}{|x : T \vdash x : T}\,axiom$$

$$\frac{\Gamma|\Delta, x : T_1 \vdash y : T_2}{\Gamma|\Delta \vdash \lambda x.y : T_1 \multimap T_2}\multimap_i \qquad \frac{\Gamma|\Delta \vdash x : T_1 \multimap T_2 \quad \Gamma'|\Delta' \vdash y : T_1}{\Gamma, \Gamma'|\Delta, \Delta' \vdash xy : T_2}\multimap_e$$

$$\frac{\Gamma, x : T_1|\Delta \vdash y : T_2}{\Gamma|\Delta \vdash \lambda x.y : T_1 \Rightarrow T_2}\Rightarrow_i \qquad \frac{\Gamma|\Delta \vdash x : T_1 \Rightarrow T_2 \quad |y : T_3 \vdash z : T_1}{\Gamma, y : T_3|\Delta \vdash xz : T_2}\Rightarrow_e$$

$$\frac{\Gamma|\Delta \vdash x : T_1}{\Gamma, \Gamma'|\Delta, \Delta' \vdash x : T_1}\,weakening \qquad \frac{x : T_1, y : T_1, \Gamma|\Delta \vdash Z : T_2}{x : T_1, \Gamma|\Delta \vdash Z[w/x, w/y] : T_2}\,contraction$$

$$\frac{|\Gamma, \Delta \vdash x : T}{\Gamma | \S\, \Delta \vdash x : \S\, T}\ \S_i \qquad \frac{\Gamma|\Delta \vdash x : \S\, T_1 \quad \Gamma'|y : \S\, T_1, \Gamma' \vdash z : T_2}{\Gamma, \Gamma'|\Delta, \Delta' \vdash z[x/y] : T_2}\ \S_e$$

$$\frac{\Gamma|\Delta \vdash x : T}{\Gamma|\Delta \vdash x : \forall\tau.T}\ \forall_i \qquad \frac{\Gamma|\Delta \vdash x : \forall\tau.T_1}{\Gamma|\Delta \vdash x : T_1[T_2/\tau]}\ \forall_e.$$

A.5 TFA rules

The proof rules for the Typeable Functional Assembly (TFA) are:

$$\frac{}{|x : T \vdash x : T}\ axiom$$

$$\frac{\Gamma|\Delta, x : T_1 \vdash y : T_2}{\Gamma|\Delta \vdash \lambda x.y : T_1 \multimap T_2}\ \multimap_i \qquad \frac{\Gamma|\Delta \vdash x : T_1 \multimap T_2 \quad \Gamma'|\Delta' \vdash y : T_1}{\Gamma, \Gamma'|\Delta, \Delta' \vdash xy : T_2}\ \multimap_e$$

$$\frac{\Gamma, x : T_1|\Delta \vdash y : T_2}{\Gamma|\Delta \vdash \lambda x.y : T_1 \Rightarrow T_2}\ \Rightarrow_i \qquad \frac{\Gamma|\Delta \vdash x : T_1 \Rightarrow T_2 \quad |\Delta' \vdash y : T_1 \quad |\Delta'| \leq 1}{\Delta, \Delta'|\Gamma \vdash xy : T_2}\ \Rightarrow_e$$

$$\frac{\Gamma|\Delta \vdash x : T_1}{\Gamma, \Gamma'|\Delta, \Delta' \vdash x : T_1}\ weakening \qquad \frac{x : T_1, y : T_1, \Gamma|\Delta \vdash Z : T_2}{x : T_1, \Gamma|\Delta \vdash Z[w/x, w/y] : T_2}\ contraction$$

$$\frac{|\Gamma, \Delta \vdash x : T}{\Gamma | \S\, \Delta \vdash x : \S\, T}\ \S_i \qquad \frac{\Gamma|\Delta \vdash x : \S\, T_1 \quad \Gamma'|y : \S\, T_1, \Gamma' \vdash z : T_2}{\Gamma, \Gamma'|\Delta, \Delta' \vdash z[x/y] : T_2}\ \S_e$$

$$\frac{\Gamma|\Delta \vdash x : T \quad \tau \notin \mathrm{fv}(\Delta, \Gamma)}{\Gamma|\Delta \vdash x : \forall\tau.T}\ \forall_i \qquad \frac{\Gamma|\Delta \vdash x : \forall\tau.T_1}{\Gamma|\Delta \vdash x : T_1[T_2/\tau]}\ \forall_e.$$

Appendix B

Grammars

This section lists the grammars of the programming languages described in this thesis. The syntax is expressed by using a set of BNF rules where:

- a production rule has the form x ::= y;
- non-terminals are written unquoted as in x;
- terminals are written quoted as in "`x`";
- the syntax x y indicates the *concatenation* between x and y;
- the syntax x | y indicates the *alternation* between x and y;
- the syntax x^* indicates the *repetition* of x zero or more times (the Kleene closure);
- the syntax x^+ indicates the *repetition* of x one or more times (the positive closure).

For the sake of simplicity, the non-terminals for declaring strings (*String*), identifiers (*Id*), integers (*Int*) and comments are omitted since they follow the common rules used in most of the C-like programming languages such as C++ and Java.

All the following languages are defined by purely left-to-right leftmost derivation grammars, that is they can be efficiently parsed without backtracking.

B.1 HRL grammar

The syntax of the HRL language (Section 5.1.1) is:

Unit ::= "`unit`" *Annotation** *GlobalSymbol**;

Annotation ::= "`:`" *Id* |
    "`:`" *Id* "`(`" (*Id* | *String* | *Int*)$^+$ "`)`"

GlobalSymbol ::= *LocalSymbol* |
    "`function`" *Id* *Function*

LocalSymbol ::= *Id* "`in`" *Variable*

ParameterSymbol ::= *Id* "`in`" *Parameter*

Variable ::= (“wo” | “rw”) *DataType* |
(“wo” | “rw” | “ro”) *DataType* “<-” *Constant*

Parameter ::= (“wo” | “rw” | “ro”) *DataType*;

DataType ::= “logical” |
“[” *Int* “,” *Int* “]” |
Int “of” *DataType*

Constant ::= *Boolean* |
Int |
“(” *Constant*$^+$ “)”

Function ::= *Annotation** *ParameterSymbol**
“{” (*LocalSymbol* | *Instruction* | *LabelSymbol* *Instruction*)* “}”

Instruction ::= *Id* “<-” (*Id* | *Constant*) |
Id “<-” (*Id* | *Constant*) ((*InOperator* (*Id* | *Constant*)) | (“[” (*Id* | *Constant*) “]”)) |
Id “<-” *PreOperator* (*Id* | *Constant*) |
Id “[” (*Id* | *Constant*) “]” “<-” (*Id* | *Constant*) |
“if” (*Id* | *Constant*) “goto” *Id* |
(“goto” | “call”) *Id* |
“push” (*Id* | *Constant*) |
“nop”

LabelSymbol ::= *Id* “:”

PreOperator ::= “not” | “-” | “reduce” | “extend” | “relative” | “findlastset”

InOperator ::= “+” | “-” | “*” | “/” | “\” |
“=” | “/=” | “<” | “<=” | “>” | “>=” |
“<<” | “>>” | “and” | “or” | “xor” | “mod”

Boolean ::= “true” | “false”

B.2 aXiom

The syntax of the aXiom language (Section 5.1.2) is:

Unit ::= (*Symbol** | “unit” *Symbol* | *Annotation** “unit” *Symbol*)*;

Annotation ::= “:” *Id* |
“:” *Id* “(” (*Id* | *String* | *Int*)$^+$ “)”

Symbol ::= *Variable* |
Function

Constant ::= *LogicalConstant* |
RelativeConstant |
TupleConstant

LogicalConstant ::= *Boolean*

RelativeConstant ::= *Int* |
  “`wordsize`”

TupleConstant ::= “`(`” *Constant* (“`,`” *Constant*)* “`)`”

Variable ::= *Id* “`in`” (“`rw`” | “`wo`”) *Type* |
  Id “`in`” (“`rw`” | “`wo`” | “`ro`”) *Type* “`<-`” *Expression* |
  “`static`” *Id* “`in`” “`ro`” *Type* “`<-`” *Expression*

Parameter ::= *Id* “`in`” (“`rw`” | “`wo`” | “`ro`”) *Type*

Type ::= *Logical* |
  Relative |
  Tuple

Logical ::= “`logical`”

Relative ::= “`[`” *Expression* “`,`” *Expression* ,“`]`” |
  “`int8`” | “`uint8`” | “`int16`” | “`uint16`” | “`int32`” | “`uint32`” | “`int64`” | “`uint64`” |
  “`int`” | “`uint`” | “`bit`”

Tuple ::= *Expression* “`of`” *Type*

Function ::= Annotation* “`function`” *Id* “`(`” “`)`”,
  “`{`” (Variable | Instruction)* “`}`” |
  Annotation* “`function`” *Id* “`(`” (*Parameter* | *Parameter* (“`,`” *Parameter*)*) “`)`”
  “`{`” (Variable | Instruction)* “`}`”

Instruction ::= *Copy* |
  If |
  While |
  DoWhile |
  For |
  Call

Copy ::= *PrimaryVariable* “`<-`” *Expression*

If ::= “`if`” *Expression Block* |
  “`if`” *Expression Block* “`else`” *Block* |
  “`static`” “`if`” *Expression Block* |
  “`static`” “`if`” *Expression Block* “`else`” *Block*

While ::= “`while`” *Expression Block*

DoWhile ::= “`do`” *Block* “`while`” *Expression*

For ::= (“`static`” “`for`” | “`for`”) *Expression Block* |
  (“`static`” “`for`” | “`for`”) *Id* “`in`” *Expression Block* |
  (“`static`” “`for`” | “`for`”) *Id* “`in`” “`[`” *Expression* “`,`” *Expression* “`]`” *Block*

Call ::= *Id* "`(`" "`)`" |
Id "`(`" (*Expression* | *Expression* ("`,`" *Expression*)*) "`)`"

Expression ::= *Or*

Or ::= *Xor* ("`or`" *Xor*)*

Xor ::= *And* ("`xor`" *And*)*

And ::= *NotEqualTo* ("`and`" *NotEqualTo*)*

NotEqualTo ::= *EqualTo* ("`/=`" *EqualTo*)*

EqualTo ::= *GreaterThanOrEqualTo* ("`=`" *GreaterThanOrEqualTo*)*

GreaterThanOrEqualTo ::= *GreaterThan* ("`>=`" *GreaterThan*)*

GreaterThan ::= *LessThanOrEqualTo* ("`>`" *LessThanOrEqualTo*)*

LessThanOrEqualTo ::= *LessThan* ("`<=`" *LessThan*)*

LessThan ::= *RightShift* ("`<`" *RightShift*)*

RightShift ::= *LeftShift* ("`>>`" *LeftShift*)*

LeftShift ::= *Subtraction* ("`<<`" *Subtraction*)*

Subtraction ::= *Addition* ("`-`" *Addition*)*

Addition ::= *ModuloReduction* ("`+`" *ModuloReduction*)*

ModuloReduction ::= *Division* ("`mod`" *Division*)*

Division ::= *CeilingDivision* ("`/`" *CeilingDivision*)*

CeilingDivision ::= *Multiplication* ("`\`" *Multiplication*)*

Multiplication ::= *PrimaryOrUnary* ("`*`" *PrimaryOrUnary*)*

PrimaryOrUnary ::= *Not* | *Plus* | *Minus* | *RelativeExpression* | *SizeExpression* |
TupleMacro | *MinMax* | *FindLastSet* | *VariadicExpression* | *TupleExpression* |
Primary

Not ::= "`not`" *PrimaryOrUnary*

Plus ::= "`+`" *PrimaryOrUnary*

Minus ::= "`-`" *PrimaryOrUnary*

RelativeExpression ::= "`relative`" "`(`" *Expression* "`)`"

SizeExpression ::= "`size`" "`(`" *Expression* "`)`"

TupleMacro ::= (“`zeros`” | “`ones`” | “`one`”) “`(`” *Expression* “`)`” |
    (“`tuple`” | “`lowones`” | “`highones`”) “`(`” *Expression* “`,`” *Expression* “`)`”

MinMax ::= (“`min`” | “`max`”) “`(`” *Expression* “`,`” *Expression* “`)`”

FindLastSet ::= “`findlastset`” “`(`” *Expression* “`)`”;

VariadicExpression ::= “`do`” *VariadicOperator* *PrimaryOrUnary*

TupleExpression ::= “`(`” *Expression* (“`,`” *Expression*)$^+$ “`)`”

Primary ::= *Constant* |
    TupleComprehension |
    PrimaryVariable |
    “`(`” *Expression* “`)`”

PrimaryVariable ::= *Id* |
    Id “`[`” *Expression* “`]`” |
    Id “`[`” *Expression* “`,`” *Expression* “`]`”

TupleComprehension ::= “`(`” *Expression* “`|`” *Id* “`in`” “`[`” *Expression* “`,`” *Expression* “`]`” “`)`”

Block ::= “`{`” (Variable | Instruction)* “`}`” |
    Instruction

Boolean ::= “`true`” | “`false`”

VariadicOperator ::= “`or`” | “`xor`” | “`and`” | “`-`” | “`+`” | “`/`” | “`*`” | “`min`” | “`max`”

Appendix C

Annotations

This section contains all the supported annotations shared by HRL (Section 5.1.1) and aXiom (Section 5.1.2).

Table C.1 lists the default unit annotations available with all the middle-ends and back-ends. Tables C.2 and C.3 respectively contain the unit annotations enabled by the aXiom front-end and the C back-end. Finally, Table C.4 reports all the supported function annotations.

synopsis	description
`:algebraicSimplification`	performs an algebraic simplification pass
`:commonSubExpressionElimination`	performs a common sub-expression elimination pass
`:constantFolding`	performs a constant folding pass
`:controlFlowGraphAnalysis`	performs a control flow graph analysis pass
`:copyPropagation`	performs a copy propagation pass
`:dagAnalysis`	performs a DAG analysis pass
`:deadCodeElimination`	performs a dead code elimination pass
`:dumpAll`	creates an HRL file after each transformation pass
`:import(x)`	imports the exported functions contained in the file `x`
`:labelsPrefix(x)`	sets to `x` the prefix of the automatically generated labels' names during the middle-end stage
`:labelsSuffix(x)`	sets to `x` the suffix of the automatically generated labels' names during the middle-end stage
`:liveVariableAnalysis`	performs a live variable analysis pass
`:operatorChainAnalysis`	performs an operator chain analysis pass
`:optimization`	performs all the optimization passes until no code change is possible
`:strengthReduction`	performs a strength reduction pass
`:symbolUsageAnalysis`	performs a symbol usage analysis pass
`:variablesPrefix(x)`	sets to `x` the prefix of the automatically generated variables' names during the middle-end stage
`:variablesSuffix(x)`	sets to `x` the suffix of the automatically generated variables' names during the middle-end stage

Table C.1: Standard unit annotations.

synopsis	description
`:dumpHRL`	creates a file containing the aXiom translation into HRL

Table C.2: Unit annotations offered by the aXiom front-end.

synopsis	description
`:bitVectorPacking`	enables the bit vector packing
`:cLabelsPrefix(x)`	sets to `x` the prefix of the automatically generated labels' names during the C back-end stage
`:cLabelsSuffix(x)`	sets to `x` the suffix of the automatically generated labels' names during the C back-end stage
`:cVariablesPrefix(x)`	sets to `x` the prefix of the automatically generated variables' names during the C back-end stage
`:cVariablesSuffix(x)`	sets to `x` the suffix of the automatically generated variables' names during the C back-end stage
`:findLastSetWindow(x)`	sets the **`findlastset`** window to `x` bit
`:loopUnrolling`	enables the loop unrolling
`:targetIdiomUsage`	enables the target idiom usage

Table C.3: Unit annotations offered by the C back-end.

synopsis	description
`:carryLessMultiplication`	indicates that the function implements a carry-less multiplication
`:carryLessSquaring`	indicates that the function implements a carry-less squaring
`:export`	exports the function, making it callable from the outside of the compilation unit
`:hasAbsorbingElement(x)`	indicates that `x` is an absorbing element for the function
`:hasIdentityElement(x)`	indicates that `x` is an identity element for the function
`:hasSpecialCase1(x, y)`	indicates that if the first input parameter of the function is `x`, then the result is `y`
`:hasSpecialCase2(x, y)`	indicates that if the second input parameter of the function is `x`, then the result is `y`
`:hasSpecialization(x)`	indicates that `x` is a specialized function of the current one, when its input parameters are the same
`:isCommutativeAndAssociative`	indicates that the function is commutative and associative
`:test(x)`	if `x` is `success` indicates that the function is a test case expected to succeed, while if `x` is `failure` the function is a test case expected to fail

Table C.4: Standard function annotations.

Appendix D

Additional results

This appendix lists all the results obtained by executing the framework described in this document.

The specifications of the hardware platforms are available in Table 7.1.

All the tests were executed via the Algorithm 4.1 with 1000 populations of 100 samples and with a sensitivity of 5 %.

The circled values in the tables indicates the best result for a specific operation group and the list of notations is shown in Table 7.4.

If an algorithm name is followed by a slash and a number, this last value indicate the algorithm's window size. For instance, COMB/4 indicates the COMB algorithm with a window of 4 bit.

All the following tests are performed on the fields and elliptic curves specified in the NIST publication *FIPS PUB 186-4 – Digital Signature Standard (DSS)* [13].

D.1 CryptoGen code performance

Table D.1 shows the performance of the binary field and elliptic curve generated code on the i7 processor. Note that the point addition, doubling and negation operations make use of the REDUCE, ADD, WINDOWMULTIPLY/64, WINDOWSQUARE/64 and BEA algorithms since they are the fastest ones on the underlying fields. The scalar multiplication algorithms work in López-Dahab space since it offers superior performance.

Analogously, the Table D.2 reports the performance on the ARMv7 processor. The algorithms WINDOWMULTIPLY and WINDOWSQUARE are not available since these processors lack a carry-less multiplication assembler instruction. The point addition, doubling and negation operations make use of the COMB method with a window width of 3 bit for the two smallest fields and a width of 4 bit for the other ones. In addition, the DCEA algorithm is used for $\mathbb{F}_{2^{163}}$ and the BEA elsewhere. The scalar multiplication instead exploits the López-Dahab coordinates to achieve greater speeds.

D.2 Library code performance

Tables D.3 and D.4 list the performance of the OpenSSL 1.0.2 and Crypto++ 5.6.1 libraries on the NIST binary fields and elliptic curves.

operation	method	time [ns]				
		B-163	*B*-233	*B*-283	*B*-409	*B*-571
$a \bmod b$	Reduce	$\boxed{8}$	$\boxed{6}$	$\boxed{8}$	$\boxed{7}$	$\boxed{17}$
$a+b$	Add	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$
$a \cdot b$	Comb/3	198	352	484	1139	5763
	Comb/4	155	270	364	849	3640
	Comb/5	177	272	365	834	1650
	Karatsuba+Comb/3	183	324	472	916	1541
	Karatsuba+Comb/4	179	286	424	723	1135
	Karatsuba+Comb/5	206	317	451	778	1182
	WindowMultiply/64	$\boxed{7}$	$\boxed{11}$	$\boxed{14}$	$\boxed{35}$	$\boxed{217}$
a^2	PrecomputeSquare/8	14	59	82	159	303
	PrecomputeSquare/9	28	51	76	144	278
	PrecomputeSquare/10	30	47	69	129	258
	WindowSquare/64	$\boxed{5}$	$\boxed{7}$	$\boxed{9}$	$\boxed{25}$	$\boxed{190}$
a^{-1}	BEA	$\boxed{1254}$	$\boxed{2206}$	$\boxed{3005}$	$\boxed{6001}$	$\boxed{11\,507}$
	DCEA	1255	2232	3133	6266	13 172
	EEA	1985	4081	6235	11 199	19 454
$P+Q$	in affine space	1255	2202	3218	6153	12 261
	in projective space	$\boxed{238}$	$\boxed{256}$	$\boxed{354}$	$\boxed{601}$	$\boxed{3301}$
	in Jacobian space	327	330	463	803	4585
	in López-Dahab space	297	311	431	747	4018
$2P$	in affine space	1236	2176	3209	6103	12 122
	in projective space	177	180	243	410	2261
	in Jacobian space	159	160	224	374	2112
	in López-Dahab space	$\boxed{155}$	$\boxed{159}$	$\boxed{223}$	$\boxed{372}$	$\boxed{2036}$
$-P$	in affine space	$\boxed{6}$	$\boxed{8}$	$\boxed{10}$	$\boxed{14}$	$\boxed{19}$
	in projective space	6	8	10	14	19
	in Jacobian space	23	24	35	57	255
	in López-Dahab space	23	24	35	57	255
$n \cdot P$	AddAndDouble	45 817	68 229	112 402	286 026	2 189 789
	NAFMultiply	40 200	59 766	97 893	249 658	1 865 688
	NAFwMultiply/2	28 002	40 898	66 052	164 016	1 256 348
	NAFwMultiply/3	$\boxed{27\,522}$	$\boxed{40\,036}$	$\boxed{65\,832}$	$\boxed{163\,318}$	1 245 761
	NAFwMultiply/4	27 869	40 351	66 116	167 218	$\boxed{1\,240\,570}$
	NAFwMultiply/5	29 277	41 651	68 745	172 167	1 265 471

Table D.1: Performance on the i7 processor.

operation	method	time [ns]				
		B-163	B-233	B-283	B-409	B-571
$a \bmod b$	REDUCE	$\boxed{104}$	$\boxed{120}$	$\boxed{281}$	$\boxed{246}$	$\boxed{565}$
$a + b$	ADD	$\boxed{45}$	$\boxed{69}$	$\boxed{88}$	$\boxed{153}$	$\boxed{239}$
$a \cdot b$	COMB/3	6546	23 038	46 169	80 894	117 895
	COMB/4	4869	14 383	26 327	61 855	113 862
	COMB/5	5207	13 876	24 903	56 404	108 160
	KARATSUBA+COMB/3	4140	8643	15 724	49 130	113 895
	KARATSUBA+COMB/4	$\boxed{3743}$	$\boxed{7266}$	$\boxed{12\,714}$	26 703	82 382
	KARATSUBA+COMB/5	4188	7620	13 078	$\boxed{25\,094}$	$\boxed{77\,572}$
	WINDOWMULTIPLY/32	–	–	–	–	–
a^2	PRECOMPUTESQUARE/8	$\boxed{1621}$	$\boxed{3282}$	$\boxed{5520}$	$\boxed{9280}$	27 605
	PRECOMPUTESQUARE/9	1745	3675	6080	10 006	23 211
	PRECOMPUTESQUARE/10	1865	7620	6742	11 154	$\boxed{19\,431}$
	WINDOWSQUARE/32	–	–	–	–	–
a^{-1}	BEA	32 023	$\boxed{56\,426}$	$\boxed{78\,188}$	$\boxed{166\,485}$	$\boxed{314\,153}$
	DCEA	$\boxed{29\,484}$	57 680	80 239	172 531	321 796
	EEA	35 071	72 281	100 648	204 165	334 727
$P + Q$	in affine space	$\boxed{50\,937}$	$\boxed{70\,045}$	$\boxed{94\,203}$	$\boxed{217\,097}$	$\boxed{429\,549}$
	in projective space	51 920	101 003	139 214	371 128	1 070 419
	in Jacobian space	64 079	123 237	168 047	489 756	1 269 222
	in López-Dahab space	58 365	112 388	157 315	426 454	1 155 117
$2P$	in affine space	36 107	60 348	96 861	223 527	519 679
	in projective space	43 599	69 167	173 192	238 174	621 439
	in Jacobian space	41 867	56 372	88 528	215 989	523 914
	in López-Dahab space	$\boxed{27\,795}$	$\boxed{53\,370}$	$\boxed{83\,751}$	$\boxed{206\,658}$	$\boxed{428\,432}$
$-P$	in affine space	$\boxed{262}$	$\boxed{312}$	$\boxed{333}$	$\boxed{468}$	$\boxed{625}$
	in projective space	262	312	333	468	625
	in Jacobian space	4139	7774	10 586	28 419	80 566
	in López-Dahab space	5744	7756	10 601	28 083	80 703
$n \cdot P$	ADDANDDOUBLE	9 441 774	25 564 965	43 932 628	181 316 012	367 270 712
	NAFMULTIPLY	7 978 762	21 457 574	37 329 096	154 057 350	326 848 597
	NAFWMULTIPLY/2	$\boxed{4\,649\,553}$	$\boxed{12\,443\,090}$	$\boxed{21\,601\,999}$	95 485 598	326 393 165
	NAFWMULTIPLY/3	4 720 973	12 597 703	21 892 988	94 969 061	300 427 337
	NAFWMULTIPLY/4	4 892 246	13 090 895	22 358 341	$\boxed{94\,854\,413}$	293 493 367
	NAFWMULTIPLY/5	5 244 866	13 591 353	23 256 801	97 503 772	$\boxed{287\,818\,708}$

Table D.2: Performance on the ARMv7 processor.

operation	times [ns]				
	B-163	B-233	B-283	B-409	B-571
$a \bmod b$	16	194	186	217	252
$a + b$	18	19	19	20	22
$a \cdot b$	255	280	303	375	501
a^2	241	275	272	329	343
a^{-1}	1955	3354	4768	9927	17 657
$P + Q$	2588	3974	5899	10 486	18 524
$2P$	2543	3892	6043	10 307	18 308
$-P$	25	27	39	27	26
$n \cdot P$	180 007	232 475	466 992	735 405	1 848 002

Table D.3: Performance of the OpenSSL library on the i7 processor.

operation	times [ns]				
	B-163	B-233	B-283	B-409	B-571
$a \bmod b$	1105	1698	2240	3713	6127
$a + b$	97	98	100	99	102
$a \cdot b$	1552	3434	3567	6759	12 922
a^2	130	291	181	229	282
a^{-1}	10 530	14 142	20 724	25 520	46 914
$P + Q$	22 185	30 985	47 295	63 869	131 279
$2P$	21 353	30 274	45 535	63 351	124 767
$-P$	103	103	110	106	112
$n \cdot P$	15 732 963	36 693 675	62 668 174	183 510 370	565 496 848

Table D.4: Performance of the Crypto++ library on the i7 processor.

Notations

The following table lists all the mathematical notations used in this dissertation.

x^*	Kleene closure of x
x^+	positive closure of x
$\circ : x \times y \to z$	binary operator $\circ$ that maps $x \times y$ to z
$\circ : x \to y$	unary operator $\circ$ that maps x to y
$x \mapsto y$	x is mapped to y
$\mathrm{Im}(f)$	image of the function f
$x \leftarrow y$	y is copied into x
$X[y]$	access of the y-th element in the tuple X
$X[y \to]$	access from the y-th element to the z-th element in the tuple X
$x \oplus y$	exclusive or between x and y
$X \gg y$	right shift of X by y positions
$X \ll y$	left shift of X by y positions
$\mathrm{fls}(X)$	find last set of X
$x \leftrightarrow y$	content exchange between the variables x and y
$\{ x, y, z \}$	unordered set of the items x, y and z
$X \smallsetminus Y$	set difference between X and Y
$X \cup Y$	set union of X and Y
$[x, y]$	integer range from x (inclusive) and y (inclusive)
(x, y, z)	ordered tuple of the items x, y and z
$(x, y, z)_2$	bit vector containing the bits x, y and z
$(x, y, z)_{NAF}$	NAF containing the values x, y and z
$(x, y, z)_{NAF_w}$	NAF$_w$ containing the values x, y and z
x_{16}	x in hexadecimal form
$(X, \circ)$	the group X with the group operator $\circ$
$\mathbb{G}$	generic group
$\lvert X \rvert$	cardinality of the set X
$(X, \circ, \square)$	the field X with the field operators $\circ$ and $\square$
$\mathbb{F}$	generic field
$\mathbb{F}_x$	field with order x
$\mathrm{char}(X)$	characteristic of the field X
ψ	field homomorphism
X/Y	extension field X over Y
$\deg(X)$	degree of the field X
$\mathbb{F}_{2^x}$	binary field with degree x
$\mathbb{F}_{2^x}[y]/z$	binary field with degree x in y with modulus z
$\mathcal{E}$	elliptic curve
$\mathcal{E}/X$	elliptic curve $\mathcal{E}$ defined over the field X

$\mathcal{E}(X)$	set of points of the elliptic curve $\mathcal{E}$ defined over the field X
P_∞	point at infinity
(x, y)	point in affine space with coordinates x and y
$(x : y : z)$	point in projective space with coordinates x, y and z
(x, y, z)	point in Jacobian space with coordinates x, y and z
$(x; y; z)$	point in López-Dahab space with coordinates x, y and z
$X \vdash Y$	sequent with antecedent X and consequent Y
$x \wedge y$	logical and between x and y
$x \vee y$	logical r between x and y
$x \Rightarrow y$	logical implication between x and y
$\neg x$	logical not of x
$x \parr y$	multiplicative disjunction between x and y
$x \otimes y$	multiplicative conjunction between x and y
$x \oplus y$	additive disjunction between x and y
$x \,\&\, y$	additive conjunction between x and y
$x \multimap y$	linear implication between x and y
$!\,x$	of course x
$?\,x$	why not x
$\S\,x$	neutral x
$x : y$	typed context of x with type y
$\langle x, y, z \rangle$	threaded words x, y and z
$\mathrm{fv}(X)$	set of the free variables in X
$X[y]$	clash-free substitution of y in X
$x \vert y \vdash z : w$	DLAL sequent with antecedent $x \vert y$ and consequent $z : w$

Acronyms

AES Advanced Encryption Standard 13

AGM Arithmetic Geometric Mean 17

ALU Arithmetic Logic Unit 75

ANSI American National Standards Institute 8, 9

API Application Programming Interface 47, 51

ARPANET Advanced Research Projects Agency NETwork 1

BEA Binary Euclidean Algorithm iv, vi, ix, 24, 25, 32, 33, 38, 39, 41–43, 84

BNF Backus-Naur Form 34–36, 38, 52, 94

BSD Berkeley Software Distribution 49

CACE Computer Aided Cryptography Engineering xii, 6

CAO C And Occam xii, 6, 52

CL Cryptography Language 6

CL Classical Logic iii, v, 34, 35, 90

CPU Central Processing Unit 22, 23, 49, 80–82, 84, 88

DAG Direct Acyclic Graph iv, vi, ix, 65–68, 70, 72, 74, 75, 79, 99

DCEA DLAL Certified Euclidean Algorithm iv, vi, ix, xii, xiii, 19, 24–26, 32, 33, 38, 43, 44, 84, 88

DH Diffie-Hellman xi, 8, 17–19

DLAL Dual Light Affine Logic iii, v, 33, 34, 38, 43, 92, 106, 108

DLP Discrete Logarithm Problem xi, 17

DNS Domain Name System vi, 1, 2, 108

DNSSEC DNS SECurity extensions 1, 2

DS Delegation Signer resource record 2

DSA Digital Signature Algorithm 8, 9, 17

DSL Domain Specific Language iii, xii, xiii, 6, 7, 52, 53, 80, 88

DSP Digital Signal Processor 50

DSS Digital Signature Standard 8

ECC Elliptic Curve Cryptography viii, xi–xiii, 3, 6–11, 21, 23, 25, 28, 33, 38, 45, 86, 88

ECDDHP Elliptic Curve Decision Diffie-Hellman Problem 19

ECDH Elliptic Curve Diffie-Hellman iii, vi, xi, 8, 9, 11, 17–19, 31

ECDHP Elliptic Curve Diffie-Hellman Problem 8, 18, 19

ECDLP Elliptic Curve Discrete Logarithm Problem xi, 8, 9, 17–19, 33

ECDSA Elliptic Curve Digital Signature Algorithm 8–10, 17

ECMQV Elliptic Curve Menezes-Qu-Vanstone 8, 9

EEA Extended Euclidean Algorithm ix, 25, 27, 32

EMF Eclipse Modeling Framework 80

ETSI European Telecommunications Standards Institute 5

FP7 7th Framework Programme xii, 6

FPGA Field Programmable Gate Array 9

GCC GNU Compiler Collection 53, 61, 77, 81, 82, 88

GNU GNU's Not UNIX 109

GSM Global System for Mobile communications 4–6

HR High level intermediate Representation iv, vi, 46–48, 52, 53, 57–60, 62, 63, 65, 75–77, 109

HRL HR Language iv, vi, viii, xiii, 6, 20, 36, 52–56, 58, 59, 63, 65, 66, 72–75, 88, 94, 99, 100

HTTP HyperText Transfer Protocol 3, 109

HTTPS HTTP over SSL or TLS 3

I/O Input/Output 37

ICC Implicit Computational Complexity iii, xiv, 33–44

IDE Integrated Development Environment 80

IEEE Institute of Electrical and Electronics Engineers 8

IETF Internet Engineering Task Force 2

IFP Integer Factorization Problem xi

IL Intuitionistic Logic iii, v, 34, 35, 37, 91

IP Internet Protocol 1, 110

IPsec IP SECurity xi, 9, 18

ISP Internet Service Provider 2

LL Linear Logic iii, v, 35, 38, 91

LSB Least Significant Bit 39, 42

MANET Mobile Ad-hoc NETwork 9

ML MetaLanguage 7

MSB Most Significant Bit 39, 42

MTS Mobile Telephone Service 4

MVC Model-View-Controller 51, 80

NAF Non-Adjacent Form ix, 29, 30, 105, 110, 112

NAF$_w$ width-w NAF ix, 30, 31, 83, 105, 112

NIST National Institute for Standards and Technology 8, 9, 86, 101

NSA National Security Agency 7, 8, 86

OS Operating System 81

PC Personal Computer 5, 50

POSIX Portable Operating System interface for UnIX 49

RAM Random Access Memory 81

RSA Rivest-Shamir-Adleman xi, 2, 8–10

SIDH Supersingular Isogeny Diffie-Hellman xii, 8

SIM Subscriber Identity Module 4, 5

SIMD Single Instruction, Multiple Data 89, 110

SSA Static Single Assignment 89

SSE Streaming SIMD Extensions 89

SSH Secure SHell xi, 9, 18

SSL Secure Socket Layer 85, 109

STLC Simply Typed λ-Calculus 36

STS Station-To-Station 19

SUPERCOP System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives 50

TDC Tele Danmark 2

TFA Typeable Functional Assembly iii–v, 38, 41–43, 93

TLS Transport Layer Security viii, xi, 3, 4, 9, 18, 85, 109

UI User Interface iv, 50, 51

UML Unified Modeling Language vi, 59, 60

URL Uniform Resource Locator 3

VANET Vehicular Ad-hoc NETwork 9

VHDL VHSIC Hardware Description Language 6, 7

VHSIC Very High Speed Integrated Circuit 111

WSN Wireless Sensor Network 9

XML eXtensible Markup Language 50

Index

Bibliography

[1] Mark Jewell. «Encryption Faulted in TJX Hackin». In: *The Washington post* (2007). URL: http://www.washingtonpost.com/wp-dyn/content/article/2007/09/25/AR2007092500836.html (visited on 03/26/2016) (cit. on p. x).

[2] Julianne Pepitone. «5 of the biggest-ever credit card hacks». In: *CNN* (2014). URL: http://money.cnn.com/gallery/technology/security/2013/12/19/biggest-credit-card-hacks/3.html (visited on 03/26/2016) (cit. on p. x).

[3] Whitfield Diffie and Martin Edward Hellman. «New directions in cryptography». In: *IEEE Transactions on Information Theory* 22 (6 1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638 (cit. on p. xi).

[4] Ronald Linn Rivest, Adi Shamir, and Leonard Max Adleman. «A Method for Obtaining Digital Signatures and Public-Key Cryptosystems». In: *Communications of the ACM* 21 (2 1978), pp. 120–126. DOI: 10.1145/359340.359342 (cit. on p. xi).

[5] Taher ElGamal. «A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms». In: *IEEE Transactions on Information Theory* 31 (4 1985), pp. 469–472. DOI: 10.1109/TIT.1985.1057074 (cit. on p. xi).

[6] Victor Saul Miller. «Use of Elliptic Curves in Cryptography». In: *CRYPTO 1985, proceedings of the 5th International Cryptology Conference*. Santa Barbara (USA), 1986. DOI: 10.1007/3-540-39799-X_31 (cit. on pp. xi, 8).

[7] Neal Koblitz. «Elliptic Curve Cryptosystems». In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. DOI: 10.1090/S0025-5718-1987-0866109-5 (cit. on pp. xi, 8).

[8] BSI. *Kryptographische Verfahren: Empfehlungen und Schlussellängen*. Tech. rep. Bundesamt für Sicherheit in der Informationstechnik, 2015. URL: https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm.html (visited on 03/26/2016) (cit. on p. xi).

[9] NSA. *Commercial National Security Algorithm Suite*. Tech. rep. National Security Agency, 2016. URL: https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm (visited on 03/26/2016) (cit. on p. xi).

[10] Kelley Burgin and Michael Peck. *Suite B Profile for Internet Protocol Security (IPsec)*. RFC 6380. RFC Editor, 2011. URL: https://www.ietf.org/rfc/rfc6380.txt (visited on 03/26/2016) (cit. on pp. xi, 9, 18).

[11] Tim Dierks and Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. RFC Editor, 2008. URL: https://www.ietf.org/rfc/rfc5246.txt (visited on 03/26/2016) (cit. on pp. xi, 9, 18).

[12] Douglas Stebila and Jon Green. *Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer*. RFC 5656. RFC Editor, 2009. URL: https://www.ietf.org/rfc/rfc5656.txt (visited on 03/26/2016) (cit. on pp. xi, 9, 18).

[13] NIST. *FIPS PUB 186-4 – Digital Signature Standard (DSS)*. Tech. rep. National Institute for Standards and Technology, 2013. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf (visited on 03/26/2016) (cit. on pp. xi, 9, 18, 20, 80, 86, 101).

[14] Luca de Feo, David Jao, and Jérôme Plût. «Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies». In: *PQCrypto 2011, proceedings of the 4th International Workshop on Post-Quantum Cryptography*. Taipei (China), 2014. DOI: 10.1007/978-3-642-25405-5_2 (cit. on pp. xii, 8).

[15] Daniele Canavese. «Generazione automatica di codice crittografico ottimizzato». M. Eng. thesis. Politecnico di Torino, 2010 (cit. on pp. xii, xiii).

[16] Galois Inc. *Cryptol*. 2015. URL: http://www.cryptol.net (visited on 03/26/2016) (cit. on pp. xii, 7, 52).

[17] Manuel Barbosa et al. *First Steps Toward a Cryptography-Aware Language and Compiler*. Research report. Cryptology ePrint Archive, 2005. URL: https://eprint.iacr.org/2005/160 (visited on 03/26/2016) (cit. on pp. xii, 6, 52).

[18] Daniele Canavese et al. «Can a Light Typing Discipline Be Compatible with an Efficient Implementation of Finite Fields Inversion?» In: *FOPARA 2013, proceedings of the 3rd International Workshop on Foundational and Practical Aspects of Resource Analysis*. Bertinoro (Italy), 2014. DOI: 10.1007/978-3-319-12466-7_3 (cit. on pp. xii, xiii, 11, 24, 33, 34, 38, 40, 84, 88).

[19] Daniele Canavese et al. «Light combinators for finite fields arithmetic». In: *Science of Computer Programming* 111 (3 2015), pp. 365–394. DOI: 10.1016/j.scico.2015.04.001 (cit. on pp. xii, xiii, 24, 33, 34, 38, 43, 88).

[20] Neal Koblitz. «Hyperelliptic cryptosystems». In: *Journal of Cryptology* 1 (3 1989), pp. 139–150. DOI: 10.1007/BF02252872 (cit. on pp. xii, 10).

[21] Gerhard Frey. *How to disguise an elliptic curve*. Talk at ECC 1998, the 2nd Workshop on Elliptit Curve Cryptography. 1998. URL: https://cr.yp.to/bib/1998/frey-disguise.ps (visited on 03/26/2016) (cit. on pp. xii, 10).

[22] Gerhard Frey. «Applications of arithmetical geometry to cryptographic constructions». In: *FFA 1999, proceedings of 5h International Conference on Finite Fields and Applications*. Augsburg (Germany), 2001. DOI: 10.1007/978-3-642-56755-1_13 (cit. on pp. xii, 10, 17).

[23] CORDIS. *European Commission: CORDIS: Projects & Results Service: Computer Aided Cryptography Engineering*. 2011. URL: http://cordis.europa.eu/project/rcn/85344_en.html (visited on 03/26/2016) (cit. on pp. xii, 6).

[24] Roberto Avanzi et al. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2005. ISBN: 1584885181 (cit. on pp. xiii, 11, 27).

[25] Darrel Hankerson, Alfred John Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2003. ISBN: 038795273X (cit. on pp. xiii, 11, 20, 21, 23).

[26] Paul Mockapetris. *Domain names – concepts and facilities*. RFC 882. RFC Editor, 1983. URL: https://www.ietf.org/rfc/rfc882.txt (visited on 03/26/2016) (cit. on p. 1).

[27] P. Mockapetris. *Domain names – implementation and specification*. RFC 883. RFC Editor, 1983. URL: https://www.ietf.org/rfc/rfc883.txt (visited on 03/26/2016) (cit. on p. 1).

[28] Donald Eastlake. *Domain Name System Security Extensions*. RFC 2535. RFC Editor, 1999. URL: https://www.ietf.org/rfc/rfc2525.txt (visited on 03/26/2016) (cit. on p. 1).

[29] Roy Arends et al. *DNS Security Introduction and Requirements*. RFC 4033. RFC Editor, 2005. URL: https://www.ietf.org/rfc/rfc4033.txt (visited on 03/26/2016) (cit. on p. 2).

[30] Roy Arends et al. *Resource Records for the DNS Security Extensions.* RFC 4034. RFC Editor, 2005. URL: https://www.ietf.org/rfc/rfc4034.txt (visited on 03/26/2016) (cit. on p. 2).

[31] Roy Arends et al. *Protocol Modifications for the DNS Security Extensions.* RFC 4035. RFC Editor, 2005. URL: https://www.ietf.org/rfc/rfc4035.txt (visited on 03/26/2016) (cit. on p. 2).

[32] Olafur Gudmundsson. *Delegation Signer (DS) Resource Record (RR).* RFC 3658. RFC Editor, 2003. URL: https://www.ietf.org/rfc/rfc3658.txt (visited on 03/26/2016) (cit. on p. 2).

[33] Caroline Ahlström. *Ökad säkerhet för dig som är bredbandskund hos TDC Song.* 2007. URL: http://wpy.observer.se/wpyfs/00/00/00/00/00/09/35/D0/release.html (visited on 03/26/2016) (cit. on p. 2).

[34] ICANN. *Status Update, 2010-07-16.* 2010. URL: http://www.root-dnssec.org/2010/07/16/status-update-2010-07-16/ (visited on 03/26/2016) (cit. on p. 2).

[35] ICANN. *TLD DNSSEC Report (2016-03-26 12:02:41).* 2016. URL: http://stats.research.icann.org/dns/tld_report/ (visited on 03/26/2016) (cit. on p. 2).

[36] APNIC. *Use of DNSSEC Validation for World (XA).* 2016. URL: http://stats.labs.apnic.net/dnssec/XA (visited on 03/26/2016) (cit. on p. 2).

[37] Daniel Julius Bernstein. *DNSCurve: Usable security for DNS.* 2016. URL: https://dnscurve.org/ (visited on 03/26/2016) (cit. on p. 2).

[38] OpenDNS. *OpenDNS adopts DNSCurve.* 2010. URL: https://blog.opendns.com/2010/02/23/opendns-dnscurve/ (visited on 03/26/2016) (cit. on p. 2).

[39] Evan Roseman. *Search more securely with encrypted Google web search.* 2010. URL: https://googleblog.blogspot.it/2010/05/search-more-securely-with-encrypted.html (visited on 03/26/2016) (cit. on p. 3).

[40] Evelyn Kao. *Making search more secure.* 2011. URL: https://googleblog.blogspot.it/2011/10/making-search-more-secure.html (cit. on p. 3).

[41] Danny Sullivan. *Post-PRISM, Google Confirms Quietly Moving To Make All Searches Secure, Except For Ad Clicks.* 2013. URL: http://searchengineland.com/post-prism-google-secure-searches-172487 (visited on 03/26/2016) (cit. on p. 3).

[42] Dan Dulay. *Out with the old: Stronger certificates with Google Internet Authority G2.* 2013. URL: https://security.googleblog.com/2013/11/out-with-old-stronger-certificates-with.html (visited on 03/26/2016) (cit. on p. 3).

[43] Adi Shamir and Eran Tromer. «On the Cost of Factoring RSA-1024». In: *RSA CryptoByte* 6 (2 2003), pp. 10–19 (cit. on p. 3).

[44] Daniel Julius Bernstein and Tan. *Security dangers of the NIST curves.* 2013. URL: https://www.hyperelliptic.org/tanja/vortraege/20130531.pdf (visited on 03/26/2016) (cit. on p. 3).

[45] ETSI. *GSM UMTS 3GPP Numbering Cross Reference.* 2016. URL: http://webapp.etsi.org/key/key.asp?full_list=y (visited on 03/26/2016) (cit. on p. 4).

[46] Wagner, Goldberg, and Briceno. *GSM Cloning.* 1998. URL: http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.html (visited on 03/26/2016) (cit. on p. 5).

[47] Alex Biryukov, Adi Shamir, and David Wagner. «Real Time Cryptanalysis of A5/1 on a PC». In: *FSE 2000, proceedings of the 7th International Workshop on Fast Software Encryption.* New York (USA), 1999. DOI: 10.1007/3-540-44706-7_1 (cit. on p. 5).

[48] Ian Goldberg, David Wagner, and Lucky Green. *The (Real-Time) Cryptanalysis of A5/2.* Talk at CRYPTO 1999, the 19th International Cryptology Conference. 1999. URL: `www.cs.berkeley.edu/~daw/tmp/a52-slides.ps` (visited on 03/26/2016) (cit. on p. 5).

[49] Mitsuru Matsui. «New Block Encryption Algorithm MISTY». In: *FSE 1997, proceedings of the 4th International Workshop on Fast Software Encryption.* Haifa (Israel), 1997. DOI: `10.1007/BFb0052334` (cit. on p. 5).

[50] Orr Dunkelman, Nathan Keller, and Adi Shamir. *A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony.* Research report. Cryptology ePrint Archive, 2010. URL: `http://eprint.iacr.org/2010/013` (visited on 03/26/2016) (cit. on p. 5).

[51] ETSI. *Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 5: Design and Evaluation Report.* Tech. rep. European Telecommunications Standards Institute, 2007. URL: `http://www.3gpp.org/ftp/Specs/archive/35_series/35.919/35919-100.zip` (visited on 03/26/2016) (cit. on p. 5).

[52] Aleksandar Kircanski and Amr Youssefr. «On the Sliding Property of SNOW 3G and SNOW 2.0». In: *IET Information Security* 5 (4 2012), pp. 199–206. DOI: `10.1049/iet-ifs.2011.0033` (cit. on p. 5).

[53] Cisco Visual Networking Index. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015–2020 White Paper.* Tech. rep. Cisco, 2016. URL: `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html` (visited on 03/26/2016) (cit. on p. 6).

[54] Matthew Smith et al. «Identity-Based Cryptography for Securing Mobile Phone Calls». In: *WAINA 2009, proceedings of the 27th International Conference on Advanced Information Networking and Applications Workshops.* Bradford (UK), 2009. DOI: `10.1109/WAINA.2009.167` (cit. on p. 6).

[55] Sukalyan Goswami et al. «Enhancement of GSM Security Using Elliptic Curve Cryptography Algorithm». In: *ISMS 2012, proceedings of the 3rd International Conference on Intelligent Systems, Modelling and Simulation.* Kota Kinabalu (Malaysia), 2012. DOI: `10.1109/ISMS.2012.137` (cit. on p. 6).

[56] Mohamed Hassan Abdelrahman, Ihab Elsayed Talkhan, and Samir Ibrahim Shaheen. «Crypto-Algorithms Maker kit». In: *ICM 2003, proceedings of the 15th International Conference on Microelectronics.* Cairo (Egypt), 2003. DOI: `10.1109/ICM.2003.1287784` (cit. on p. 6).

[57] Daniel Julius Bernstein. *Writing high-speed software.* 2007. URL: `http://cr.yp.to/qhasm.html` (visited on 03/26/2016) (cit. on p. 6).

[58] Peter Schwabe. *maq - a preprocessor for qhasm.* 2015. URL: `https://cryptojedi.org/programming/maq.shtml` (visited on 03/26/2016) (cit. on p. 6).

[59] Luis Julian Dominguez Perez and Michael Scott. «Designing a Code Generator for Pairing Based Cryptographic Functions». In: *PAIRING 2010, proceedings of the 4th International Conference on Pairing-Based Cryptography.* Yamanaka (Japan), 2010. DOI: `10.1007/978-3-642-17455-1_14` (cit. on p. 7).

[60] Clifford Scott Ananian. «Reconfigurable Cryptography: A Hardware Compiler for Cryptographic Applications». 1997. URL: `http://cscott.net/Projects/ele580a/writeup.pdf` (visited on 03/26/2016) (cit. on p. 7).

[61] Andrew Wilson Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998. ISBN: 0-521-60765-5 (cit. on p. 7).

[62] Stefan Lucks, Nico Schmoigl, and Emin Islam Tatl. «Issues on Designing a Cryptographic Compiler». In: *WEWoRC 2005, proceedings of the 1st Western European Workshop on Research in Cryptology*. Leuven (Belgium), 2005 (cit. on p. 7).

[63] Karthikeyan Bhargavan et al. «Cryptographic Protocol Synthesis and Verification for Multiparty Sessions». In: *CSF 2009, proceedings of the 22nd Computer Security Foundations Symposium*. Port Jefferson (USA), 2008. DOI: 10.1109/CSF.2009.26 (cit. on p. 7).

[64] Endre Bangerter et al. «cPLC — A cryptographic programming language and compiler». In: *ISSA 2011, proceedings of 4h Conference on Information Security for South Africa*. Johannesburg (South Africa), 2011. DOI: 10.1109/ISSA.2011.6027533 (cit. on p. 7).

[65] José Bacelar Almeida et al. «Full Proof Cryptography: Verifiable Compilation of Efficient Zero-Knowledge Protocols». In: *CCS 2012, proceedings of the 19th Conference on Computer and Communications Security*. Raleigh (USA), 2012. DOI: 10.1145/2382196.2382249 (cit. on p. 7).

[66] Oded Goldreich, Silvio Micali, and Avi Wigderson. «Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems». In: *Journal of the ACM* 38 (3 1991), pp. 690–728. DOI: 10.1145/116825.116852 (cit. on p. 7).

[67] Ronald Linn Rivest et al. «Responses to NIST's Proposal». In: *Communications of the ACM* 35 (7 1992), pp. 41–54. DOI: 10.1145/129902.129905 (cit. on p. 8).

[68] René Schoof. «Counting points on elliptic curves over finite fields». In: *Journal de Théorie des Nombres de Bordeaux* 7 (1 1995), pp. 219–254. DOI: 10.5802/jtnb.142 (cit. on p. 8).

[69] Dan Boneh and Richard Lipton. «Searching for elements in black box fields and applications». In: *CRYPTO 1996, proceedings of the 16th International Cryptology Conference*. Santa Barbara (USA), 1996. DOI: 10.1007/3-540-68697-5_22 (cit. on p. 8).

[70] Laurie Law et al. «An Efficient Protocol for Authenticated Key Agreement». In: *Designs, Codes and Cryptography* 28 (2 1998), pp. 119–134. DOI: 10.1023/A:1022595222606 (cit. on p. 8).

[71] NSA. *NSA Suite B Cryptography*. Tech. rep. National Security Agency, 2005. URL: https://www.nsa.gov/ia/programs/suiteb_cryptography/ (visited on 03/26/2016) (cit. on p. 8).

[72] IEEE. *IEEE Standard Specifications for Public-Key Cryptography*. Tech. rep. Institute of Electrical and Electronics Engineers, 2000. DOI: 10.1109/IEEESTD.2000.92292 (cit. on p. 8).

[73] ANSI. *ANSI X9.62:2005, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. Tech. rep. American National Standards Institute, 2005 (cit. on p. 8).

[74] ECC Brainpool. *ECC Brainpool Standard Curves and Curve Generation*. Tech. rep. ECC Brainpool, 2005. URL: http://www.ecc-brainpool.org/download/Domain-parameters.pdf (visited on 03/26/2016) (cit. on p. 8).

[75] Daniel Julius Bernstein. «Curve25519: new Diffie-Hellman speed records». In: *PKC 2006, proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography*. New York (USA), 2006. DOI: 10.1007/11745853_14 (cit. on p. 9).

[76] Certicom Research. *SEC 2: Recommended Elliptic Curve Domain Parameters*. Tech. rep. Certicom Research, 2010. URL: http://www.secg.org/sec2-v2.pdf (visited on 03/26/2016) (cit. on pp. 9, 86).

[77] ANSI. *ANSI X9.62:2011, Public Key Cryptography for the Financial Services Industry - Key Agreement and Key Transport Using Elliptic Curve Cryptography*. Tech. rep. American National Standards Institute, 2011 (cit. on p. 9).

[78] An Liu and Peng Ning. «TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks». In: *IPSN 2008, proceedings of the 7th International Conference on Information Processing in Sensor Networks*. St. Louis (USA), 2008. DOI: 10.1109/IPSN.2008.47 (cit. on p. 9).

[79] Ravi Kishore Kodali et al. «Fast elliptic curve point multiplication for WSNs». In: *TENCON 2013, proceedings of 28th International Technical Conference of IEEE Region 10*. Sydney (Australia), 2013. DOI: 10.1109/TENCONSpring.2013.6584439 (cit. on p. 9).

[80] Muhammad Hammad Ahmed et al. «Security for WSN based on elliptic curve cryptography». In: *ICCNIT 2001, proceedings of the 1st International Conference on Computer Networks and Information Technology*. Bara Gali (Pakistan), 2011. DOI: 10.1109/ICCNIT.2011.6020911 (cit. on p. 9).

[81] Hisham Dahshan and James Irvine. «An Elliptic Curve Distributed Key Management for Mobile Ad Hoc Networks». In: *VTC-Spring 2010, proceedings of 71st Vehicular Technology Conference*. Taipei (China), 2010. DOI: 10.1109/VETECS.2010.5494203 (cit. on p. 9).

[82] Sunilkumar Manvi, Mahabaleshwar Kakkasageri, and D. G. Adiga. «Message Authentication in Vehicular Ad Hoc Networks: ECDSA Based Approach». In: *ICFCC 2009, proceedings of the 1st International Conference on Future Computer and Communication*. Kuala Lumpar (Malaysia), 2009. DOI: 10.1109/ICFCC.2009.120 (cit. on p. 9).

[83] Jiun-Long Huang, Lo-Yao Yeh, and Hung-Yu Chien. «ABAKA: An Anonymous Batch Authenticated and Key Agreement Scheme for Value-Added Services in Vehicular Ad Hoc Networks». In: *IEEE Transactions on Vehicular Technology* 60 (1 2011), pp. 248–262. DOI: 10.1109/TVT.2010.2089544 (cit. on p. 9).

[84] Robert McEliece. «A Public-Key Cryptosystem Based On Algebraic Coding Theory». In: *Deep Space Network Progress Report* 44 (1978), pp. 114–116. URL: http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF (visited on 03/26/2016) (cit. on p. 10).

[85] Daniel Julius Bernstein, Tanja Lange, and Christiane Peters. «Attacking and defending the McEliece cryptosystem». In: *PQCrypto 2008, proceedings of the 2nd International Workshop on Post-Quantum Cryptography*. Cincinnati (USA), 2008. DOI: 10.1007/978-3-540-88403-3_3 (cit. on p. 10).

[86] Jacques Patarin. «Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new Families of Asymmetric Algorithms». In: *EUROCRYPT 1996, proceedings of the 15th International Conference on the Theory and Application of Cryptographic Techniques*. Zaragoza (Spain), 2001. DOI: 10.1007/3-540-68339-9_4 (cit. on p. 10).

[87] Andrew John Wiles. «Modular elliptic curves and Fermat's Last Theorem». In: *Annals of Mathematics* 141 (3 1995), pp. 443–551. DOI: 10.2307/2118559 (cit. on p. 11).

[88] Kenny Fong et al. «Field Inversion and Point Halving Revisited». In: *IEEE Transactions on Computers* 53 (8 2004), pp. 1047–1059. DOI: 10.1109/TC.2004.43 (cit. on pp. 11, 24).

[89] NIST. *FIPS PUB 197 – Advanced Encryption Standard (AES)*. Tech. rep. National Institute for Standards and Technology, 2001. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (visited on 03/26/2016) (cit. on p. 13).

[90] Tetsuya Izu et al. «Efficient Implementation of Schoof's Algorithm». In: *ASIACRYPT 1998, proceedings of the 4th International Conference on the Theory and Application of Cryptology and Information Security*. Beijing (China), 1998. DOI: 10.1007/3-540-49649-1_7 (cit. on p. 17).

[91] T. Satoh. «The canonical lift of an ordinary elliptic curve over a prime field and its point counting». In: *Journal of the Ramanujan Mathematical Society* 15 (1 2000), pp. 247–270 (cit. on p. 17).

[92] Pierrick Gaudry. «A comparison and combination of SST and AGM algorithms for counting points of elliptic curves in characteristic 2». In: *ASIACRYPT 2002, proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security*. Queenstown (New Zealand), 2002. DOI: 10.1007/3-540-36178-2_20 (cit. on p. 17).

[93] Daniel Shanks. «Class Number, a Theory of Factorization, and Genera». In: *PSPUM 1969, proceedings of the 10th Symposia in Pure Mathematics*. New York (USA), 1971. DOI: 10.1090/pspum/020 (cit. on p. 17).

[94] John Michael Pollard. «A Monte Carlo method for factorization». In: *BIT Numerical Mathematics* 15 (3 1975), pp. 331–334. DOI: 10.1007/BF01933667 (cit. on p. 17).

[95] Gerhard Frey and Hans Georg Rück. «A remark concerning m-divisibility and the discrete logarithm problem in the divisor class group of curves». In: *Journal of Mathematics of Computation* 62 (206 1994), pp. 865–874. DOI: 10.2307/2153546 (cit. on p. 17).

[96] Stephen Pohlig and Martin Edward Hellman. «An improved algorithm for computing logarithms over GF(p) and its cryptographic significance». In: *IEEE Transactions on Information Theory* 24 (1 1978), pp. 106–110. DOI: 10.1109/TIT.1978.1055817 (cit. on p. 17).

[97] Alfred John Menezes, Tatsuaki Okamoto, and Scott Vanstone. «Reducing elliptic curve logarithms to a finite field». In: *IEEE Transactions on Information Theory* 39 (5 1993). DOI: 10.1109/18.259647 (cit. on p. 17).

[98] Victor Shoup. «Lower Bounds for Discrete Logarithms and Related Problems». In: *EUROCRYPT 1997, proceedings of the 16th International Conference on the Theory and Application of Cryptographic Techniques*. Konstanz (Germany), 1997. DOI: 10.1007/3-540-69053-0_18 (cit. on p. 19).

[99] Anatolii Alexeevich Karatsuba and Yuri Ofman. «Multiplication of multidigit numbers on automata». In: *Soviet Physics Doklady* 7 (7 1963), pp. 595–596 (cit. on p. 22).

[100] William Alvin Howard. «The formulae-as-type notion of construction». 1969. URL: http://www.cs.rhul.ac.uk/~zhaohui/Howard80.pdf (visited on 03/26/2016) (cit. on p. 33).

[101] Emanuele Cesena, Marco Pedicini, and Luca Roversi. «Typing a Core Binary Field Arithmetic in a Light Logic». In: *FOPARA 2011, proceedings of the 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis*. Madrid (Spain), 2011. DOI: 10.1007/978-3-642-32495-6_2 (cit. on pp. 33, 34, 38).

[102] Torben Braüner. «Introduction to Linear Logic». 1996. URL: http://www.brics.dk/LS/96/6/BRICS-LS-96-6.pdf (visited on 03/26/2016) (cit. on pp. 33, 35).

[103] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*. Cambridge University Press, 1989. ISBN: 0-521-37181-3 (cit. on pp. 33, 40).

[104] Patrick Baillot and Kazushige Teruib. «Light types for polynomial time computation in lambda calculus». In: *Information and Computation* 207 (1 2008), pp. 41–62. DOI: 10.1016/j.ic.2008.08.005 (cit. on pp. 34, 38).

[105] Alan Mathison Turing. «Computability and λ-Definability». In: *The Journal of Symbolic Logic* 2 (4 1937), pp. 153–163. DOI: 10.2307/2268280 (cit. on p. 36).

[106] Graham Upton and Ian Cook. *Understanding Statistics*. Oxford University Press, 1996. ISBN: 9780199143917 (cit. on p. 49).

[107] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Tech. rep. Intel Corporation, 2010. URL: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (visited on 03/26/2016) (cit. on p. 50).

[108] VAMPIRE Research Lab. *SUPERCOP*. 2015. URL: `http://bench.cr.yp.to/supercop.html` (visited on 03/26/2016) (cit. on p. 50).

[109] Alfred Vaino Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006. ISBN: 978-0321486813 (cit. on pp. 53, 61, 63, 71, 89).

[110] GNU. *GIMPLE - GNU Compiler Collection (GCC) Internals*. 2015. URL: `https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html` (visited on 03/26/2016) (cit. on p. 53).

[111] Ken Kennedy and John Randal Allen. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2001. ISBN: 1-55860-286-0 (cit. on pp. 61, 70, 71, 89).

[112] Shay Gueron and Michael Kounavis. *Intel©Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. Tech. rep. Intel Corporation, 2014. URL: `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/carry-less-multiplication-instruction-in-gcm-mode-paper.pdf` (visited on 03/26/2016) (cit. on p. 77).

[113] Daniel Julius Bernstein and Tanja Lange. *hyperelliptic.org*. 2016. URL: `http://hyperelliptic.org` (visited on 03/26/2016) (cit. on p. 77).

[114] Ron Cytron et al. «Efficiently computing static single assignment form and the control dependence graph». In: *ACM Transactions on Programming Languages and Systems* 13 (4 1991), pp. 451–490. DOI: `10.1145/115372.115320` (cit. on p. 89).