Automatic generation of high speed elliptic curve cryptography code

Original
Automatic generation of high speed elliptic curve cryptography code / Canavese, Daniele. - (2016).

Availability:
This version is available at: 11583/2652694 since: 2016-10-11T14:13:00Z

Publisher:
Politecnico di Torino

Published
DOI:10.6092/polito/porto/2652694

Terms of use:
Altro tipo di accesso
This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)
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 vi
List of Tables viii
List of Algorithms ix

Introduction x

1 High speed cryptography 1
  1.1 Real world applications ........................................ 1
  1.1.1 Domain name system security ................................ 1
  1.1.2 Google encrypted search .................................... 3
  1.1.3 Mobile phone protection .................................... 4
  1.2 Related works .................................................. 6
  1.2.1 Cryptography-aware DSLs and compilers .................... 6
  1.2.2 Elliptic curve cryptography ................................ 7
  1.2.3 Other high speed cryptographic techniques ................ 10

2 Elliptic curve cryptography 11
  2.1 Mathematical background ....................................... 11
  2.1.1 Groups ...................................................... 11
  2.1.2 Fields ...................................................... 12
  2.1.3 Elliptic curves ............................................. 13
  2.1.4 Security of elliptic curve crypto-systems .................. 16
  2.1.5 ECDH ....................................................... 18
  2.2 Algorithmic background ........................................ 19
  2.2.1 Binary field arithmetic .................................... 20
  2.2.2 Elliptic curve arithmetic .................................. 25
  2.2.3 ECDH ....................................................... 31
  2.2.4 Summary of decisions ...................................... 32

3 Implicit computational complexity 33
  3.1 Logic background ............................................... 34
  3.1.1 Classical logic ............................................. 34
  3.1.2 Intuitionistic logic ......................................... 34
  3.1.3 Linear logic ............................................... 35
  3.1.4 λ-calculus .................................................. 36
  3.1.5 Dual light affine logic .................................... 38
  3.1.6 Typeable functional assembly .............................. 38
## List of Figures

1. Suggested key lengths for 2016. ................................................................. xi
1.1 DNS security time-line. ........................................................................... 2
1.2 Google encrypted search time-line. ......................................................... 3
1.3 Mobile phone security time-line. ............................................................. 5

2.1 Plots of two elliptic curves over \( \mathbb{R} \). .................................................. 14
2.2 Point addition and doubling in \( \mathbb{F}/\mathbb{R} \). .................................................. 15
2.3 Steps of the ECDH protocol. ................................................................. 18
2.4 Man-in-the-middle attack on the ECDH protocol. ............................... 19
2.5 Algorithm choices for the scalar multiplication implementation. ......... 32

3.1 Example of functional right shift. ......................................................... 40
3.2 BEA and DCEA in comparison. ............................................................ 43

4.1 General architecture of the framework. ............................................... 45
4.2 Architecture of the cryptographic compiler. ........................................... 47
4.3 Architecture of the C back-end. ............................................................ 48
4.4 A sample library configuration file. ......................................................... 51

5.1 Two equivalent snippets, in C and HRL. .............................................. 54
5.2 Two equivalent snippets, in C and aXiom. ............................................ 57
5.3 An aXiom snippet and its output HRL code. ....................................... 58
5.4 The modular reduction algorithm in aXiom. ........................................ 58
5.5 UML class diagram of HR. ................................................................... 60

6.1 Example of a control flow graph. ......................................................... 63
6.2 DAG representation for the \((2 \cdot x + x)/y\) expression. ..................... 65
6.3 Example of the DAG for a basic block. ............................................... 65
6.4 Example of the DAG for some tuple instructions. ............................... 66
6.5 Example of an operator chain. ............................................................. 66
6.6 Constant folding example. ................................................................. 68
6.7 Algebraic simplification example. ....................................................... 69
6.8 Copy propagation example. ............................................................... 70
6.9 Dead code elimination example. .......................................................... 71
6.10 Example of hidden strength reduction possibilities. ......................... 72
6.11 Strength reduction example. .............................................................. 72
6.12 Common sub-expression elimination example. .................................. 74
6.13 Bit vector packing example. .............................................................. 75
6.14 Loop unrolling example. ................................................................. 76
7.1 Comparison of various scalar multiplication implementations. . . . . . . . . . . . . . 85
7.2 Comparison of the $B_{-233}$ and $B_{-283}$ curve performance against similar curves. . . 87
List of Tables

1.1 Supposed TLS 1.2 cipher-suite priorities for Google Search in 2016. ................. 4
1.2 Protocol supports of the ECC algorithms. .................................................. 9

2.1 Elliptic curve formulas for various coordinate systems. ............................. 28
2.2 Elliptic curve costs for various coordinate systems. ................................. 29
2.3 Elliptic curve costs for various scalar multiplication algorithms. ................. 31

5.1 The instructions available in the HRL language. ....................................... 55

6.1 CryptoGen algebraic simplification rules. ................................................ 69

7.1 Specifications of the tested hardware platforms. ....................................... 81
7.2 Specifications of the tested elliptic curves and fields. ............................... 81
7.3 Effects of the optimizations on the B-233 curve’s scalar multiplication code. .... 82
7.4 Legend of the symbols. ................................................................................. 83
7.5 Performances on the i7 processor. ............................................................. 83
7.6 Performances on the ARMv7 processor. ..................................................... 84
7.7 Scalar multiplication performance on different implementations. ................. 85
7.8 Comparison of the B-233 curve performance against similar curves. ............ 86
7.9 Comparison of the B-283 curve performance against similar curves. ............. 86

C.1 Standard unit annotations. .......................................................................... 99
C.2 Unit annotations offered by the aXiom front-end. ....................................... 100
C.3 Unit annotations offered by the C back-end. ............................................. 100
C.4 Standard function annotations. ................................................................. 100

D.1 Performance on the i7 processor. ............................................................. 102
D.2 Performance on the ARMv7 processor. ..................................................... 103
D.3 Performance of the OpenSSL library on the i7 processor. ......................... 104
D.4 Performance of the Crypto++ library on the i7 processor. ......................... 104
List of Algorithms

2.1 Binary field modular reduction algorithm. ................................................. 20
2.2 Binary field addition algorithm. ............................................................... 21
2.3 Binary field comb multiplication algorithm. ............................................. 21
2.4 Binary field Karatsuba-Ofman multiplication algorithm. ......................... 22
2.5 Binary field window-based multiplication algorithm. .................................. 23
2.6 Binary field precomputation-based squaring algorithm. ............................ 23
2.7 Binary field window-based squaring algorithm. ......................................... 24
2.8 Binary field BEA inversion algorithm. ...................................................... 25
2.9 Binary field DCEA inversion algorithm. .................................................... 26
2.10 Binary field EEA inversion algorithm. ...................................................... 27
2.11 Elliptic curve add-and-double scalar multiplication algorithm. ................. 29
2.12 NAF computation algorithm. ................................................................. 29
2.13 Elliptic curve NAF-based scalar multiplication algorithm. ....................... 30
2.14 NAF_w computation algorithm. .............................................................. 30
2.15 Elliptic curve NAF_w-based scalar multiplication algorithm. ..................... 31

4.1 Generic measurement algorithm. ............................................................... 49

6.1 Basic block analysis algorithm. ............................................................... 62
6.2 Control flow graph analysis algorithm. .................................................... 64
6.3 Live variable analysis algorithm. ............................................................ 64
6.4 DAG analysis algorithm. ........................................................................ 67
6.5 Operator chain analysis algorithm. ........................................................... 68
6.6 Copy propagation algorithm. ..................................................................... 70
6.7 Dead code elimination algorithm. ............................................................. 71
6.8 Strength reduction algorithm. ................................................................. 73
6.9 Common sub-expression elimination algorithm. ........................................ 74
6.10 Naive implementation of the point doubling with \( z_R = 1 \). ..................... 78
6.11 Point doubling code after some initial simplifications. ............................. 78
6.12 Point doubling code after some DAG-based optimizations. ..................... 78
6.13 Optimal point doubling code. ................................................................. 79
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 Schlüssellängen [8] and the Commercial National Security Algorithm Suite [9], respectively for the commercial and the top secret uses.

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, but 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


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


Thesis organization

This dissertation is structured as follows.

Chapter 1 introduces some real-world scenarios where 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
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.

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>Google Search is launched</td>
</tr>
<tr>
<td>1998</td>
<td>Google Inc. is founded</td>
</tr>
<tr>
<td>2009</td>
<td>Initial encrypted search tests</td>
</tr>
<tr>
<td>2010</td>
<td>U. S. logged users by default use encryption</td>
</tr>
<tr>
<td>2011</td>
<td>Full encrypted search in place (except for ads)</td>
</tr>
<tr>
<td>2013</td>
<td>Security for everyone?</td>
</tr>
</tbody>
</table>

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 tool1. 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.

---

1cipherscan can be freely downloaded from https://github.com/jvehent/cipherscan.
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 it 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
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 disserta-
tion 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 auto-
matic 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 qasm and its related tools. It was initially presented
by Bernstein [57] in a talk at the Fast Software Encryption 2005 conference⁴. 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,

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/.

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.

<table>
<thead>
<tr>
<th>protocol</th>
<th>ECDH</th>
<th>ECDSA</th>
<th>ECMQV</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPsec</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>TLS</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>SSH</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

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.
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:


## 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 \((G, \circ)\) is a set \(G\) coupled with a binary operation \(\circ : G \times G \to G\), called the group operation, which satisfies the following properties:

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

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

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 \(G\) is the number of elements that contains and it is denoted by \(|G|\).

Definition (Finite group). A group \((G, \circ)\) is a finite group if \(|G| < \infty\).

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 \((\mathbb{F}, \circ, \cdot)\) is a set \(\mathbb{F}\) coupled with two binary operations \(\circ : \mathbb{F} \times \mathbb{F} \to \mathbb{F}\) and \(\cdot : \mathbb{F} \times \mathbb{F} \to \mathbb{F}\), called the field operations, such that:

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

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 $F$ is a finite field if $|F| < \infty$.

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 $F$, denoted by $\text{char}(F)$, is zero if the field is not finite, otherwise it is the smallest positive integer $n$ such that:

$$1 + \ldots + 1 = n \cdot 1 = 0.$$

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 $\text{char}(\mathbb{F}_2) = 2$.

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

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

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

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 $F'/F$, its dimension is called degree and it is denoted by $\deg(F'/F)$.

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 $\mathbb{F}_2[t]/t^8 + t^4 + t^3 + t + 1$.

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 $F$, a $F$-rational point is a point on an algebraic curve with all of its coordinates in $F$. 
Definition (Elliptic curve). An elliptic curve $E$ over a field $\mathbb{F}$ is denoted by $E/\mathbb{F}$. It is a smooth curve with at least one $\mathbb{F}$-rational point and defined by the Weierstraß equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$  \hfill (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 $\mathbb{R}$.

![Figure 2.1: Plots of two elliptic curves over $\mathbb{R}$](image)

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

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

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

The point $P_\infty$ 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 $E/\mathbb{F}$ an elliptic curve. Then, $(E(\mathbb{F}), +)$ 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:

\[
\begin{align*}
R = P + P_\infty & \implies \begin{cases} 
  x_R = x_P \\
  y_R = y_P
\end{cases} \\
R = -P & \implies \begin{cases} 
  x_R = x_Q \\
  y_R = -y_Q - a_1x_Q - a_3 \\
  \lambda = (3x_P^2 + 2a_2x_P + a_4 - a_1y_P)/(2y_P + a_1x_P + a_3)
\end{cases} \\
R = P + P & \implies \begin{cases} 
  x_R = \lambda^2 + a_1\lambda - a_2 - 2x_P \\
  y_R = \lambda(x_P - x_R) - y_P - a_1x_R - a_3 \\
  \lambda = (y_P - y_Q)/(x_P - x_Q)
\end{cases} \\
R = P + Q & \implies \begin{cases} 
  x_R = \lambda^2 + a_1\lambda - a_2 - x_P - x_Q \\
  y_R = \lambda(x_P - x_R) - y_P - a_1x_R - a_3.
\end{cases}
\end{align*}
\]
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 \( \mathcal{E}/\mathbb{F} \), 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 \( -R \).

![Graphical explanation of \( R = P + Q \).](image1)

![Graphical explanation of \( R = 2P \).](image2)

**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:

\[
n \cdot P = P + \ldots + P, \quad \forall P \in \mathcal{E}(\mathbb{F}) \land \forall n \in \mathbb{N}.
\]

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

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

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

Definition (Supersingular elliptic curve). Let $E/F_{2^n}$ 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 $E$ into a supersingular elliptic curve defined by the equation:

$$
y^2 + a_3y = x^3 + a'_2x + a'_6 \quad \text{with} \quad a_3 \neq 0.
$$

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 $E/F_{2^n}$ be an elliptic curve defined by the Equation 2.1 where $a_1 \neq 0$. The admissible change of variables $(x, y) \mapsto (a_1^2x + a_3y + (a_1^3a_4 + a_5^3)/a_1^2)$ transform the curve $E$ into a non-supersingular elliptic curve defined by the equation:

$$
y^2 + xy = x^3 + a'_2x^2 + a'_6 \quad \text{with} \quad a'_6 \neq 0.
$$

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

$$
\begin{align*}
R = P + P_\infty &\implies \begin{cases}
  x_R = x_P \\
  y_R = y_P
\end{cases} \\
R = -P &\implies \begin{cases}
  x_R = x_Q \\
  y_R = x_Q + y_Q
\end{cases} \\
R = P + P &\implies \begin{cases}
  x_R = \lambda^2 + \lambda + a'_2 \\
  y_R = \lambda(x_P + x_R) + x_R + y_P
\end{cases} \\
R = P + Q &\implies \begin{cases}
  x_R = \lambda^2 + \lambda + x_P + x_Q + a'_2 \\
  y_R = \lambda(x_P + x_R) + x_R + y_P.
\end{cases}
\end{align*}
$$

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 $\Sigma^*$ be the set of all the binary strings and $f$ a function such that $f: \Sigma^* \to \Sigma^*$. 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;
• ∀y ∈ Σ*, there is no PTime algorithm that return either x ∈ Σ* such that y = f(x) or ‘error’ if y ∉ 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 infeasible.

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 \( n \cdot P \) is easy, but retrieving \( n \) knowing \( P \) and \( Q \) is hard.

**Definition (ECDLP).** Let be \( E/F \) an elliptic curve and \( P,Q \in E(F) \) 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 \( n = \log_Q P \).

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 \( E/F \) is its number of points \( |E(F)| \). Since Equation 2.1 has at most two solutions for each \( x \in F \), it is easy to prove that:

\[
1 \leq |E(F)| \leq 2|F| + 1.
\]

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 \( l = \log_2(|E(F)|) \).

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 \( O(2^l) \). 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 \( O(2^{k/2}) \).

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 \( E/F_{2^k} \), some of the minimum security checks include:

• \( |E(F_{2^k})| \) should be large enough (at least \( 2^{160} \)) to avoid brute-force attacks;
• \( E \) 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];
• \( |E(F_{2^k})| \) 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 \( |E(F_{2^k})| \) 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 $K = n_A \cdot R_B$.

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 $E/F$ an elliptic curve, $G, A, B, K \in E(F)$ 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 $\sqrt{|E/F|}$ 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.

![Figure 2.4: Man-in-the-middle attack on the ECDH protocol.](image)

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 work on an elliptic curve $E$ 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 $\text{deg}(a(t)) < 2n$ into another polynomial $b(t) \in \mathbb{F}_{2^n}[t]/m(t)$ with $\text{deg}(b(t)) < n$.

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 $\text{deg}(r(t)) \leq \lfloor n/2 \rfloor$, 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 2.1: Binary field modular reduction algorithm.

```plaintext
algorithm Reduce(a field element a(t)) return a(t) mod m(t)
1    u ← a[0 → n - 1]
2    foreach exponent e_i of r(t) do
3        u ← u ⊕ (a >> e_i)
4    end
5    v ← u[0 → n - 1]
6    foreach exponent e_i of r(t) do
7        v ← v ⊕ (u >> e_i)
8    end
9    return v[0 → n - 1]
```

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 2.2: Binary field addition algorithm.

\[
\text{Algorithm 2.2: Binary field addition algorithm.}
\]

\[
\text{Add(two field elements } a(t) \text{ and } b(t)) \quad \text{return } a(t) + b(t)
\]

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 \(2^n\) bits, so that it must be further reduced.

The \textit{comb} method \cite{25}, also known as \textit{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 \(a(t) \cdot c(t), \forall c(t) \text{ with } \deg(c(t)) \leq w\) 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 2.3: Binary field comb multiplication algorithm.

\[
\text{Algorithm 2.3: Binary field comb multiplication algorithm.}
\]

When the window width equals to 1, this method becomes the so called \textit{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 Karatsuba-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:

\[
\begin{align*}
a(t) &= a_H(t) \cdot t^{[n/2]} + a_L(t) \\
b(t) &= b_H(t) \cdot t^{[n/2]} + b_L(t).
\end{align*}
\]

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

\[
\begin{align*}
a(t) \cdot b(t) &= (a_H(t) \cdot t^{[n/2]} + a_L(t)) \cdot (b_H(t) \cdot t^{[n/2]} + b_L(t)) = \\
&= a_H(t)b_H(t)\cdot t^n + \\
&+ ((a_H(t) + a_L(t))(b_H(t) + b_L(t)) + a_H(t) + b_H(t) + a_L(t) + b_L(t))t^{[n/2]} + \\
&+ a_L(t)b_L(t).
\end{align*}
\]

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** \text{Karatsuba}(two field elements \( a(t) \) and \( b(t) \)) \text{return} \( a(t) \cdot b(t) \)

\[
\begin{align*}
a_L &\leftarrow a[0 \rightarrow [n/2] - 1] \\
a_H &\leftarrow a[[n/2] \rightarrow n] \\
b_L &\leftarrow b[0 \rightarrow [n/2] - 1] \\
b_H &\leftarrow b[[n/2] \rightarrow n] \\
u &\leftarrow a_H \cdot b_H \\
v &\leftarrow a_L \cdot b_L \\
w &\leftarrow (a_H \oplus a_L) \cdot (b_H \oplus b_L) \\
w &\leftarrow w \oplus u \oplus v \\
z_1 &\leftarrow 0 \\
z_2 &\leftarrow [n \rightarrow [n/2] - 1 + n] - u \\
z_3 &\leftarrow 0 \\
z_3 &\leftarrow [0 \rightarrow [n/2] - 1] - v \\
z_4 &\leftarrow 0 \\
z_5 &\leftarrow [n/2] \rightarrow n - 1] - w \\
\text{return} &\ z_1 \oplus z_2 \oplus z_3
\end{align*}
\]

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 \times 64 bits multiplication can be fully performed via a special assembler operation (Section 6.3.3).
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 \((1, 1, 0, 1, 0)\) is \((1, 0, 1, 0, 0, 1, 0, 0)\). 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 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.
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 \( \text{fls}(\cdot) \) 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 \( a \), computing the last set is the same as calculating \( \lfloor \log_2(a) \rfloor \). This is also equivalent to compute the degree of the polynomial \( a(t) \), so that \( \text{fls}(a) \equiv \lfloor \log_2(a) \rfloor \equiv \deg(a(t)). \)

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 $a^{-1}(t)$

1. $u \leftarrow a$
2. $v \leftarrow m$
3. $g_1 \leftarrow 1$
4. $g_2 \leftarrow 0$
5. while true do
   6.     while $u[0] = 0$ do
       7.         $u \leftarrow u \gg 1$
       8.     if $g_1[0] = 0$ then
           9.         $g_1 \leftarrow g_1 \gg 1$
       10.    else
            11.        $g_1 \leftarrow (g_1 \oplus m) \gg 1$
       12.    end
   13.  end
   14. if $u = 1$ then
   15.     return $g_1$
   16.  end
   17. if \text{fls}(u) < \text{fls}(v) then
   18.     $u \leftrightarrow v$
   19.     $g_1 \leftrightarrow g_2$
   20. end
   21. $u \leftarrow u \oplus v$
   22. $g_1 \leftarrow g_1 \oplus g_2$
23. 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 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 $z = 0$;

```plaintext
algorithm DCEA(a field elements $a(t)$) return $a^{-1}(t)$
1 $u \leftarrow a$
2 $v \leftarrow m$
3 $g_1 \leftarrow 1$
4 $g_2 \leftarrow 0$
5 $direction \leftarrow \text{backward}$
6 while true do
7     if $direction = \text{backward}$ then
8         if $u[0] = 0$ and $g_1[0] = 1$ then
9             $g_1 \leftarrow g_1 \oplus m$
10        end
11        $direction \leftarrow \text{forward}$
12     else
13         if $u[0] = 0$ then
14             $u \leftarrow u \gg 1$
15             $g_1 \leftarrow g_1 \gg 1$
16        end
17         if $	ext{fils}(u) < \text{fils}(v)$ then
18             $u \leftarrow v$
19             $g_1 \leftarrow g_2$
20         end
21         $u \leftarrow u \oplus v$
22         $g_1 \leftarrow g_1 \oplus g_2$
23     end
24     end
25     if $u = 1$ then
26         return $g_1$
27     end
29 end
```
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:

\[
n = n_02^0 + n_12^1 + n_22^2 + n_32^3 + \ldots = n_0 + 2(n_1 + 2(n_2 + 2(n_3 + \ldots))).
\]
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 deteriorates 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.

\begin{definition}
(Signed-digit representation). The signed-digit representation of an integer \( n \) in radix \( 2 \).
\end{definition}
Table 2.2: Elliptic curve costs for various coordinate systems.

<table>
<thead>
<tr>
<th>system</th>
<th>addition</th>
<th>doubling</th>
<th>negation</th>
</tr>
</thead>
<tbody>
<tr>
<td>affine</td>
<td>$1I + 2M + 1S$</td>
<td>$1I + 2M + 1S$</td>
<td>$0M$</td>
</tr>
<tr>
<td>projective</td>
<td>$14M + 1S$</td>
<td>$7M + 3S$</td>
<td>$0M$</td>
</tr>
<tr>
<td>Jacobian</td>
<td>$14M + 5S$</td>
<td>$4M + 5S$</td>
<td>$1M$</td>
</tr>
<tr>
<td>López-Dahab</td>
<td>$13M + 4S$</td>
<td>$3M + 5S$</td>
<td>$1M$</td>
</tr>
</tbody>
</table>

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

Algorithm 2.12: NAF computation algorithm.

Algorithm 2.13 shows the add-and-double procedure modified to use the NAF representation.
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 $(n_{l-1}, \ldots, n_0)_{NAF_w}$.

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

**Algorithm 2.14:** NAF$_w$ computation algorithm.

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

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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average cost with an (l) bit length integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddAndDouble</td>
<td>(\frac{l}{2}A + lD)</td>
</tr>
<tr>
<td>NAFMultiply</td>
<td>(\frac{l}{3}A + lD)</td>
</tr>
<tr>
<td>NAFwMultiply</td>
<td>((2^{w-2} + \frac{l}{w+1} - 1)A + (l+1)D)</td>
</tr>
</tbody>
</table>

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

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.

Figure 2.5: Algorithm choices for the scalar multiplication implementation.
Chapter 3  

Implicit computational complexity

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);
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 ∆) range over list of formulas.

The notation Γ ⊢ ∆ 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 ∆ hold. The parameters of the sequent operator ⊢ are known as contexts (Γ and ∆ 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, Γ ⊢ B means that B is true and A₁ ∧ A₂ ⊢ B means that A₁ ∧ A₂ ⇒ B.

3.1.1 Classical logic

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

\[ s ::= \text{"0"} \mid \text{"1"} \mid \text{s "\&" s} \mid \text{s "\|" s} \mid \text{s "⇒" s}. \]

Where ∧, ∨ and ⇒ are the usual logical and, or and implication operators. The negation ¬X does not appear in the list since it can be defined as X ⇒ 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 \( \exists x : x \mod 2 = 0 \) 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:

- $X \lor Y$ 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;
- $X \land Y$ 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, $X \lor (\neg X)$ 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:

$$s ::= "0" \mid "1" \mid "\top" \mid "\bot" \mid s "\otimes" s \mid s "\odot" s \mid s "\&" s \mid s "\&" s \mid s "\oplus" s \mid s "\odot" s \mid "{!}\" s \mid "{?}\" s.$$

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 $X \odot Y$ 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 $X \rightarrow Y$ 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 $\otimes$, $\odot$, $\&$ and $\odot$.

The LL allows to express several intuitionistic rules. For instance, $X \Rightarrow Y$ becomes $!X \Rightarrow Y$, 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 \(\lambda\)-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 \(\lambda\)-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 \(\lambda\)-calculus expression is known as a \(\lambda\)-term. There exists three kind of \(\lambda\)-terms:

- \(\text{variables}\);
- \(\lambda\)-abstractions denoted as \(\lambda x. Y\), where \(x\) is a variable and \(Y\) a \(\lambda\)-term;
- \(\lambda\)-applications denoted by \(XY\), where \(X\) and \(Y\) are \(\lambda\)-terms.

A \(\lambda\)-abstraction models an anonymous function with a single input parameter. For instance, \(\lambda x. (2x)\) is the \(\lambda\)-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 \(\lambda\)-application instead represents the instantiation of a \(\lambda\)-abstraction with a particular input. For instance, \((\lambda x. (2x)) 3\) = 6 since it means \(f(3) = 2 \cdot 3 = 6\).

The \(\lambda\)-calculus might be typed or untyped. In a *typed \(\lambda\)-calculus*, each \(\lambda\)-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 \(\lambda\)-Calculus (STLC), denoted by the symbol \(\lambda \to\). More complex typed systems can be created by extending it.

If \(\tau\) indicates any basic type, then the following BNF rule can be used to describe the types of a \(\lambda\)^\("\)-term:

\[
t :: \tau | t \Rightarrow t.
\]

If \(c\) indicate any constant and \(x\) any variable, a \(\lambda\)^\("\)-term can be defined by the rule:

\[
s :: c | x | s s | \"\lambda\" x \:" t \:" | s.
\]

With respect to the untyped \(\lambda\)-calculus, the \(\lambda\)-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:

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \frac{\Gamma \vdash X : (T_1 \Rightarrow T_2) \quad \Gamma \vdash Y : T_1}{\Gamma \vdash XY : T_2} \quad \frac{\Gamma, x : T_1 \vdash Y : T_2}{\Gamma \vdash (\lambda x : T_1) Y : (T_1 \Rightarrow T_2)}
\]

In other words:
the rule \textit{var} states that if we know that the variable \(x\) has type \(T\) in the context \(\Gamma\), then we deduce that \(x\) has always type \(T\);

- the rule \textit{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 \textit{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 \(\lambda^*\) program can be demonstrated, then the program is type-correct. For instance, let suppose to have a \(\lambda^*\) program where:

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

Let consider the program:

\(((\lambda x : \textit{int}(\lambda y : \textit{bool}.x))3)\textit{false}.)\]

By performing the substitution in \(x\) and then in \(y\), it becomes \((\lambda y : \textit{bool}.3)\textit{false}\) and finally only 3. The result type is obviously an \textit{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 \(\vdash((\lambda x : \textit{int}(\lambda y : \textit{bool}.x))3)\textit{false} : \textit{int}\).

This can be accomplished via the following proof tree:

\[
\begin{array}{c}
\text{var} \quad \vdash x : \textit{int} & & \vdash y : \textit{bool} \\
\text{abs} \quad x : \textit{int}, y : \textit{bool} \vdash x : \textit{int} \\
\text{abs} \quad \vdash \lambda x : \textit{int}(\lambda y : \textit{bool}.x) : \textit{bool} \Rightarrow \textit{int} \\
\text{app} \quad \vdash (\lambda x : \textit{int}(\lambda y : \textit{bool}.x))3 : \textit{bool} \Rightarrow \textit{int} & & \vdash 3 : \textit{int} \\
\text{app} \quad \vdash (\lambda x : \textit{int}(\lambda y : \textit{bool}.x))3)\textit{false} : \textit{bool} \\
\end{array}
\]

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 \(\lambda\)-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 \(\lambda\)-calculi and some logic families. For instance, the \(\lambda^*\) corresponds to a fragment of IL known as \textit{minimal logic}.

\(^1\)In this context, a \textit{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:

\[ t ::= \tau \mid t \to t \mid t \Rightarrow t \mid \forall \tau t \mid \& t. \]

Where \( \tau \) represents any basic type. The linear implication \( \to \) has the same semantics 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 represent 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 \( f : \mathbb{F}_2^n \to \mathbb{F}_2^n \).

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:

\[
\text{False} = 0 = \lambda x.\lambda y.x \\
\text{True} = 1 = \lambda x.\lambda y.y
\]

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:

\[
\emptyset = \lambda f. \lambda x. x
\]

\[
(a_0) = \lambda f. \lambda x. (f \ a_0 \ x)
\]

\[
(a_1, a_0) = \lambda f. \lambda x. (f \ a_1 (f \ a_0 \ x))
\]

\[
\vdots
\]

\[
(a_{n-1}, \ldots, a_0) = \lambda f. \lambda x. (f \ a_{n-1} (f \ a_{n-2} (f \ a_{n-3} (\ldots \ x))))
\]

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

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

\[
\lambda f. \lambda x. (f \ 1 \ (f \ 0 \ (f \ 1 \ x))).
\]

In this context, each element of a Church word (a bit) can have three values: 0, 1 and \(\bot\). The bottom value \(\bot\) 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 \((a_{n-1}, \ldots, 0)\) will be used to denote a set of threaded words, that is a bit matrix where the columns contains the values specified in the tuples \(a_{n-1}, \ldots, 0\).

### 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 \(\lambda\)-term \textsc{Xor} defined as:

\[
\text{Not} = \lambda x. \lambda y. \lambda z. (x \ z \ y)
\]

\[
\text{Xor} = \lambda x. \lambda y. (x (\text{Not} \ y) \ y).
\]

For example:

\[
\text{Xor} \ 0 \ 1 = (\lambda x. \lambda y. (x (\text{Not} \ y) \ y)) \ 0 \ 1 =
\]

\[
= 0 \ (\text{Not} \ 1) \ 1 =
\]

\[
= 0 \ ((\lambda x. \lambda y. \lambda z. (x \ z \ y)) \ 1) \ 1 =
\]

\[
= 0 \ ((\lambda y. \lambda z. (1 \ z \ y)) \ 1) \ 1 =
\]

\[
= 0 \ ((\lambda y. \lambda z. y) \ 1) \ 1 =
\]

\[
= 0 \ 1 \ 1 =
\]

\[
= 1.
\]
In $\lambda$-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:

\[
\text{Head} = \lambda x. (x \text{ True}) \quad \text{retrieves the head of the tuple } x
\]
\[
\text{Tail} = \lambda x. (x \text{ False}) \quad \text{retrieves the tail of the list } x.
\]

The bitwise exclusive or can be obtained by means of the $\text{Map}[F]$ 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 the $\text{wRev}$ 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.

\[
\begin{pmatrix}
0 & 0 & 1 \\
1 & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1
\end{pmatrix}
\quad\quad
\begin{pmatrix}
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 0
\end{pmatrix}
\quad\quad
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 1 \\
0 & 1 & 1 \\
0 & 1 & 1
\end{pmatrix}
\quad\quad
\begin{pmatrix}
1 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1 \\
1 & 0 & 0
\end{pmatrix}
\]

(a) Initial setup. (b) Extended matrix. (c) Reversed matrix. (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 $\lambda$-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$^{\text{st}}$. 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$^{\text{nd}}$. the matrix is reversed vertically and the last row is deleted (see Figure 3.1c);

3$^{\text{rd}}$. 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 \( \lambda \)-term of BEA in TFA can be written as:

\[
wInv = \lambda u. \{ wProj(d \\
wRevInit(BkwVst(wRev(FwdVst tw)))) \\
(\text{MapThread}[\lambda u. \lambda v. \lambda g_1. \lambda g_2. \{u, v, g_1, g_2, m, \text{stop, } b_0, b_1, v', g_2', m'\}]
\]
\[
\left(\begin{array}{c}
0, \ldots, 0, 1 \\
0, \ldots, 0 \\
m, m \\
0, \ldots, 0 \\
\vdots
\end{array}\right)
\]

(3.1)

The \( \lambda \)-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:

\[
\lambda f. \lambda x. (f (u_0, m_0, \ 1, \ 0, \ m_0, \ 0, \ 1, \ 1, \ 0, \ 0, \ 0)) \\
(f (u_1, m_1, \ 0, \ 0, \ m_1, \ 0, \ 1, \ 1, \ 0, \ 0, \ 0)) \\
(f (u_2, m_2, \ 0, \ 0, \ m_2, \ 0, \ 1, \ 1, \ 0, \ 0, \ 0)) \\
(f (u_3, m_3, \ 0, \ 0, \ m_3, \ 0, \ 1, \ 1, \ 0, \ 0, \ 0)) \\
\vdots
\]

This matrix is represented by the set of threaded words:

\[\langle u, v, g_1, g_2, m, \text{stop, } b_0, b_1, v', g_2', m'\rangle.\]

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;
- \( \text{stop} \), 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 \( u[0] \equiv 0 \) and \( g_1[0] \equiv 0 \);
• \(v', g'_2\) 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\(^1\). if \(stop[0] = 1\) then \(FwdVst\) does nothing since the algorithm has already computed the inverse;
2\(^1\). if \(u[0] = 0 \land g_1[0] = 0\), the step \(1^\infty\) is performed in order to avoid changes to the columns \(u\), \(g_1\) and \(m\);
3\(^1\). if \(u[0] = 0 \land g_1[0] = 1\), the column \(m\) is added to the column \(g_1\) (via an exclusive or) and the step \(1^\infty\) is executed as before;
4\(^1\). 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\(^1\). if \(stop[0] = 1\) then \(BkwVst\) does nothing since the algorithm has already computed the inverse;
2\(^1\). if the steps 2\(^1\) or 3\(^1\) 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 \(2^\infty\);
3\(^1\). if \(u \neq 1 \land u[0] = 1\), 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 \( w_{\text{RevInit}} \) and another pass might begin.
At the end of all the \( d^2 \) iterations, the \( w_{\text{Proj}} \) function is used to extract the column \( g_1 \) from the threaded words, since it will contain the multiplicative inverse.

Note that writing \( w_{\text{Inv}} \) 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 \( w_{\text{Inv}} \) 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 \( \lambda \)-term \( w_{\text{Inv}} \) (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.

![Figure 3.2: BEA and DCEA in comparison.](image)

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 \( \text{direction} \) variable is used to emulate the forward and backward visits in \( w_{\text{Inv}} \). 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.

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
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 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 passes 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 if 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.

![Architecture of the C back-end.](image)

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 at 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
1  T ← ∅
2  for i ← 1 to p do
3    start ← current time
4    for s times do
5      call f with some random parameters
6    end
7    end ← current time
8    T[i] ← (end - start)/p
9  end
10 q1 ← compute the lower quartile of T
11 q3 ← compute the upper quartile of T
12 foreach t ∈ T do
13  if t ∉ [q1 - θ(q3 - q1), q3 + θ(q3 - q1)] then
14    remove t from T
15  end
16 end
17 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 $0 ≤ θ ≤ 1$ (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 thought 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.
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 $x ← y + z + w$ can be rewritten as $t ← y + z$ and $x ← t + w$.

Jumps and branches can only be expressed by gotos.

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

$$\text{if } x \leq 20 \text{ then } y ← 1 \text{ else } y ← −1 \text{ can be simplified into } y ← 1.$$  

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 metadata 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.

\(^1\)Given two unsigned integers \(x\) and \(y\), the *round-up division* computes the value \(\lceil x/y \rceil\), while the *integer division* returns \([x/y]\).
### Syntax and Names

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

#### 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 constraints imposed by **HRL** by removing the three-address code limitation, so that complex statements can be easier written, such as \( x \leftarrow (y \text{ or } (0, 1, 1)) \ll (1 + 2) \). 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 \( x \leftarrow \text{log}(y) \). 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 com-
 pact math-like representation without explicitly using a loop. For instance, \((2 \times i \mid i \in [0, 99])\)
creates the tuple containing the first 100 even numbers since it has the same meaning as the
mathematical expression \( (2 \times i : \forall i \in [0, 99]) \).

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

Some (a few, actually) programming languages perform a preprocessing phase before the actual
compilation, modifying the source code accordingly to some transformation rules. aXiom imple-
ments 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
\( 4 \times M + 1 \) 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 preproces-
sor 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.

```
1 static SIZE in ro uint8 <- 10 * 7 + 9
2 function f(x in rw SIZE of bit)
3 {
4     i in ro uint
5     static if SIZE mod 2 = 0
6         for i in [1, 3]
7             x <- x << i
8     } else
9         for i in [2, 4]
10            x <- x >> i
11 }
12 }
(a) Snippet in aXiom.
```

```
function f
2 x in rw 79 of bit
3 {
4     i in ro uint
5     i <- 2
6     i <- 3
7     i <- 4
8 }
9 x <- x >> i
10 x <- x >> i
11 }
12 }
(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.

```
1 u <- a[0 → n - 1]
2 foreach exponent e_i of r(t) do
3     u <- u ∘ (a >> n) ∘ e_i
4 end
5 v <- u[0 → n - 1]
6 foreach exponent e_i of r(t) do
7     v <- v ∘ (u >> n) ∘ e_i
8 end
9 return v[0 → n - 1]
(a) Reduction pseudo-code.
```

```
1 u <- a and lowones(N, 2 * N)
2 static for e in E
3     u <- u xor ((a >> N) << e)
4
5 v <- u and lowones(N, 2 * N)
6 static for e in E
7     v <- v xor ((u >> N) << e)
8
9 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 \texttt{test} function annotation. This annotation accepts a single parameter that can be \texttt{success} or \texttt{failure} if the function is expected respectively to succeed or to fail. A test function must declare exactly one write-only \texttt{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.

```aXiom
1 export
2 function f(out in wo int, in1 in ro int, in2 in ro int)
3 {
4   out <- in1 / in2 + in1 \text{mod} in2
5 }
6
7 :test(success)
8 function test1(result in wo logical)
9 {
10   x in rw int
11   f(x, 10, 4)
12   result <- x = 4
13 }
14
15 :test(failure)
16 function test2(result in wo logical)
17 {
18   x1 in rw int
19   x2 in rw int
20   f(x1, 10, 4)
21   f(x2, 4, 10)
22   result <- x1 = x2
23 }
```

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 \texttt{main()} function calling all the declared tests and printing the results on the standard output via a number of \texttt{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.
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:


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 blocks**, 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.

```plaintext
algorithm ComputeBasicBlocks(a function F) return the basic blocks of F
1  B ← ( <<ENTRY>> )
2  foreach instruction i ∈ F do
3      if i is the first instruction of F or
4          i has a label or
5              the instruction preceding i is a jump then
6          B ← new empty basic block
7          B ← B ∪ ( B )
8      end
9  B ← B ∪ ( i )
10 end
11 B ← B ∪ ( <<EXIT>> )
12 return 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 { <<ENTRY>>, <<EXIT>> }.

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.

![Figure 6.1: Example of a control flow graph.](image)

The notations $\text{succ}[B]$ and $\text{prec}[B]$ 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 $\text{controlFlowGraphAnalysis}$ 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 $\text{use}[B]$ 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, $\text{def}[B]$ 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 $\leftarrow$ 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 $\text{out}[B]$, 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 $\text{liveVariableAnalysis}$ annotation. This pass automatically activates the control flow graph analysis, if needed.
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.
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 $x \leftarrow y + 1$ and $w \leftarrow y + 1$ are merged together in the graph since the analysis detected that they compute the same value.

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 $x \leftarrow y[i]$ and $v \leftarrow y[i]$ that can be apparently merged during the DAG construction. However, if $i = j$ this cannot be done since the instruction $y[j] \leftarrow u$ 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 $x \leftarrow y[z]$ is represented by means of a node with two children $y$ and $z$ and labeled with $x$;
- representing a copy to tuple instruction $x[y] \leftarrow z$ 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.
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 \( x = ((c \cdot d) \cdot b) \cdot (b \cdot v) \).

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 \( \text{operatorChainAnalysis} \) 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 6.4: DAG analysis algorithm.

```plaintext
def ComputeDAG(B):
    V[::] <- ∅, O[::] <- ∅, N <- ∅
    foreach instruction i ∈ B do
        out <- the output operand of i or NIL if does not exist
        in1 <- the first input operand of i or NIL if does not exist
        in2 <- the second input operand of i or NIL if does not exist
        if in1 ≠ NIL then
            n1 <- V[in1]
            if n1 = NIL then
                n1 <- new node labeled with in1 and add it to N
                V[in1] <- n1
            end
        end
        if in2 ≠ NIL then
            n2 <- V[in2]
            if n2 = NIL then
                n2 <- new node labeled with in2 and add it to N
                V[in2] <- n2
            end
        end
        if i is a copy to tuple then
            n0 <- V[out]
            if n0 = NIL then
                n0 <- new node labeled with out and add it to N
            end
        else
            n0 <- NIL
        end
        n <- O[\{ t, n0, n1, n2 \}]
        if n = NIL then
            n <- new node with the type of i
            if i is a copy to tuple then
                set n0, n1 and n2 as children of n
                V[out] <- NIL
            else
                set out as the label of n and n1, n2 as children of n
                V[out] <- n
            end
            O[\{ t, n0, n1, n2 \}] <- n and add it to N
        else
            add out to the labels of n
        end
    end
    return N
```

67
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.

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;
By means of this optimization technique, for instance, some identities that CryptoGen can do.

Table 6.1: CryptoGen algebraic simplification rules.

<table>
<thead>
<tr>
<th>original code</th>
<th>constraints</th>
<th>transformed code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow y /= z$</td>
<td>$y, z$ are relative and $\max(y) &lt; \min(z)$</td>
<td>$x \leftarrow \text{true}$</td>
</tr>
<tr>
<td>$x \leftarrow y /= z$</td>
<td>$y, z$ are relative and $\max(y) &gt; \min(z)$</td>
<td>$x \leftarrow \text{true}$</td>
</tr>
<tr>
<td>$x \leftarrow y &lt; z$</td>
<td>$y, z$ are relative and $\min(y) &gt; \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>$x \leftarrow y &lt; z$</td>
<td>$y, z$ are relative and $\min(y) &lt; \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>$x \leftarrow y &lt;= z$</td>
<td>$y, z$ are relative and $\min(y) \geq \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>$x \leftarrow y &lt;= z$</td>
<td>$y, z$ are relative and $\min(y) &lt; \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>$x \leftarrow y &gt; z$</td>
<td>$y, z$ are relative and $\min(y) &gt; \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>$x \leftarrow y &gt; z$</td>
<td>$y, z$ are relative and $\min(y) \leq \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>$x \leftarrow y &gt;= z$</td>
<td>$y, z$ are relative and $\min(y) \geq \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>$x \leftarrow y &gt;= z$</td>
<td>$y, z$ are relative and $\min(y) &lt; \max(z)$</td>
<td>$x \leftarrow \text{false}$</td>
</tr>
<tr>
<td>call f, push x, push y, push z</td>
<td>:hasIdentityElement($y$) holds for f</td>
<td>$x \leftarrow z$</td>
</tr>
<tr>
<td>call f, push x, push y, push z</td>
<td>:hasIdentityElement($z$) holds for f</td>
<td>$x \leftarrow y$</td>
</tr>
<tr>
<td>call f, push x, push y, push z</td>
<td>:hasAbsorbingElement($y$) holds for f</td>
<td>$x \leftarrow a$</td>
</tr>
<tr>
<td>call f, push x, push y, push z</td>
<td>:hasAbsorbingElement($z$) holds for f</td>
<td>$x \leftarrow a$</td>
</tr>
<tr>
<td>call f, push x, push y, push z</td>
<td>:hasSpecialCase1($y, w$) holds for f</td>
<td>$x \leftarrow w$</td>
</tr>
<tr>
<td>call f, push x, push y, push z</td>
<td>:hasSpecialCase2($z, w$) holds for f</td>
<td>$x \leftarrow w$</td>
</tr>
</tbody>
</table>

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.

\[
\begin{align*}
1 & \quad x \leftarrow (1, 2, 3) \\
2 & \quad y \leftarrow x \leftarrow 1 \\
3 & \quad x \leftarrow z \\
4 & \quad w \leftarrow x \leftarrow z
\end{align*}
\]

(a) Original code.

\[
\begin{align*}
1 & \quad x \leftarrow (1, 2, 3) \\
2 & \quad y \leftarrow (1, 2, 3) \leftarrow 1 \\
3 & \quad x \leftarrow z \\
4 & \quad w \leftarrow z
\end{align*}
\]

(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.

\begin{algorithm}
\begin{algorithmic}
1 \text{V}[\cdot] \leftarrow \emptyset \\
2 \text{foreach} \quad \text{instruction} \quad i \in B \quad \text{do} \\
3 \quad \text{out} \leftarrow \text{the output operand of } i \text{ or NIL if does not exist} \\
4 \quad \text{in}_1 \leftarrow \text{the first input operand of } i \text{ or NIL if does not exist} \\
5 \quad \text{in}_2 \leftarrow \text{the second input operand of } i \text{ or NIL if does not exist} \\
6 \quad \text{if } \text{in}_1 \text{ is a variable and } \text{V}[\text{in}_1] \neq \text{NIL} \text{ then} \\
7 \quad \quad \text{replace } \text{in}_1 \text{ with } \text{V}[\text{in}_1] \text{ in } i \\
8 \quad \text{end} \\
9 \quad \text{if } \text{in}_2 \text{ is a variable and } \text{V}[\text{in}_2] \neq \text{NIL} \text{ then} \\
10 \quad \quad \text{replace } \text{in}_2 \text{ with } \text{V}[\text{in}_2] \text{ in } i \\
11 \quad \text{end} \\
12 \quad \text{if } i \text{ is a copy then} \\
13 \quad \quad \text{V}[\text{out}] \leftarrow \text{in}_1 \\
14 \quad \text{end} \\
15 \end{algorithmic}
\end{algorithm}

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 \texttt{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
1   $D \leftarrow \text{ComputeDAG}(B)$
2   $\left( \{ \text{IN}[B] \} \cup \{ \text{OUT}[B] \} \right) \leftarrow \text{ComputeLiveliness}$\text{(the function containing } B)\text{)}
3   \text{foreach root node } n \in D \text{ do}
4       remove all the variables not in OUT\text{[}B\text{]} attached to n
5       \text{if } n \text{ has no variable attached then}
6           $D \leftarrow D \setminus \{ n \}$
7       \text{end}
8   \text{end}
9   $B \leftarrow \text{reassemble } D$
10  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.

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

(b) Transformed code.
```

```
(a) Original code.
```

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 $x = 2 \ast y$; 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(\textsf{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.

![Figure 6.10: Example of hidden strength reduction possibilities.](image)

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 \( v = c^2 \cdot (d^2 \cdot (a + b)) \).

![Figure 6.11: Strength reduction example.](image)

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

```
algorithm REDUCESTRENGTHS(a basic block B) return the transformed basic block B
1 D ← COMPUTEDAG(B)
2 (IN[B]) B\pi \cdot (OUT[B]) B\pi ← COMPUTELIVELINESS(the function containing B)
3 foreach root node n ∈ D do
   4 N ← (n)
   5 while N ≠ ∅ do
      6 N ← N[0]
      7 N ← N \ N[0]
      8 N ← the children of n ∪ N
      9 if n is related to a binary operator that is commutative and associative then
         10 C ← COMPUTECHAIN(n)
         11 remove all the non-frontier nodes from C
         12 remove all the nodes with more than 1 parent from C
         13 remove all the nodes with at least one variable in OUT[B] from C
         14 repeat
            15 pick two nodes n₁ ≠ n₂ with a common children c
            16 if n₁ is a successor of n₂ then
               17 d ← the children of n₂ different from c
               18 set d as the child of n₂ equal to c
               19 set c as the child of n₂ equal to d
               20 else
                  21 d ← the children of n₁ different from c
                  22 set c as the child of n₁ equal to d
                  23 set d as the child of n₁ equal to c
               24 end
            25 until any change in C occurs
         16 end
      17 end
   18 foreach node n ∈ D do
      19 if n has the same two children and its type has a specialization G then
         20 change the type of n to G and remove one of its children
      21 end
   22 end
   23 B ← reassemble D
   24 return B
```
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
1 D ← ComputeDAG(B)
2 C[·] ← ∅
3 foreach node n ∈ D do
4     C[n] ← ComputeChain(n)
5 end
6 foreach node n ∈ D do
7     if the type of n is associative and commutative then
8         C1 ← C[n]
9         foreach chain C2 ∈ C[·] such that C1 ≠ C2 and |C1| = |C2| do
10            if C1 and C2 have the same type and the same leaf nodes in any order then
11                merge n and the root of C2
12            end
13        end
14    end
15 end
16 B ← reassemble D
17 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 \( w = a + c + b \).

![DAG](image1)

(a) Source code in HRL.

![DAG](image2)

(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).

```
1 uint8_t x[79];
2 uint8_t y[79];
3 int i;
4
5 for (i = 78; i > 0; --i)
6   x[i] = y[i - 1];
7 x[0] = 0;
(a) Code without bit vector packing.
```

```
1 uint8_t x[10];
2 uint8_t y[10];
3 int i;
4
5 for (i = 9; i > 0; --i)
6   x[i] = (y[i] << 1) ^
7      (y[i - 1] >> 7);
8 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

75
time slow down due to an increased register spilling and filling\footnote{The \textit{register spilling} is the operation of moving a variable from a register to the memory. The opposite procedure is known as \textit{register filling}.}.

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 \texttt{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.

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
for (i = 0; i < 5; ++i)
  x[i] = y[i] & z[i];

(b) Code with loop unrolling.
\end{verbatim}

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 \texttt{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 \texttt{__builtin_clzl()} and the \texttt{__builtin_ia32 pclmulqdq128()} functions.

The \texttt{__builtin_clzl()} is used to compute the number of leading zero bits in a \texttt{long} variable. This instruction is automatically used when translating the \texttt{findlastset} HR construct. GCC can
implement this built-in differently w.r.t. the hardware platform, for instance:

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

The \texttt{\_builtin\_ia32\_pclmulqdq128}() is available only on the AMD64 processors and it implements the \texttt{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 \( \mathbb{F}_{2^64} \times \mathbb{F}_{2^64} \rightarrow \mathbb{F}_{2^{128}} \). In order to be effectively used by CryptoGen, a function computing the field multiplication must be annotated with the \texttt{:carryLessMultiplication} annotation. When performing the translation of the annotated function, CryptoGen will replace its body with a call to the \texttt{\_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, \texttt{:carryLessSquaring}, is available for the field squaring function.

This optimization is enabled when the \texttt{: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:

\[
\begin{aligned}
a &= a_6 z_R^6 \\
z_R &= z_R^2 x_R^2 \\
x_R &= x_R^4 + a \\
b &= y_R^2 + a_2 z_R + a \\
y_R &= b x_R + z_R a.
\end{aligned}
\]

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 \( 14M + 0S \).

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 \( 8M + 0S \).

The strength reduction and common sub-expression elimination passes further decreases the code cost to \( 4M + 2S \), 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 \( 3M + 3S \). This is the same optimal formula known in literature [113].
Algorithm 6.10: Naive implementation of the point doubling with $z_R = 1$.

1. $t_0 \leftarrow (0,0,0,0,0,0,0,0,0,0,0,0,0,0,1) \cdot a_6$
2. $t_1 \leftarrow t_0 \cdot (0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)$
3. $t_2 \leftarrow t_1 \cdot (0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)$
4. $a \leftarrow t_2 \cdot (0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)$
5. $t_4 \leftarrow (0,0,0,0,0,0,0,0,0,0,0,0,0,0,1) \cdot (0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)$
6. $t_4 \leftarrow t_3 \cdot x_P$
7. $z_R \leftarrow t_4 \cdot x_P$
8. $t_5 \leftarrow x_P \cdot x_P$
9. $t_6 \leftarrow t_5 \cdot x_P$
10. $t_7 \leftarrow t_6 \cdot x_P$
11. $x_R \leftarrow t_7 + a$
12. $t_8 \leftarrow y_P \cdot y_P$
13. $t_9 \leftarrow a_2 \cdot z_R$
14. $t_{10} \leftarrow t_8 + t_9$
15. $b \leftarrow t_{10} + a$
16. $t_{11} \leftarrow b \cdot x_R$
17. $t_{12} \leftarrow z_R \cdot a$
18. $y_R \leftarrow t_{11} + t_{12}$

Algorithm 6.11: Point doubling code after some initial simplifications.

1. $z_R \leftarrow x_P \cdot x_P$
2. $t_5 \leftarrow x_P \cdot x_P$
3. $t_6 \leftarrow t_5 \cdot x_P$
4. $t_7 \leftarrow t_6 \cdot x_P$
5. $x_R \leftarrow t_7 + a_6$
6. $t_8 \leftarrow y_P \cdot y_P$
7. $t_9 \leftarrow a_2 \cdot z_R$
8. $t_{10} \leftarrow t_8 + t_9$
9. $b \leftarrow t_{10} + a_6$
10. $t_{11} \leftarrow b \cdot x_R$
11. $t_{12} \leftarrow z_R \cdot a_6$
12. $y_R \leftarrow t_{11} + t_{12}$

Algorithm 6.12: Point doubling code after some DAG-based optimizations.
Algorithm 6.13: Optimal point doubling code.

\begin{verbatim}
1 \( z_R \leftarrow x_P^2 \)
2 \( t_7 \leftarrow z_R^2 \)
3 \( x_R \leftarrow t_7 + a_6 \)
4 \( t_8 \leftarrow y_P^2 \)
5 \( t_9 \leftarrow a_2 \cdot z_R \)
6 \( t_{10} \leftarrow t_8 + t_9 \)
7 \( b \leftarrow t_{10} + a_6 \)
8 \( t_{11} \leftarrow b \cdot x_R \)
9 \( t_{12} \leftarrow z_R \cdot a_6 \)
10 \( y_R \leftarrow t_{11} + t_{12} \)
\end{verbatim}
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.
Table 7.1: Specifications of the tested hardware platforms.

<table>
<thead>
<tr>
<th>curve name</th>
<th>field</th>
<th>$a_6^i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-163</td>
<td>$t^{163} + t^7 + t^6 + t^3 + 1$</td>
<td>20a60190786c953ca1481eb10512f78744a3205fd16</td>
</tr>
<tr>
<td>B-233</td>
<td>$t^{233} + t^{74} + 1$</td>
<td>066647ed6332c7f8e0923b658213633b20e90e4281f7e15f7d8f90ad6</td>
</tr>
<tr>
<td>B-283</td>
<td>$t^{283} + t^{12} + t^7 + t^5 + 1$</td>
<td>27b60a6e88859d5a0a4a08a09a03f0ac9f7d645309fa2a5</td>
</tr>
<tr>
<td>B-409</td>
<td>$t^{409} + t^{87} + 1$</td>
<td>021a5e28c949e9eb5e4b9a753b7b4767fd6422ef11f3dd67476</td>
</tr>
<tr>
<td>B-571</td>
<td>$t^{571} + t^{10} + t^5 + t^2 + 1$</td>
<td>545f16</td>
</tr>
</tbody>
</table>

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;
Experimental results

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

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.

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.

---

3In this case the low level 64bit × 64bit WINDOWMULTIPLY function implements the COMB algorithm with a window width of 4bit if the target idiom usage is not enabled, otherwise it is translated as a single __builtin_ia32_pclmulqdq128() function call.
### 7.2.1 Results on the i7 processor

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

<table>
<thead>
<tr>
<th>operation</th>
<th>time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B-163</td>
</tr>
<tr>
<td>$a \mod b$</td>
<td>$R_8$</td>
</tr>
<tr>
<td>$a + b$</td>
<td>$A_1$</td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td>$M_7$</td>
</tr>
<tr>
<td>$a^2$</td>
<td>$S_5$</td>
</tr>
<tr>
<td>$a^{-1}$</td>
<td>$B_{1254}$</td>
</tr>
<tr>
<td>$P + Q$</td>
<td>LD_{238}</td>
</tr>
<tr>
<td>$2P$</td>
<td>LD_{177}</td>
</tr>
<tr>
<td>$-P$</td>
<td>LD_{6}</td>
</tr>
<tr>
<td>$n \cdot P$</td>
<td>W_{3}27522</td>
</tr>
</tbody>
</table>

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 Lópe-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 $\frac{1}{7}A + ID$. Let’s look at $B-233$, so that $l = 233$, $I/M \approx 200.5$ and $M/S \approx 1.6$. By using the
formulas listed in Table 2.2, the following costs can be estimated:

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

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.

<table>
<thead>
<tr>
<th>operation</th>
<th>$B=163$</th>
<th>$B=233$</th>
<th>$B=283$</th>
<th>$B=409$</th>
<th>$B=571$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \mod b$</td>
<td>$R_{156}$</td>
<td>$R_{120}$</td>
<td>$R_{281}$</td>
<td>$R_{246}$</td>
<td>$R_{565}$</td>
</tr>
<tr>
<td>$a + b$</td>
<td>$A_{45}$</td>
<td>$A_{69}$</td>
<td>$A_{88}$</td>
<td>$A_{153}$</td>
<td>$A_{239}$</td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td>$K_{4343}$</td>
<td>$K_{7266}$</td>
<td>$K_{412714}$</td>
<td>$K_{525094}$</td>
<td>$K_{77572}$</td>
</tr>
<tr>
<td>$a^2$</td>
<td>$P_{4621}$</td>
<td>$P_{3282}$</td>
<td>$P_{5520}$</td>
<td>$P_{9280}$</td>
<td>$P_{1019431}$</td>
</tr>
<tr>
<td>$a^{-1}$</td>
<td>$D_{2904}$</td>
<td>$B_{56426}$</td>
<td>$B_{78188}$</td>
<td>$B_{166485}$</td>
<td>$B_{314153}$</td>
</tr>
<tr>
<td>$P + Q$</td>
<td>$L_{27795}$</td>
<td>$L_{53370}$</td>
<td>$L_{83751}$</td>
<td>$L_{206658}$</td>
<td>$L_{428432}$</td>
</tr>
<tr>
<td>$2P$</td>
<td>$L_{5744}$</td>
<td>$L_{7756}$</td>
<td>$L_{10601}$</td>
<td>$L_{28083}$</td>
<td>$L_{80703}$</td>
</tr>
<tr>
<td>$-P$</td>
<td>$L_{2449553}$</td>
<td>$W_{12443090}$</td>
<td>$W_{21601999}$</td>
<td>$W_{9854413}$</td>
<td>$W_{287818708}$</td>
</tr>
</tbody>
</table>

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 reasons, 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

---

4These 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.
Experimental results

7 – Experimental results

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.

<table>
<thead>
<tr>
<th>Curve</th>
<th>CryptoGen time [ns]</th>
<th>OpenSSL time [ns]</th>
<th>Crypto++ time [ns]</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-163</td>
<td>27 522</td>
<td>180 007</td>
<td>15 732 963</td>
<td>6.5</td>
</tr>
<tr>
<td>B-233</td>
<td>40 036</td>
<td>232 475</td>
<td>36 693 675</td>
<td>5.8</td>
</tr>
<tr>
<td>B-283</td>
<td>65 832</td>
<td>466 992</td>
<td>62 668 174</td>
<td>7.1</td>
</tr>
<tr>
<td>B-409</td>
<td>163 318</td>
<td>735 405</td>
<td>183 510 370</td>
<td>4.5</td>
</tr>
<tr>
<td>B-571</td>
<td>1 240 570</td>
<td>1 848 002</td>
<td>565 496 848</td>
<td>1.5</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th></th>
<th>CryptoGen</th>
<th>OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$B$-233</td>
<td>$B$-233</td>
</tr>
<tr>
<td>time [ns]</td>
<td>40 036</td>
<td>232 475</td>
</tr>
<tr>
<td>ratio</td>
<td>–</td>
<td>5.8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>CryptoGen</th>
<th>OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$B$-283</td>
<td>$B$-283</td>
</tr>
<tr>
<td>time [ns]</td>
<td>65 832</td>
<td>466 992</td>
</tr>
<tr>
<td>ratio</td>
<td>–</td>
<td>1.1</td>
</tr>
</tbody>
</table>

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](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.

![Graph showing the performance comparison of different curves](image)

(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. $\Gamma$ and $\Delta$) range over list of formulas;
- the symbol $\tau$ represents a basic type in a logic system;
- the function $fv(\Gamma)$ retrieves the set of free variables$^1$ in $\Gamma$;
- 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 $\Gamma \vdash X : T$ means that if the variables in $\Gamma$ are linear (they appear only once) and the variables in $\Delta$ are non-linear, then it can be implied that $X : T$.

A.1 CL rules

The proof rules for the Classical Logic (CL) are:

\[
\frac{}{X \vdash X} \text{ axiom}
\]

\[
\frac{\Gamma \vdash X, \Delta, \Gamma', X \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta} \text{ cut}
\]

\[
\frac{\Gamma \vdash \Delta}{\Gamma, X \vdash \Delta} \text{ weakening}_l
\]

\[
\frac{\Gamma \vdash X, \Delta}{\Gamma \vdash \Delta} \text{ weakening}_r
\]

$^1$In $\lambda$-calculus, a variable is free in a $\lambda$-term when it does not appear as an input parameter in a $\lambda$-abstraction. For instance, in $\lambda x.(x + y)$ the variable $x$ is not free (it is bound), while $y$ is free.
A.2 IL rules

The proof rules for the Intuitionistic Logic (IL) are:

\[
\begin{align*}
\frac{\Gamma, X, X \vdash \Delta}{\Gamma, X \vdash \Delta} & \quad \text{contraction} \\
\frac{\Gamma, X \vdash \Delta}{\Gamma, X \land Y \vdash \Delta^L_1} & \quad \frac{\Gamma, Y \vdash \Delta}{\Gamma, X \land Y \vdash \Delta^L_2} \\
\frac{\Gamma, X \vdash \Delta, \Gamma', Y \vdash \Delta'}{\Gamma, \Gamma', X \lor Y \vdash \Delta, \Delta'} & \quad \frac{\Gamma, X, \Delta \vdash Y, \Delta'}{\Gamma, \Gamma', X \lor Y \vdash \Delta, \Delta'} \\
\frac{\Gamma, X \vdash \Delta \quad \Gamma, X \Rightarrow Y \vdash \Delta \quad \Delta' \Rightarrow \Gamma}{\Gamma, \Gamma', X \Rightarrow Y \vdash \Delta, \Delta'} & \quad \frac{\Gamma, X \vdash \Delta \quad \Gamma, X \Rightarrow Y \vdash \Delta \quad \Delta \Rightarrow \Gamma}{\Gamma, \Gamma', X \Rightarrow Y \vdash \Delta, \Delta'} \\
\frac{\Gamma, 1 \vdash \Delta}{\Gamma \vdash \Delta} & \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash 0} \\
\frac{\Gamma \vdash X \lor Y \Rightarrow i}{0 \vdash 0} & \quad \frac{\Gamma, X \vdash Y \Rightarrow e}{0 \vdash X} \\
\end{align*}
\]

A.3 LL rules

The proof rules for the Linear Logic (LL) are:

\[
\begin{align*}
\frac{X_1, \ldots, X_n \vdash X}{axiom} \\
\frac{\Gamma \vdash X \quad \Gamma \vdash Y}{\Gamma \vdash X \land Y} & \quad \frac{\Gamma \vdash X \land Y}{\Gamma \vdash X} \quad \frac{\Gamma \vdash X \land Y}{\Gamma \vdash Y} \\
\frac{\Gamma \vdash X}{\Gamma \vdash X \lor Y \Rightarrow i} & \quad \frac{\Gamma \vdash Y}{\Gamma \vdash X \lor Y \Rightarrow i} \quad \frac{\Gamma \vdash X \lor Y}{\Gamma \vdash X} \quad \frac{\Gamma \vdash X \lor Y}{\Gamma \vdash Y} \\
\frac{\Gamma, X \Rightarrow Y \Rightarrow i}{\Gamma \vdash 1} & \quad \frac{\Gamma \vdash 0}{\Gamma \vdash X \Rightarrow e} \\
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma \vdash \Delta \quad \Gamma', \Delta' \vdash \Delta}{\text{cut}} \\
\frac{\Gamma \vdash \Delta}{\Gamma, !X \vdash \Delta \Rightarrow weakening} & \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?X \vdash \Delta \Rightarrow weakening_r} \\
\end{align*}
\]

91
A.4 DLAL rules

The proof rules for the Dual Light Affine Logic (DLAL) are:

\[
\begin{align*}
\Gamma, ! X, ! X & \vdash \Delta \quad \text{contraction} & \Gamma, ? X, ? X & \vdash \Delta \quad \text{contraction}_r \\
\Gamma, ! X, Y & \vdash \Delta & \Gamma, ? X, Y & \vdash \Delta' \quad \text{contract}_{r} \\
\Gamma, X & \vdash \Delta & \Gamma_{l}, X @ Y & \vdash \Delta & \Gamma'_{l}, X @ Y, \Delta & \vdash \Delta'_{r} \quad \text{weakening} \\
\Gamma, X & \vdash \Delta & \Gamma_{r}, X, Y & \vdash \Delta & \Gamma_{r}'_{r}, X, Y, \Delta & \vdash \Delta'_{l} \quad \text{contraction} \\
\Gamma, ! X & \vdash \Delta & \Gamma_{l}, Y & \vdash \Delta & \Gamma_{l}'_{l}, Y & \vdash \Delta \\
\Gamma, ? X & \vdash \Delta & \Gamma_{r}, Y & \vdash \Delta & \Gamma_{r}'_{r}, Y & \vdash \Delta \\
\Gamma, X & \vdash \Delta & \Gamma_{l}, X & \vdash \Delta & \Gamma_{l}'_{l}, X & \vdash \Delta \\
\Gamma, ! X, Y & \vdash \Delta & \Gamma_{r}, X & \vdash \Delta & \Gamma_{r}'_{r}, X & \vdash \Delta \\
\Gamma, ! X & \vdash \Delta & \Gamma_{l}, X & \vdash \Delta & \Gamma_{l}'_{l}, X & \vdash \Delta \\
\Gamma, ? X & \vdash \Delta & \Gamma_{r}, X & \vdash \Delta & \Gamma_{r}'_{r}, X & \vdash \Delta \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, ! X & \vdash \Delta' & \Gamma_{l}, ! X & \vdash \Delta' & \Gamma_{l}'_{l}, ! X & \vdash \Delta' \\
\Gamma, ? X & \vdash \Delta' & \Gamma_{r}, ? X & \vdash \Delta' & \Gamma_{r}'_{r}, ? X & \vdash \Delta' \\
\Gamma, ! X & \vdash \Delta' & \Gamma_{l}, ! X & \vdash \Delta' & \Gamma_{l}'_{l}, ! X & \vdash \Delta' \\
\Gamma, ? X & \vdash \Delta' & \Gamma_{r}, ? X & \vdash \Delta' & \Gamma_{r}'_{r}, ? X & \vdash \Delta' \\
\end{align*}
\]
A.5 TFA rules

The proof rules for the Typeable Functional Assembly (TFA) are:

\[
\frac{\Gamma \vdash x : T}{\Gamma \vdash \text{e} : x : T} \quad \text{axiom}
\]

\[
\frac{\Gamma, \Delta \vdash x : T_1 \vdash y : T_2}{\Gamma, \Delta \vdash \lambda x. y : T_1 \rightarrow T_2} \quad \text{\textemdash}_i
\]

\[
\frac{\Gamma, \Delta \vdash \tau \notin \text{fv} (\Delta, \Gamma)}{\Gamma \vdash \forall \tau. T} \quad \text{\textemdash}_v
\]

\[
\frac{\Gamma, \Delta \vdash x : T_1 \vdash y : T_1 \vdash z : T_2}{\Gamma, \Delta \vdash \tau \notin \text{fv} (\Delta, \Gamma)}
\]

\[
\frac{\Gamma, \Delta \vdash x : T_1 \vdash y : T_1 \vdash z : T_2}{\Gamma, \Delta \vdash \forall \tau. T} \quad \text{\textemdash}_v
\]

\[
\frac{\Gamma \vdash \text{e} : z \rightarrow y : T_2}{\Gamma, \Delta \vdash \Delta' \vdash xy : T_2} \quad \text{\textemdash}_e
\]

\[
\frac{\Gamma \vdash x : T_1 \vdash y : y : T_1 \vdash z : T_2}{\Gamma \vdash \Delta' \vdash z \rightarrow y : T_2} \quad \text{\textemdash}_e
\]

\[
\frac{\Gamma \vdash \tau \notin \text{fv} (\Delta, \Gamma)}{\Gamma \vdash \forall \tau. T} \quad \text{\textemdash}_v
\]

\[
\frac{\Gamma \vdash \tau \notin \text{fv} (\Delta, \Gamma)}{\Gamma \vdash \forall \tau. T} \quad \text{\textemdash}_v
\]
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:

\[
\begin{align*}
\text{Unit} &::= \text{“unit” Annotation}^* \text{ GlobalSymbol}^*; \\
\text{Annotation} &::= \text{“:” Id | “::” Id “(” (Id | String | Int)+ “)”} \\
\text{GlobalSymbol} &::= \text{LocalSymbol | “function” Id Function} \\
\text{LocalSymbol} &::= \text{Id “in” Variable} \\
\text{ParameterSymbol} &::= \text{Id “in” Parameter}
\end{align*}
\]
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

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 ::= "=+" | ";=" | "*/" | ";\" |
             "<" | ";<=" | "<" | ";==" | ";>=" |

Boolean ::= "true" | "false"
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.

<table>
<thead>
<tr>
<th>synopsis</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:algebraicSimplification</td>
<td>performs an algebraic simplification pass</td>
</tr>
<tr>
<td>:commonSubExpressionElimination</td>
<td>performs a common sub-expression elimination pass</td>
</tr>
<tr>
<td>:constantFolding</td>
<td>performs a constant folding pass</td>
</tr>
<tr>
<td>:controlFlowGraphAnalysis</td>
<td>performs a control flow graph analysis pass</td>
</tr>
<tr>
<td>:copyPropagation</td>
<td>performs a copy propagation pass</td>
</tr>
<tr>
<td>:dagAnalysis</td>
<td>performs a DAG analysis pass</td>
</tr>
<tr>
<td>:deadCodeElimination</td>
<td>performs a dead code elimination pass</td>
</tr>
<tr>
<td>:dumpAll</td>
<td>creates an HRL file after each transformation pass</td>
</tr>
<tr>
<td>:import(x)</td>
<td>imports the exported functions contained in the file x</td>
</tr>
<tr>
<td>:labelsPrefix(x)</td>
<td>sets to x the prefix of the automatically generated labels’ names during the middle-end stage</td>
</tr>
<tr>
<td>:labelsSuffix(x)</td>
<td>sets to x the suffix of the automatically generated labels’ names during the middle-end stage</td>
</tr>
<tr>
<td>:liveVariableAnalysis</td>
<td>performs a live variable analysis pass</td>
</tr>
<tr>
<td>:operatorChainAnalysis</td>
<td>performs an operator chain analysis pass</td>
</tr>
<tr>
<td>:optimization</td>
<td>performs all the optimization passes until no code change is possible</td>
</tr>
<tr>
<td>:strengthReduction</td>
<td>performs a strength reduction pass</td>
</tr>
<tr>
<td>:symbolUsageAnalysis</td>
<td>performs a symbol usage analysis pass</td>
</tr>
<tr>
<td>:variablesPrefix(x)</td>
<td>sets to x the prefix of the automatically generated variables’ names during the middle-end stage</td>
</tr>
<tr>
<td>:variablesSuffix(x)</td>
<td>sets to x the suffix of the automatically generated variables’ names during the middle-end stage</td>
</tr>
</tbody>
</table>

Table C.1: Standard unit annotations.
Table C.2: Unit annotations offered by the aXiom front-end.

<table>
<thead>
<tr>
<th>synopsis</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:dumpHRL</td>
<td>creates a file containing the aXiom translation into HRL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>synopsis</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:bitVectorPacking</td>
<td>enables the bit vector packing</td>
</tr>
<tr>
<td>:cLabelsPrefix(x)</td>
<td>sets to x the prefix of the automatically generated labels’ names during the C back-end stage</td>
</tr>
<tr>
<td>:cLabelsSuffix(x)</td>
<td>sets to x the suffix of the automatically generated labels’ names during the C back-end stage</td>
</tr>
<tr>
<td>:cVariablesPrefix(x)</td>
<td>sets to x the prefix of the automatically generated variables’ names during the C back-end stage</td>
</tr>
<tr>
<td>:cVariablesSuffix(x)</td>
<td>sets to x the suffix of the automatically generated variables’ names during the C back-end stage</td>
</tr>
<tr>
<td>:findLastSetWindow(x)</td>
<td>sets the findlastset window to x bit</td>
</tr>
<tr>
<td>:loopUnrolling</td>
<td>enables the loop unrolling</td>
</tr>
<tr>
<td>:targetIdiomUsage</td>
<td>enables the target idiom usage</td>
</tr>
</tbody>
</table>

Table C.3: Unit annotations offered by the C back-end.

<table>
<thead>
<tr>
<th>synopsis</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:carryLessMultiplication</td>
<td>indicates that the function implements a carry-less multiplication</td>
</tr>
<tr>
<td>:carryLessSquaring</td>
<td>indicates that the function implements a carry-less squaring</td>
</tr>
<tr>
<td>:export</td>
<td>exports the function, making it callable from the outside of the compilation unit</td>
</tr>
<tr>
<td>:hasAbsorbingElement(x)</td>
<td>indicates that x is an absorbing element for the function</td>
</tr>
<tr>
<td>:hasIdentityElement(x)</td>
<td>indicates that x is an identity element for the function</td>
</tr>
<tr>
<td>:hasSpecialCase1(x, y)</td>
<td>indicates that if the first input parameter of the function is x, then the result is y</td>
</tr>
<tr>
<td>:hasSpecialCase2(x, y)</td>
<td>indicates that if the second input parameter of the function is x, then the result is y</td>
</tr>
<tr>
<td>:hasSpecialization(x)</td>
<td>indicates that x is a specialized function of the current one, when its input parameters are the same</td>
</tr>
<tr>
<td>:isCommutativeAndAssociative</td>
<td>indicates that the function is commutative and associative</td>
</tr>
<tr>
<td>:test(x)</td>
<td>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</td>
</tr>
</tbody>
</table>

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.
Table D.1: Performance on the i7 processor.
D – Additional results

<table>
<thead>
<tr>
<th>operation</th>
<th>method</th>
<th>time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>B-163</td>
</tr>
<tr>
<td>$a \mod b$</td>
<td><strong>REDUCE</strong></td>
<td>104</td>
</tr>
<tr>
<td>$a + b$</td>
<td><strong>ADD</strong></td>
<td>45</td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td><strong>COMB/3</strong></td>
<td>6546</td>
</tr>
<tr>
<td></td>
<td><strong>COMB/4</strong></td>
<td>4869</td>
</tr>
<tr>
<td></td>
<td><strong>COMB/5</strong></td>
<td>5207</td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td><strong>KARATSUBA+COMB/3</strong></td>
<td>4140</td>
</tr>
<tr>
<td></td>
<td><strong>KARATSUBA+COMB/4</strong></td>
<td>[3743]</td>
</tr>
<tr>
<td></td>
<td><strong>KARATSUBA+COMB/5</strong></td>
<td>4188</td>
</tr>
<tr>
<td></td>
<td><strong>WINDOWMULTIPLY/32</strong></td>
<td>–</td>
</tr>
<tr>
<td>$a^2$</td>
<td><strong>PRECOMPUTESQUARE/8</strong></td>
<td>[1621]</td>
</tr>
<tr>
<td></td>
<td><strong>PRECOMPUTESQUARE/9</strong></td>
<td>1745</td>
</tr>
<tr>
<td></td>
<td><strong>PRECOMPUTESQUARE/10</strong></td>
<td>1865</td>
</tr>
<tr>
<td></td>
<td><strong>WINDOWSQUARE/32</strong></td>
<td>–</td>
</tr>
<tr>
<td>$a^{-1}$</td>
<td><strong>BEA</strong></td>
<td>32023</td>
</tr>
<tr>
<td></td>
<td><strong>DCEA</strong></td>
<td>29484</td>
</tr>
<tr>
<td></td>
<td><strong>EEA</strong></td>
<td>35071</td>
</tr>
<tr>
<td>$P + Q$</td>
<td>in affine space</td>
<td>50937</td>
</tr>
<tr>
<td></td>
<td>in projective space</td>
<td>51920</td>
</tr>
<tr>
<td></td>
<td>in Jacobian space</td>
<td>64079</td>
</tr>
<tr>
<td></td>
<td>in López-Dahab space</td>
<td>58365</td>
</tr>
<tr>
<td>$2P$</td>
<td>in affine space</td>
<td>36107</td>
</tr>
<tr>
<td></td>
<td>in projective space</td>
<td>43599</td>
</tr>
<tr>
<td></td>
<td>in Jacobian space</td>
<td>41867</td>
</tr>
<tr>
<td></td>
<td>in López-Dahab space</td>
<td>[27795]</td>
</tr>
<tr>
<td>$-P$</td>
<td>in affine space</td>
<td>262</td>
</tr>
<tr>
<td></td>
<td>in projective space</td>
<td>262</td>
</tr>
<tr>
<td></td>
<td>in Jacobian space</td>
<td>4139</td>
</tr>
<tr>
<td></td>
<td>in López-Dahab space</td>
<td>5744</td>
</tr>
<tr>
<td>$n \cdot P$</td>
<td><strong>ADDANDDOUBLE</strong></td>
<td>9441774</td>
</tr>
<tr>
<td></td>
<td><strong>NAFMULTIPLY</strong></td>
<td>7978762</td>
</tr>
<tr>
<td></td>
<td><strong>NAFWMULTIPLY/2</strong></td>
<td>[4649553]</td>
</tr>
<tr>
<td></td>
<td><strong>NAFWMULTIPLY/3</strong></td>
<td>4720973</td>
</tr>
<tr>
<td></td>
<td><strong>NAFWMULTIPLY/4</strong></td>
<td>4892246</td>
</tr>
<tr>
<td></td>
<td><strong>NAFWMULTIPLY/5</strong></td>
<td>5244866</td>
</tr>
</tbody>
</table>

Table D.2: Performance on the ARMv7 processor.
### Table D.3: Performance of the OpenSSL library on the i7 processor.

<table>
<thead>
<tr>
<th>operation</th>
<th>B-163</th>
<th>B-233</th>
<th>B-283</th>
<th>B-409</th>
<th>B-571</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \mod b$</td>
<td>16</td>
<td>194</td>
<td>186</td>
<td>217</td>
<td>252</td>
</tr>
<tr>
<td>$a + b$</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td>255</td>
<td>280</td>
<td>303</td>
<td>375</td>
<td>501</td>
</tr>
<tr>
<td>$a^2$</td>
<td>241</td>
<td>275</td>
<td>272</td>
<td>329</td>
<td>343</td>
</tr>
<tr>
<td>$a^{-1}$</td>
<td>1955</td>
<td>3354</td>
<td>4768</td>
<td>9927</td>
<td>17657</td>
</tr>
<tr>
<td>$P + Q$</td>
<td>2588</td>
<td>3974</td>
<td>5899</td>
<td>10486</td>
<td>18524</td>
</tr>
<tr>
<td>$2P$</td>
<td>2543</td>
<td>3892</td>
<td>6043</td>
<td>10307</td>
<td>18308</td>
</tr>
<tr>
<td>$-P$</td>
<td>25</td>
<td>27</td>
<td>39</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>$n \cdot P$</td>
<td>180007</td>
<td>232475</td>
<td>466992</td>
<td>735405</td>
<td>1848002</td>
</tr>
</tbody>
</table>

### Table D.4: Performance of the Crypto++ library on the i7 processor.

<table>
<thead>
<tr>
<th>operation</th>
<th>B-163</th>
<th>B-233</th>
<th>B-283</th>
<th>B-409</th>
<th>B-571</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \mod b$</td>
<td>1105</td>
<td>1698</td>
<td>2240</td>
<td>3713</td>
<td>6127</td>
</tr>
<tr>
<td>$a + b$</td>
<td>97</td>
<td>98</td>
<td>100</td>
<td>99</td>
<td>102</td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td>1552</td>
<td>3434</td>
<td>3567</td>
<td>6759</td>
<td>12922</td>
</tr>
<tr>
<td>$a^2$</td>
<td>130</td>
<td>291</td>
<td>181</td>
<td>229</td>
<td>282</td>
</tr>
<tr>
<td>$a^{-1}$</td>
<td>10530</td>
<td>14142</td>
<td>20724</td>
<td>25520</td>
<td>46914</td>
</tr>
<tr>
<td>$P + Q$</td>
<td>22185</td>
<td>30985</td>
<td>47295</td>
<td>63869</td>
<td>131279</td>
</tr>
<tr>
<td>$2P$</td>
<td>21353</td>
<td>30274</td>
<td>45535</td>
<td>63351</td>
<td>124767</td>
</tr>
<tr>
<td>$-P$</td>
<td>103</td>
<td>103</td>
<td>110</td>
<td>106</td>
<td>112</td>
</tr>
<tr>
<td>$n \cdot P$</td>
<td>15732963</td>
<td>36693675</td>
<td>62668174</td>
<td>183510370</td>
<td>565496848</td>
</tr>
</tbody>
</table>
## Notations

The following table lists all the mathematical notations used in this dissertation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^*$</td>
<td>Kleene closure of $x$</td>
</tr>
<tr>
<td>$x^+$</td>
<td>positive closure of $x$</td>
</tr>
<tr>
<td>$\circ x \times y \rightarrow z$</td>
<td>binary operator $\circ$ that maps $x \times y$ to $z$</td>
</tr>
<tr>
<td>$\circ x \rightarrow y$</td>
<td>unary operator $\circ$ that maps $x$ to $y$</td>
</tr>
<tr>
<td>$x \mapsto y$</td>
<td>$x$ is mapped to $y$</td>
</tr>
<tr>
<td>$\text{Im}(f)$</td>
<td>image of the function $f$</td>
</tr>
<tr>
<td>$x \leftarrow y$</td>
<td>$y$ is copied into $x$</td>
</tr>
<tr>
<td>$X[y]$</td>
<td>access of the $y$-th element in the tuple $X$</td>
</tr>
<tr>
<td>$X[y \rightarrow]$</td>
<td>access from the $y$-th element to the $z$-th element in the tuple $X$</td>
</tr>
<tr>
<td>$x @ y$</td>
<td>exclusive or between $x$ and $y$</td>
</tr>
<tr>
<td>$X \gg y$</td>
<td>right shift of $X$ by $y$ positions</td>
</tr>
<tr>
<td>$X \ll y$</td>
<td>left shift of $X$ by $y$ positions</td>
</tr>
<tr>
<td>$\text{fls}(X)$</td>
<td>find last set of $X$</td>
</tr>
<tr>
<td>$x \leftrightarrow y$</td>
<td>content exchange between the variables $x$ and $y$</td>
</tr>
<tr>
<td>${x, y, z}$</td>
<td>unordered set of the items $x$, $y$ and $z$</td>
</tr>
<tr>
<td>$X \setminus Y$</td>
<td>set difference between $X$ and $Y$</td>
</tr>
<tr>
<td>$X \cup Y$</td>
<td>set union of $X$ and $Y$</td>
</tr>
<tr>
<td>$[x, y]$</td>
<td>integer range from $x$ (inclusive) and $y$ (inclusive)</td>
</tr>
<tr>
<td>$(x, y, z)$</td>
<td>ordered tuple of the items $x$, $y$ and $z$</td>
</tr>
<tr>
<td>$(x, y, z)_2$</td>
<td>bit vector containing the bits $x$, $y$ and $z$</td>
</tr>
<tr>
<td>$(x, y, z)_{\text{NAF}}$</td>
<td>$\text{NAF}$ containing the values $x$, $y$ and $z$</td>
</tr>
<tr>
<td>$(x, y, z)_{\text{NAF}_w}$</td>
<td>$\text{NAF}_w$ containing the values $x$, $y$ and $z$</td>
</tr>
<tr>
<td>$x_{16}$</td>
<td>$x$ in hexadecimal form</td>
</tr>
<tr>
<td>$(X, \circ)$</td>
<td>the group $X$ with the group operator $\circ$</td>
</tr>
<tr>
<td>$G$</td>
<td>generic group</td>
</tr>
<tr>
<td>$</td>
<td>X</td>
</tr>
<tr>
<td>$(X, \circ, \cdot)$</td>
<td>the field $X$ with the field operators $\circ$ and $\cdot$</td>
</tr>
<tr>
<td>$\mathbb{F}$</td>
<td>generic field</td>
</tr>
<tr>
<td>$\mathbb{F}_x$</td>
<td>field with order $x$</td>
</tr>
<tr>
<td>$\text{char}(X)$</td>
<td>characteristic of the field $X$</td>
</tr>
<tr>
<td>$\psi$</td>
<td>field homomorphism</td>
</tr>
<tr>
<td>$X/Y$</td>
<td>extension field $X$ over $Y$</td>
</tr>
<tr>
<td>$\deg(X)$</td>
<td>degree of the field $X$</td>
</tr>
<tr>
<td>$\mathbb{F}_{2^x}$</td>
<td>binary field with degree $x$</td>
</tr>
<tr>
<td>$\mathbb{F}_{2^x}[y]/z$</td>
<td>binary field with degree $x$ in $y$ with modulus $z$</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>elliptic curve</td>
</tr>
<tr>
<td>$\mathcal{E}/X$</td>
<td>elliptic curve $\mathcal{E}$ defined over the field $X$</td>
</tr>
</tbody>
</table>
$\mathcal{E}(X)$ set of points of the elliptic curve $\mathcal{E}$ defined over the field $X$

$P_\infty$ 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 $Y$

$x \land y$ logical and between $x$ and $y$

$x \lor y$ logical r between $x$ and $y$

$x \Rightarrow y$ logical implication between $x$ and $y$

$\neg x$ logical not of $x$

$x \nabla y$ 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 $y$

$x \rightarrow y$ linear implication between $x$ and $y$

$! x$ of course $x$

$? x$ why not $x$

$\exists x$ neutral $x$

$x : y$ typed context of $x$ with type $y$

$(x, y, z)$ threaded words $x$, $y$ and $z$

fv$(X)$ set of the free variables in $X$

$X[y]$ clash-free substitution of $y$ in $X$

$x[y] \vdash z : w$ DLAL sequent with antecedent $x[y]$ 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
Acronyms

**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 \( \lambda \)-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

\(\lambda\)-abstractions, 36
\(\lambda\)-applications, 36
\(\lambda\)-calculus, 36
\(\lambda\)-term, 36
2G, 4
A5/1, 4
A5/2, 4
A5/3, 4
Abelian group, 12
add-and-double, 27
additive conjunction, 35
additive disjunction, 35
admissible change of variables, 16
affine coordinate system, 25
algebraic simplification, 68
alternation, 94
analysis algorithm, 62
analysis pass, 47
annotation, 52, 99
antecedent, 34
associativity of a group, 12
aXiom, 55, 95
back-end
stage, 48
basic block, 62
BEA, 24, 41
binary Euclidean algorithm, 24, 41
binary field, 12, 13
bit vector packing, 75
bound variable, 90
Brouwer-Heyting-Kolmogorov functional interpretation, 34
CAO, 6
characteristic of a field, 12, 13
Church boolean, 38
Church word, 38
CL, 6, 34
clash free substitution, 90
classical logic, 34
closure of a group, 12
code compilation, 46
comb, 21
combA, 21
common sub-expression, 73
common sub-expression elimination, 73
commutative group, 12
COMP128, 4
concatenation, 94
consequent, 34
constant folding, 68
context, 34
collection flow graph, 62
collection flow graph analysis, 62
copy propagation, 70
cPLC, 7
Crypto++, 84
CryptoGen, 46
cryptography developer, 45
Cryptol, 7
CryptoStudio, 45
Curve25519, 8
DAG, 64
DAG analysis, 64
data instance, 59
DCEA, 24, 43
dead code, 70
dead code elimination, 70
dead variable, 63
defined variable, 63
degree of a field, 12, 13
digital signature standard, 9
discrete logarithm, 16
distributivity of a field, 12
DLAL, 38
DLAL certified Euclidean algorithm, 24, 43
DNS, 1
DNS cache poisoning attack, 1
DNS spoofing attack, 1
DNSCurve, 1
DNSSEC, 1
dSS, 9
dual light affine logic, 38

ECC, x, 7, 11
ECC challenge, 8
ECDDHP, 19
ECDH, 9, 18, 31
ECDHP, 18
ECDFP, 16
ECDSS, 9
ECMQV, 9
EEA, 25
elliptic curve, 13, 14
elliptic curve cryptography, x, 11
elliptic curve decision Diffie-Hellman problem, 19
elliptic curve Diffie-Hellman, 9, 18, 31
elliptic curve Diffie-Hellman problem, 18
elliptic curve digital signature algorithm, 9
elliptic curve discrete logarithm problem, 16
elliptic curve DSA, 9
elliptic curve Menezes-Qu-Vanstone, 9
elliptic curve MQV, 9
end-compiler, 61
end-user, 45
extended Euclidean algorithm, 25
extension field, 12, 13

field, 12
field addition, 20
Field degree, 13
field division, 24
field homomorphism, 12, 13
field inversion, 24
field modular reduction, 20
field modulus, 13
field multiplication, 21
field multiplicative inversion, 24
field operation, 12
field order, 12
field squaring, 23
field subtraction, 20
finite field, 13
finite group, 12
FIPS PUB 186-4, 9
framework developer, 45
free variable, 90
front-end stage, 47
function, 59

Google, 3
Google Search, 3
group, 11, 12
group operation, 12
group order, 12
GSM, 4

head of a tuple, 39
hidden field equations, 10
high speed cryptography, x, 1
HR, 59
HRL, 52, 94
hyperelliptic curve cryptography, 10
ICC, 33
IL, 34
implicit computational complexity, 33
instance, 59
intuitionistic logic, 34
IPsec, 9

Jacobian coordinate system, 25
Karatsuba, 22
Karatsuba-Ofman, 22
KASUMI, 4
Kleene closure, 94

López-Dahab coordinate system, 25
label, 59
leader instruction, 62
library builder, 50
linear implication, 35
linear logic, 35
linear variable, 90
live variable, 63
live variable analysis, 63
LL, 35
logical, 52
loop unrolling, 75
loop unwinding, 75

MANET, 9
maq, 6
INDEX

McEliece crypto-system, 10
middle-end stage, 47
minimal logic, 36
MISTY1, 4
mobile ad-hoc network, 9
modulus, 12
mote, 9
multiplicative conjunction, 35
multiplicative disjunction, 35
NAF, 29
NAF_w, 30
non-supersingular elliptic curve, 16
non-terminal symbol, 94

of course, 35
one-way function, 16
OpenSSL, 84
operator chain, 66
operator chain analysis, 66
operator overloading, 55
optimization, 61
order of a field, 12
order of a group, 12
pass, 47
performance evaluator, 48
point addition, 14, 25
point at infinity, 14
point doubling, 25
point multiplication, 15, 27
point negation, 25
point subtraction, 15
positive closure, 94
preprocessing, 55
projective coordinate system, 25

qhasm, 6

rational point, 13
register filling, 75
register spilling, 75
relative, 52
repetition, 94
scalar multiplication, 15, 27
school-book method, 21
SEC 2, 9
sequent, 34
set of points on an elliptic curve, 14
side effect, 36
SIDH, 8
signed-binary digit representation, 29
signed-digit representation, 28
SNOW 3G, 4
SSH, 9
static constant, 55
static for, 55
static if, 55
station-to-station, 19
strength reduction, 71
STS, 19
succeedent, 34
supersingular elliptic curve, 16
symbol, 59
tail of a tuple, 39
target idiom usage, 76
terminal symbol, 94
test case, 59
TFA, 38
threaded word, 38
three-address code, 52
TLS, 9
trace zero varieties, 10
transformation algorithm, 66
transformation pass, 47
tuple comprehension, 55
tuple slicing, 55
typeable functional assembly, 38
typed λ-calculus, 36

UI, 50
unit, 59
untyped λ-calculus, 36
used variable, 63

VANET, 9
variadic operator, 55
vehicular ad-hoc network, 9

Weierstraß equation, 14
why not, 35
wireless sensor network, 9
WSN, 9
X9.62, 8

ZKCrypt, 7

114
Bibliography


[33] Caroline Ahlström. Ökad säkerhet för dig som är bredbandskund hos TDC Song. 2007. URL: http://www.observer.se/wpyfs/00/00/00/00/00/00/00/35/00/release.html (visited on 03/26/2016) (cit. on p. 2).


