HAL open science

A formalization of elliptic curves for cryptography

Evmorfia-Iro Bartzia

To cite this version:

Evmorfia-Iro Bartzia. A formalization of elliptic curves for cryptography. Cryptography and Security [cs.CR]. Université Paris Saclay (COmUE), 2017. English. NNT : 2017SACLX002 . tel-01563979

HAL Id: tel-01563979

https://pastel.hal.science/tel-01563979v1

Submitted on 18 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

NNT : 2017SACLX002

Thèse de doctorat de l'Université Paris-Saclay préparée à Inria Paris Équipe Prosecco

École doctorale n° 573

INTERFACES - Approches interdisciplinaires : fondements, applications et innovations

Spécialité de doctorat : informatique

par

Evmorfia-Iro Bartzia

A Formalization of Elliptic Curves for Cryptography

Thèse présentée et soutenue à Inria-Paris, le 15 février 2015.

Composition du Jury :

M. Gilles Barthe, *Professeur* — Relecteur
IMDEA Software Institute, Madrid, Espagne

M. Karthikeyan Bhargavan, *Directeur de recherche* — Directeur de thèse
Inria Paris, France

M. Philippe Guillot, *Maître de conférences* — Examinateur
Université de Paris 8, France

Mme. Assia Mahboubi, *Chargé de recherche* — Examinatrice
Inria Saclay - Île de France, France

Mme. Christine Paulin-Mohring, *Professeur* — Présidente du jury
Université Paris-Sud, France

M. Ben Smith, *Chargé de recherche* — Examinateur
Inria Saclay - Île de France, France

M. Bas Spitters, *Professeur associé* — Relecteur
Université d'Aarhus, Danemark

M. Pierre-Yves Strub, *Maître de conférences* — Co-directeur de thèse
École Polytechnique, France

To Fredo and Robert

Contents

1 Introduction **1**

1.1 History of formal methods . . . 1
1.2 Proof assistants . . . 2
1.3 Use of formal methods in mathematics . . . 3
1.4 Use of formal methods in cryptography . . . 5
1.5 Elliptic curves and cryptography . . . 7
1.6 Contribution of this thesis . . . 8

2 Background **10**

2.1 Mathematical background . . . 10
2.1.1 Elliptic curves definitions . . . 10
2.1.2 Defining addition . . . 14
2.1.3 Riemann Roch and the group law . . . 16
2.2 Use of elliptic curves in cryptography . . . 21
2.2.1 Algorithms for scalar multiplication . . . 25
2.2.2 Use of different coordinate systems . . . 30
2.3 Coq and its Ssreflect extension . . . 31
2.3.1 Propositions and Types . . . 32
2.3.2 Coq by example . . . 33
2.3.3 Functions and Equality . . . 36
2.3.4 Inductive types . . . 36
2.3.5 The Ssreflect extension . . . 39

3 A formal library for elliptic curves **44**

3.1 Elliptic curves definitions . . . 44
3.2 The Picard group of divisors . . . 48
3.2.1 The field of rational functions $\mathbb{K}(\mathcal{E})$ . . . 49
3.2.2 Principal Divisors . . . 59
3.2.3 Divisor of a line . . . 62
3.2.4 The Picard Group . . . 62
3.3 Linking $\mathrm{Pic}(\mathcal{E})$ to $\mathcal{E}(\mathbb{K})$ . . . 64
3.4 The Projective Plane . . . 67

3.5 Related work 76

4 A formalization of the GLV algorithm **78**
4.1 The multi-exponentiation algorithm 80
4.2 The decomposition algorithm 83
4.3 Computing the endomorphisms 88
4.4 The GLV algorithm 91
4.5 Related work 92
4.6 Comments and Future work 93

5 Applications **95**
5.1 Verifying GLV with CoqEAL (future work) 95
5.2 A Verified Library of Elliptic Curves in F* 96
5.3 An Ssreflect library for monoidal algebras 100

6 Conclusion **103**

References **106**

Une formalisation des courbes elliptiques pour la cryptographie.

Le sujet de ma thèse s'inscrit dans le domaine des preuves formelles et de la vérification des algorithmes cryptographiques. L'implémentation des algorithmes cryptographiques est souvent une tâche assez compliquée, parce qu'ils sont optimisés pour être efficaces et sûrs en même temps. Par conséquent, il n'est pas toujours évident qu'un programme cryptographique en tant que fonction, corresponde exactement à l'algorithme mathématique, c'est-à-dire que le programme soit correct. Les erreurs dans les programmes cryptographiques peuvent mettre en danger la sécurité de systèmes cryptographiques entiers et donc, des preuves de correction sont souvent nécessaires. Les systèmes formels et les assistants de preuves comme Coq et Isabelle-HOL sont utilisés pour développer des preuves de correction des programmes. Les courbes elliptiques sont largement utilisées en cryptographie surtout en tant que groupe cryptographique très efficace. Pour le développement des preuves formelles des algorithmes utilisant les courbes elliptiques, une théorie formelle de celles-ci est nécessaire. Dans ce contexte, nous avons développé une théorie formelle des courbes elliptiques en utilisant l'assistant de preuves Coq. Cette théorie est par la suite utilisée pour prouver la correction des algorithmes de multiplication scalaire sur le groupe des points d'une courbe elliptique.

Plus précisément, mes travaux de thèse peuvent être divisées en deux parties principales. La première concerne le développement de la théorie des courbes elliptiques en utilisant l'assistant des preuves Coq. Notre développement de plus de 15000 lignes de code Coq comprend la formalisation des courbes elliptiques données par une équation de Weierstrass, la théorie des corps des fonctions rationnelles sur une courbe, la théorie des groupes libres et des diviseurs des fonctions rationnelles sur une courbe. Notre résultat principal est la formalisation du théorème de Picard ; une conséquence directe de ce théorème est l'associativité de l'opération du groupe des points d'une courbe elliptique qui est un résultat non trivial à prouver. La seconde partie de ma thèse concerne la vérification de l'algorithme GLV pour effectuer la multiplication scalaire sur des courbes elliptiques. Pour ce développement, nous avons vérifier trois algorithmes indépendants : la multiexponentiation dans un groupe, la décomposition du scalaire et le calcul des endomorphismes sur une courbe elliptique. Nous avons également développé une formalisation du plan projectif et des courbes en coordonnées projectives et nous avons prouvé que les deux représentations (affine et projective) sont isomorphes.

Mon travail est à la fois une première approche à la formalisation de la géométrie algébrique élémentaire qui est intégré dans les bibliothèques de Ssreflect mais qui sert aussi à la certification de véritables programmes cryptographiques.

La manuscrit contient six chapitres.

— Le premier chapitre donne une introduction aux méthodes formelles, notamment dans le cadre de la formalisation de résultats mathématiques ainsi que dans le contexte de la sécurité des infrastructures.

— Le deuxième chapitre présente une introduction aux courbes elliptiques et de leur utilisation en cryptographie. Il y est aussi donné une introduction à l'assistant à la preuve Coq et à son extension Ssreflect.

— Le troisième chapitre présente la contribution centrale de cette thèse : une bibliothèque formellement vérifiée en Coq/Ssreflect pour la théorie des courbes elliptiques. Le résultat principal est une preuve de l'existence d'un isomorphisme entre les points d'une courbe elliptique et le groupe de Picard de ses diviseurs. Ces travaux ont été publiés dans *Interactive Theorem Proving 2014*.

— Le quatrième chapitre présente une preuve de correction de l'algorithme GLV (Gallant, Lambert, et Vanstone) qui est utilisé pour la multiplication scalaire dans les courbes elliptiques — primitive centrale de certaines constructions cryptographiques. Cet algorithme se décompose en trois parties, dont l'une se repose sur la formalisation des courbes elliptiques présentée au chapitre précédent.

— Le cinquième chapitre conclue avec la présentation de trois applications et souligne l'importance d'établir des liens formels entre bibliothèques formellement vérifiées de courbes elliptiques et les implémentations afférentes. Une partie de ces travaux ont été publiés dans *Computer Security Foundations Symposium 2016*.

— Le dernier chapitre conclue la thèse et donne des directions de recherche pour des travaux futurs.

Mots-clés : Cryptographie, Méthodes formelles (informatique), Courbes elliptiques, Coq (logiciel).

1
Introduction

1.1 History of formal methods

Kurt Gödel, *On Formally Undecidable Propositions of Principia Mathematica and Related Systems I*:

The development of mathematics towards greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.

The idea of mechanized reasoning probably began in the 17th century with Leibniz who was the first to imagine a universal language (*characteristica universalis*) where all statements could be expressed and checked for their truth value via a calculus of reasoning (*calculus ratiocinator*). Interestingly, Leibniz's *characteristica* was not limited to expressing mathematical statements. He imagined a universal language where all controversial statements could be resolved by his calculus and therefore no disagreement could take place. Nevertheless, one had to wait until the beginning of the 20th century, when important progress in the domain of mathematical logic introduced once again Leibniz's idea limited to mathematical statements. The realization that common mathematical statements can be expressed using formal axiomatic systems in such a way that it would be possible (at least in principle) to automatically check if they are correct or not, was one of the most important steps in the history of mathematics in

the 20th century. The first to present such a system was Frege [Fre93] in 1893, followed by Zermelo with axiomatic set theory (in 1908), Russell and Whitehead with ramified type theory in *Principia Mathematica* (in 1913), and Church with simple type theory (in 1940).

Formalization consists of two aspects: (i) expressing statements in some formal language and (ii) develop proofs based on a fixed set of rules, in a way that their correctness can be checked by some algorithm. However, in practice, formalizing mathematical theorems and proofs is extremely difficult to do by hand:

"...the tiniest proof at the beginning of the Theory of Sets would already require several hundreds of signs for its complete formalization... formalized mathematics cannot in practice be written down in full... We shall therefore very quickly abandon formalized mathematics." (Bourbaki)

Even Russell himself stated that his *intellect never recovered from the strain* of writing Principia Mathematica. To sum up, as Rasiowa and Sikorski report: *The mechanical method of deducing some mathematical theorems has no practical value, because this is too complicated in practice.* As a result, the idea of formalizing proofs has not prevailed and few mathematicians have actually exercised it. However, the rise of computer science in the late 20th century has made possible the complete formalization of complex mathematical theorems and proofs. Since then, major progress has been made in the development of axiomatic formal systems, and several impressive results have followed, like the formalization of the prime number theorem [ADGR07] or the four color theorem [Gon07].

1.2 Proof assistants

The use of computers makes formalization a lot more realistic, because computers can check and sometimes even generate proofs. In practice, we use a *proof assistant* or a *theorem prover*, which is a computer program that assists the user develop formal proofs by human - machine interaction. It works as a calculator: the user writes an expression (definition, theorem, proof, ...) in the language and the assistant mechanically checks the validity of the expression. More precisely, the assistant includes a proof engine that provides *tactics* which help the user interactively construct the proofs, and a *kernel* that checks if the proof relies on valid reasoning. There exist many different proof assistants [Wie06] that come with large mathematical libraries, such as *Mizar* based on set theory, *Isabelle-HOL* based on higher order logic, *Coq* based on constructive dependent type theory, *ACL2* based on primitive recursive arithmetic, and *PVS* based on classical dependent type theory.

But why one should believe that a formal proof checked by some proof assistant is equally or even more reliable than a common hand proof? Given the proof assistant's architecture, the validity of a formal statement and its proof is checked by the proof assistant's kernel. Yet, a question naturally arises: If the kernel checks the proof, who checks the kernel? The answer is that one needs to trust that the kernel is correct. Nevertheless, the reliability of proof assistants relies on their architecture: Usually the size of the kernel is much smaller and simpler compared to the prover itself. Keeping the kernel negligibly small and readable by humans minimizes the probability of errors. If the kernel is correct and bug-free, then any proof checked by the kernel is guaranteed to be correct too, and one can be certain that all the mathematical theories formalized on top of it are correct too. As a result, the proof assistant is much easier to trust, since we just need to trust a small readable part and then everything that is constructed on top of it is mechanically checked for correctness. In that sense, any proof checked by a theorem prover is more reliable than a human paper proof, since (as will be explained next) it is not unusual for hand proofs to lack rigor or to contain unspotted errors.

Coq [The10, BC04] is the proof assistant used in this thesis. Coq comes with a pure functional programming language and a set of deduction and computational rules that characterize the logic. There are proof tactics that allow the user to interactively construct proofs and there are libraries of proved mathematical theorems available for use. A short introduction to Coq is given in Section 2.3.

Coq logic is the Calculus of Inductive Constructions [The10, PP89], a dependently typed polymorphic lambda calculus. In contrast to classical logic, Coq logic is constructive, which means that the excluded middle principle does not hold. Furthermore, it means that to prove any existential statement, one needs to provide an explicit witness for the statement to hold. These consequences of constructiveness need to be taken into account, and it remains an important difference when trying to construct a formal proof in Coq from a non-constructive mathematical paper proof. Another significant aspect of Coq logic is the Curry–Howard correspondence: *The relation between a program (i.e. a function) and its type is the same as the relation between a proposition and its proof.* The Curry–Howard correspondence makes the Coq language suitable for writing both programs and logical formulas.

1.3 Use of formal methods in mathematics

To begin with, a first concern is that we want to be certain that the methods used in a mathematical proof are valid. For example, is it correct to use the axiom of choice or the principle of excluded middle in all cases? A second concern is, given a certain background of allowed methods, whether a proof meets these

particular standards: in other words whether it is correct. The debates on the foundations of mathematics at the beginning of the 20th century were aiming to address the first concern of what methods are legitimate to prove a mathematical statement. However, formal methods are not designed to address this particular problem, because formal correctness can be guaranteed only with respect to some axiomatic logical framework, which is determined in advance. In a certain sense, formal methods can be used to check the validity of mathematical statements but only modulo a pre-existing underlying set of rules. Nevertheless, establishing correctness and improving the rigor of mathematical statements is an important issue for which formal methods are extremely helpful.

Indeed, guaranteeing the correctness of a mathematical statement is not a trivial concern: unfortunately, imprecise statements and definitions, missing cases, unclear hypotheses and unexplained inferences are very common in mathematical literature. Moreover, mathematical proofs can sometimes be so complex that even after being subjected to extensive peer-review, mistakes often escape unnoticed. A large number of mathematical proofs have been found to contain errors throughout the years: in 1935 Lecat wrote a book that includes 130 pages of errors made by mathematicians up to 1900 [AH14]. Formal verification is especially interesting for

1. proofs that are very long and complex such as the Classification of finite simple groups or the Seymour-Robertson graph minor theorem,
2. proofs that rely on extensive calculation or that need explicit checking of cases such as the Four-colour theorem or Hales's proof of the Kepler conjecture,
3. proofs where complete rigor is particularly painful such as program verification.

During the last few years, significant formalization efforts have taken place and many impressive results have been formalized such as: the prime number theorem (Avigad et al using Isabelle/HOL, Harrison et al using HOL Light), the four-color theorem (Gonthier et al using Coq), the Jordan curve theorem (Hales et al using HOL Light, Trybulec et al. using Mizar), the Hales proof of the Kepler conjecture (Flyspeck project using HOL light and Isabelle) and the Feit–Thompson Odd Order Theorem (Georges Gonthier et al using Coq). However, maybe even more important than the results themselves are the mathematical theories developed to support these formalizations. Those developments include libraries about number theory, finite group theory, Galois theory, linear algebra, real and complex analysis, probability theory and more which can be used for future formalizations.

Formal verification is also useful in computer science to prove the correctness of computer software and hardware. In most developments (software or hardware) we do not even have informal proofs of correctness. To certify correctness,

software and hardware are routinely tested for bugs. Nevertheless it is impossible to test exhaustively in most cases and bugs are sometimes not detected with really disastrous consequences. Formal methods can be used to certify the correctness of software and hardware.

1.4 Use of formal methods in cryptography

From the early 90s formal methods have been used to model and analyze the computational security of cryptographic protocols [But99, Mea03]. Formal techniques are employed in several different phases in designing cryptographic protocols such as the specification, construction and verification of protocols.

In abstract cryptographic models, security is provided against adversaries who can only query the algorithm on inputs of their choice and then interpret the outputs which are computed according to the correct secret key. Nevertheless, real life physical implementations do not always correspond to such a model and actual adversaries exploiting physical leakage turn out to be much more powerful. During the last years, there has been significant research on evaluating physical security of cryptographic systems, notably against side–channel attacks, such as power consumption or electromagnetic radiation. Formal analysis of side channel attacks has been proposed to model more general physical leakages [SMY06, BDK13, PR13, MR03]. while recent works [BBC+14, GPP+16] have addressed side-channel resistance (such as timing and memory accesses) for low level cryptographic code.

Provable security [Ste03, GM84] is used to establish the security of cryptographic systems in terms of rigorous mathematical proofs, by reduction: if an adversary is able to compromise the security of the system, then she possesses a way to solve a computationally hard problem. Although, provable security aims to provide strong guarantees of security for cryptographic schemes, the complexity of the proofs is difficult to handle and many proofs have resulted to be flawed. Formal methods modelling game-based proofs, have been successfully introduced to confront this problem and have resulted to many impressive results in this area [BBGO09, BGLB11, BGJB07].

Proofs of correctness In all the above uses of formal methods to verify cryptographic schemes, one assumes that the cryptographic functions are correct. Indeed, correctness of implementations is essential when focusing on cryptographic programs: Apart from the obvious fact that the user of a program needs to be certain that the program does what is supposed to do, any bug or backdoor in an implementation can be catastrophic for security [BBPV11b]. Yet, although the precise computational security of composite constructions and protocols has been widely studied using formal tools [BGHB11, BGZB09, BFK+14], imple-

mentations of the underlying primitives have received far less attention from the formal verification community.

For symmetric primitives such as block ciphers and hash functions, the algorithm **is** the specification. Hence, verifying a block cipher implementation amounts to proving the equivalence between a concrete program written for some platform and an abstract program given in the standard specification. Practitioners commonly believe that a combination of careful code inspection and comprehensive testing is enough to provide high-assurance for such primitives, although a more formal approach can also be used to prove correctness [App15a].

For asymmetric primitives such as RSA encryption, finite field Diffie–Hellman, or elliptic curves, the gap between specification and code can be large. Abstractly, such primitives compute well-defined mathematical functions in some finite field, whereas concretely, their implementations manipulate arrays of bytes that represent arbitrary precision integers. Moreover, asymmetric cryptography is based on a more complex mathematical theory than symmetric cryptography, such as number theory and algebraic geometry in the case of elliptic curves. As a result, asymmetric algorithms designed to exploit certain mathematical properties (of the underlying field for example) are more complicated to understand and to implement correctly. Such an example is the Montgomery reduction algorithm to perform modular reduction in prime fields.

Furthermore, since asymmetric primitives are typically much slower and can form the bottleneck in a cryptographic protocol, most implementations incorporate a range of subtle performance optimizations that further distance the code from the mathematical specification. Further optimizations may take place in order for the implementation to satisfy security criteria such as side channel resistance. Consequently, even for small prime fields, comprehensive testing is ineffective for guaranteeing the correctness of asymmetric primitive implementations, leading to bugs even in well-vetted cryptographic libraries [BBPV12, Ope15]. Even worse, asymmetric primitives are often used with long-term keys, so any bug that leaks underlying key material can be disastrous.

To sum up, besides functional correctness, cryptographic algorithms need to achieve contradictory goals such as efficiency and side–channel resistance. Faulty implementations of algorithms may endanger security [BBPV11a]. This is why formal assurance of their correctness is essential; even more so when it comes to asymmetric primitives which are based on complex mathematical theories.

Our motivation in this thesis is to develop libraries to allow the formal verification of asymmetric cryptographic algorithms, more precisely of elliptic curve algorithms. Until now work on formal verification of security protocols assumed that the cryptographic libraries correctly implement all algorithms [APS12]. The first step towards the formal verification of cryptographic algorithms is the development of libraries that formally express the corresponding mathematical

theory. This thesis presents a formal library for elementary elliptic curve theory that will enable formal analysis of elliptic-curve algorithms. We also present the formalization of the GLV algorithm for scalar multiplication on an elliptic curve group [GLV01].

1.5 Elliptic curves and cryptography

Elliptic curves have been used since the 19th century to approach a wide range of problems, such as the fast factorization of integers and the search for congruent numbers. In the 20th century, researchers have increased interest in elliptic curves because of their applications in cryptography, first suggested in 1985 independently by Neal Koblitz [Kob87] and Victor Miller [Mil85]. Their use in cryptography relies principally on the existence of a group law which makes them a good candidate for public key cryptography, as its Discrete Logarithm Problem is hard relative to the size of the parameters used. Elliptic curves also allow the definition of digital signatures and of new cryptographic primitives, such as identity-based encryption [Sha84], based on bilinear (Weil and Tate) pairings [BF01].

Elliptic curves are used in public key cryptography mainly as an alternative to traditional public-key cryptosystems such as RSA and finite field discrete logarithm based systems. Elliptic curve cryptosystems present an efficiency and security advantage over finite field Diffie–Hellman cryptosystems, known to be slow and vulnerable to the number field sieve attack using precomputation [ABD+15], two limitations that do not apply to elliptic curves, as far as currently known. Indeed, up to now and with the exception of some curves of special form [MOV93], there has not been found a generic attack for elliptic curves over prime fields with a subgroup of large prime order better than the Pollard's rho attack [Pol78] which runs in exponential time. Therefore, when compared to standard finite field Diffie–Hellman or RSA, elliptic curve systems require much shorter keys to achieve the same security level. Because of their efficiency advantage, elliptic curves were widely adopted, especially in cases of constrained devices such as smartcards, cellphones and smartphones and also in web servers for which public key cryptography is a bottleneck. As a result, the use of elliptic curves has been encouraged by several institutions such as the U.S. National Institute of Standards and Technology (NIST) [Nat99], the U.S. National Security Agency (NSA) and l'Agence Nationale de la Securité des Systemes d'Information (ANSSI) proposing sets of recommended elliptic curves and algorithms on top of them. It is indicative that in the report [Nat13] of NSA, Elliptic Curve Diffie–Hellman (ECDH) and the Elliptic Curve Digital Signature (ECDSA) are proposed as primitives suitable for communications requiring a top secret level of security. In the recent years, concerns about mass

surveillance have led to a shift towards the use of elliptic curves in preference to older public–key primitives such as RSA, which no longer provide a sufficient level of security. In that context, elliptic curve cryptosystems are of major significance for many protocols and applications nowadays, and therefore the efficient and secure implementation of elliptic curves schemes is of key importance.

The main operation performed in elliptic curve schemes is scalar multiplication, denoted $[k]P$ in this thesis (where P is a point on an elliptic curve and k is an integer). Several different algorithms are used to speed up scalar multiplication, ranging from generic exponentiation algorithms such as binary exponentiation to curve specific algorithms such as GLV [GLV01] and GLS [GLS09]. Usually, these algorithms are further optimized to achieve better performance by using special curve forms and alternative curve coordinate systems [CMO98, CC86]. Moreover, sometimes further optimizations are used to accelerate the underlying field arithmetic. In practice, implementations have to take security criteria such as side–channel resistance into consideration, so they are even further modified. As a result, implementations of elliptic curve algorithms can be particularly tricky, and in most cases it is not evident that an implementation is correct. This problem can be approached with the use of formal methods, which can provide formal certification that an implementation is correct. In that context, libraries to provide a formal theory for elliptic curves are needed. And this is what we provide.

1.6 Contribution of this thesis

This thesis is in the domain of formalization of mathematics. Our motivation is to formalize elliptic curve theory using the Coq proof assistant, which will enable formal analysis of elliptic-curve schemes and algorithms. For this purpose, we used the Ssreflect extension and the mathematical libraries developed by the Mathematical Components team during the formalization of the Four Color Theorem. Our central result is a formal proof of Picard's theorem for elliptic curves: there exists an isomorphism between the Picard group of divisor classes and the group of points of an elliptic curve. An important immediate consequence of this proposition is the associativity of the elliptic curve group operation. This development has resulted in more than 15000 lines of code and includes formal theory about Weierstrass curves, the field of rational functions on a curve, theory about free groups, divisors of rational functions on curves and isomorphic representations in different coordinate systems. Our results have been published as the article *A Formal Library for Elliptic Curves in the Coq Proof Assistant* at the International Theorem Proving conference 2014.

Furthermore, we present a formal proof of correctness for the GLV algorithm [GLV01] for scalar multiplication on elliptic curve groups. The GLV algorithm

exploits properties of the elliptic curve group in order to accelerate computation. It is composed of three independent algorithms : multiexponentiation on a generic group, decomposition of the scalar and computing endomorphisms on algebraic curves. This development includes theory about endomorphisms on elliptic curves and is more than 5000 lines of code.

An application of our formalization is presented in Chapter 5. This work consists of formally proving real-life implementations of elliptic curve algorithms combining our development in Coq and F*, which is a new higher order programming language designed for program verification.

The entire development presented in this thesis is available at https://github.com/strub/glv.

2
Background

2.1 Mathematical background

2.1.1 Elliptic curves definitions

Definition 2.1 (Projective plane)**.** *The projective plane* $\mathbb{P}^2$ *over* $\mathbb{F}$ *is the quotient* $\mathbb{P}^2 = {}^{(\mathbb{F}^3 \setminus (0,0,0))}/_{\sim}$ *where* $(x, y, z) \sim (x', y', z')$ *if and only if there exists a* $\lambda \in \mathbb{F}^*$ *such that* $(x', y', z') = (\lambda x, \lambda y, \lambda z)$*.*

Definition 2.2 (Curve of the projective plane)**.** *A curve of the projective plane is the set of projective points* $(x : y : z)$ *whose coordinates are a solution of a homogeneous equation* $f(x, y, z) = 0$*.*

A curve represented by the homogeneous equation $f(x, y, z) = 0$ is *smooth* or *non-singular* if the partial derivatives of f with respect to x, y, z do not all vanish simultaneously on the curve. If a curve is *smooth*, then there are no singular points, i.e. no cusps or nodes (self-intersections).

The definition of an elliptic curve in full generality is the following:

Definition 2.3 (Elliptic Curve)**.** *An elliptic curve* $\mathcal{E}$ *over some field* $\mathbb{F}$ *is a smooth projective plane curve over* $\mathbb{F}$ *of the form*

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

with the $a_i \in \mathbb{F}$*. This form is called a Generalized Weierstrass form.*

There exists a one-to-one correspondence between the projective plane $\mathbb{P}^2$ and the union of an affine plane $\mathbb{F}^2$ and the projective line at infinity. (To be more precise, there exists an isomorphism of algebraic varieties between the projective plane and the above union). Indeed, let $v = (x, y, z)$ be a non zero vector of $\mathbb{F}^3$. If $z \neq 0$ then the equivalence class of v (denoted here $[v]$) is $[v] = [z^{-1}v] = (\frac{x}{z} : \frac{y}{z} : 1)$ and there exists a unique representative of the class of the form $(x', y', 1)$. Hence, there is a 1-to-1 map between the set of projective points with $z \neq 0$ and $\mathbb{F}^2$. If $z = 0$, then x and y cannot both be zero because v is a non zero vector. Moreover, if $x \neq 0$ then $[v] = [x^{-1}v] = (\frac{x}{x} : \frac{y}{x} : 0)$ and so there exists a unique representative of the class of the form $(1, y', 0)$. So, there exists a 1-to-1 map between the set of projective points with $z, x \neq 0$ and $\mathbb{F}$. If $z = 0$ and $x = 0$ then $[v] = [y^{-1}v] = [0, \frac{y}{y}, 0] = (0 : 1 : 0)$. Hence, $\mathbb{P}^2 \cong \mathbb{F}^2 \cup \mathbb{F} \cup \{(0 : 1 : 0)\}$.

In this setting, an elliptic curve of equation $Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$ in projective coordinates is isomorphic to the curve of equation $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ of the affine plane, together with a separate point $\mathcal{O}$ called the point at infinity. A projective point of the elliptic curve with $z = 0$ is of the form $(0 : y : 0)$, and since a projective point is an equivalence class, we can choose the representative $(0, 1, 0)$ for this class. This is the point at infinity $(0 : 1 : 0)$ in projective coordinates, which plays an important role when moving to the definition of the group law. This part is explained in details in Chapter 3.

If the characteristic of the field $\mathbb{F}$ is not 2 or 3, then by an appropriate change of variables [Sil09], an elliptic curve can be written in the form $y^2z = x^3 + axz^2 + bz^3$, where a and b are elements of $\mathbb{F}$. The condition that the curve is smooth reduces to $\Delta = 4a^3 + 27b^2 \neq 0$.

Most of the time, when introducing elliptic curves in a cryptographic context, we use its short Weierstrass form:

Definition 2.4 (Elliptic Curve Short Weierstrass Form)**.** *An elliptic curve $\mathcal{E}$ over some field $\mathbb{F}$ with characteristic different from 2 and 3 is an affine curve of equation $y^2 = x^3 + ax + b$ together with a separate point $\mathcal{O}$ called the point at infinity, where $a, b \in \mathbb{F}$ satisfy $\Delta = 4a^3 + 27b^2 \neq 0$.*

In standard elliptic curve implementations we often use projective coordinate systems when performing cryptographic operations in order to improve performance and avoid expensive field inversions.

The below remarks are not necessary to understand the mathematics of our development and the reader may skip them, but they provide interesting context to elliptic curve theory.

Remark 1 : Isomorphisms of algebraic curves

The notion of *isomorphism* of curves is much stronger than bijection of sets of points, and reveals more about the relationship between two curves. One of the reasons that isomorphisms are more interesting is that they are defined by polynomials (or rational functions), so they're something that one might actually compute while bijections are much looser. In particular, every isomorphism induces a bijection of sets of points, but the converse does not hold.

For example, if we take two 256-bit primes p and q that are very close together, then their Hasse intervals $(p+1-2\sqrt{p}, p+1+2\sqrt{p})$ and $(q+1-2\sqrt{q}, q+1+2\sqrt{q})$ intersect, and we can hope to find a prime r in the intersection. Then there must exist (by Deuring's theorem) [Sch87] a curve E_p over $\mathbb{F}_p$ and a curve E_q over $\mathbb{F}_q$ such that $r = \#E_p(\mathbb{F}_p) = \#E_q(\mathbb{F}_q)$. These groups have the same order, so there exists a bijection between them. But since the curves are defined over different fields, there is simply no way that we can efficiently realize that bijection as a polynomial mapping. In general, we cannot compute such a bijection without solving discrete logarithms which is cryptographically hard.

Remark 2 : The Projective space

The construction of the projective plane is a special case of a projective space:

Definition 2.5 (Projective space)**.** *Let $\mathbb{F}$ be a field. The* Projective n-space *over $\mathbb{F}$, denoted $\mathbb{P}^n$, is the set of all lines through $(0, 0, ..., 0)$ in $\mathbb{F}^{n+1}$.*

Two non zero points $A = (a_1, a_2, ..., a_{n+1})$, $B = (b_1, b_2, ..., b_{n+1})$ determine the same line if there exists a $\lambda \in \mathbb{F}^*$ such that $\forall i, a_i = \lambda b_i$. Hence, we can alternatively define the projective space $\mathbb{P}^n$ as the quotient set $\mathbb{P}^n = {}^{(\mathbb{F}^{n+1}\setminus(0,\ldots,0))}/_{\sim}$, where $(a_1, a_2, \ldots, a_{n+1}) \sim (b_1, b_2, \ldots, b_{n+1})$ if and only if there exists a $\lambda \in \mathbb{F}^*$ such that $a_i = \lambda b_i, \forall i$. Elements of $\mathbb{P}^n$ are equivalence classes of the quotient set (i.e. lines from a geometric point of view), and are called points of the projective space. If $(x_1, x_2, ..., x_{n+1})$ is a representative of a certain class of $\mathbb{P}^n$ we say that $(x_1, x_2, ..., x_{n+1})$ is a set of homogeneous coordinates for this class and we denote the projective point $(x_1 : x_2 : ... : x_{n+1})$. It can be shown that there is an isomorphism between a projective n-space $\mathbb{P}^n$ and the union of an affine n-space and a hyperplane at infinity. More precisely, $\mathbb{P}^n \cong \mathbb{F}^n \cup H_\infty$, where $H_\infty = \{(x_1 : x_2 : ..., x_n : x_{n+1}) \mid x_{n+1} = 0\}$, i.e. $H_\infty \cong \mathbb{P}^{n-1}$. For more details, see [Ful89].

Remark 3 : The genus of a curve

The *genus* of a curve can be intuitively understood as a measure of the curve's geometric complexity.

One reasonable classification of algebraic curves could be according to the degree of the curve (i.e. the degree of the polynomial that defines the curve

equation), but unfortunately it does not work for curves of higher degree. For example, let us consider the non-singular curves $L : y = 0$ and $C : yz - x^2 = 0$. The curves L and C are isomorphic: There exists a mapping $m : (x : y : z) \mapsto (x : 0 : z)$ from C to L and a mapping $n : (x : y : z) \mapsto (xz : x^2 : z^2)$ from L to C such that $(m \circ n)(x : y : z) = (x : y : z)$ for all $(x : y : z) \in L$ and $(n \circ m)(x : y : z) = (x : y : z)$ for all $(x : y : z) \in C$. One would want the measure that characterizes the geometric complexity of algebraic curves to be invariant under isomorphism. Nevertheless, in this case L is of degree 1 and C is of degree 2.

The *genus* is a non-negative integer that can be associated to any algebraic curve and characterizes the geometric complexity of the curve. For example:

— the curve $C_1 : y = x$ has genus 0,

— the curve $C_2 : y^2 = x^2 + Bz^2$ has genus 0,

— the curve $C_3 : y^2z = x^3 + Axz^2 + Bz^3$ has genus 1,

— the curve $C_4 : y^2z^2 = x^4$ has genus 1, and

— the curve $C_5 : y^2z^3 = x^5 + xz^4$ has genus 2.

To compute the genus of a non-singular curve, we can use the formula $\frac{(d-1)(d-2)}{2}$, where d is the degree of the curve. For singular curves, which is out of scope for this thesis, see [Cas91].

A smooth projective plane curve of genus 1 is a cubic defined by a homogeneous polynomial of the form

$$f(x, y, z) = Ax^3+By^3+Cz^3+Dx^2y+ Ex^2z+Fy^2x+Gy^2z+Hz^2x+Iz^2y+Jxyz.$$

This is a direct consequence of the Riemann–Roch theorem which will be explained in the end of this section. An equivalent definition of an elliptic curve is the following:

Definition 2.6. *An elliptic curve $\mathcal{E}$ over some field $\mathbb{F}$ is defined as a smooth projective plane curve of genus 1 together with a point $\mathcal{O} \in \mathcal{E}(\mathbb{F})$.*

Remark 4 : Abelian Varieties and Elliptic Curves

An alternative definition of an elliptic curve, in a more algebraic context, is that an elliptic curve is an *abelian variety of dimension one*. To give some intuition, a variety is the zeros of a set of polynomials subject to some irreducibility conditions. For a precise definition and further details, see [Ful89]. A variety of dimension one is a curve. An abelian variety is a smooth projective variety where one can define a group operation by ratio of polynomials. This operation makes the variety a commutative group. In this sense, an abelian variety of dimension one is a smooth projective curve equipped with a group operation defined by polynomial fractions, and it can be shown that every such curve is an elliptic curve.

2.1.2 Defining addition

From this point on, we consider fields with characteristic different from 2, 3 and so we are free to interchange between the projective and the short Weierstrass form of elliptic curves.

To understand the group law, one first needs to understand Bézout's theorem for elliptic curves:

Lemma 2.1 (Bézout for elliptic curves)**.** *Let $\mathbb{F}$ be an algebraically closed field, and $f \in \mathbb{F}[X, Y, Z]$ a homogeneous polynomial of degree 3. Let $\mathcal{E}$ be the elliptic curve defined by the equation $\mathcal{E} : Y^Z = X^3 + aXZ^2 + bZ^3$ and L a line (not contained in $\mathcal{E}$). Then $L \cap \mathcal{E}$ has exactly 3 points counted with multiplicity.*

More precisely, a line in the projective space is the set of projective points $(X : Y : Z)$ which are the solutions of the equation $kX + lY + mZ = 0$ for some $k, l, m \in \mathbb{F}$ (k, l, m not all zero). Given an elliptic curve $\mathcal{E}$ of equation $\mathcal{E} : Y^Z = X^3 + aXZ^2 + bZ^3$, there exist three different kinds of lines:

1. the line of equation $Z = 0$, which intersects the elliptic curve $\mathcal{E}$ at the point at infinity $(0 : 1 : 0)$ with multiplicity 3,
2. the line of equation $X + cZ = 0$ (with $c \neq 0$) which intersects the elliptic curve $\mathcal{E}$ at the point at infinity and at the points $(-c : \sqrt{(-c)^3 - ac + b} : 1)$, $(-c : -\sqrt{(-c)^3 - ac + b} : 1)$ which may coincide, and
3. the line of equation $kX + lY + lZ = 0$, (with $k, l, m \neq 0$) which intersects the elliptic curve $\mathcal{E}$ at three finite points of the form $(x : y : 1)$, which may coincide. To compute the x, y we have to solve the system

$$\begin{cases} y^2 = x^3 + ax + b \\ kx + ly + m = 0. \end{cases}$$

Bézout's theorem allows us to geometrically define an operation on elliptic curve points. Let P and Q be points on an elliptic curve $\mathcal{E}$, and l be the line through P and Q (or the tangent to the curve at P if $P = Q$). By the Bézout theorem, l intersects $\mathcal{E}$ at a third point, denoted by $P \boxdot Q$. The sum $P + Q$ is the opposite of $P \boxdot Q$, obtained by taking the symmetric of $P \boxdot Q$ with respect to the x axis (in affine coordinates) or the point $(x : -y : z)$ (in projective coordinates).

Addition in affine coordinates.

1. $\mathcal{O}$ is defined to be the neutral element: $\forall P, P + \mathcal{O} = \mathcal{O} + P = P$.
2. the negative of a point (x_P, y_P) (resp. $\mathcal{O}$) is $(x_P, -y_P)$ (resp. $\mathcal{O}$), and
3. if three points are collinear, their sum is equal to $\mathcal{O}$.

This geometrical definition can be translated into an algebraic setting, obtaining the following polynomial formulas: Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be two finite points. Then:

1. if $P \neq Q$, then $P + Q = (x_S, y_S)$ with:

$$\begin{cases} x_S = \lambda^2 - x_P - x_Q \\ y_S = -\lambda^3 + 2\lambda x_P - \lambda x_Q - y_Q \end{cases} \quad \text{where } \lambda = \frac{y_P - y_Q}{x_P - x_Q},$$

2. if $P = Q$ with $y_P = y_Q \neq 0$, then will apply the previous formulas with $\lambda = {}^{(3x_P^2+a)}/_{2y_P}$,
3. if $P = Q$ with $y_P = y_Q = 0$, then $P + Q = \mathcal{O}$.

Addition in projective coordinates. In projective coordinates, addition is defined by the following rules:

1. The zero element is the point at infinity $(0 : 1 : 0)$.
2. The negative of a projective point $(x : y : z)$ is the point $(x : -y : z)$.
3. If three points are collinear, their sum is equal to $\mathcal{O}$.

Like in the affine setting, these rules can be translated into polynomial formulas: For all $P = (x_P : y_P : z_P)$ on E and $Q = (x_Q : y_Q : z_Q)$ on E, let $S = P + Q = (x_S : y_S : z_S)$ be the sum of the two points.

— If $P = Q$ then

$$\begin{cases} u = 3x_P^2 + az_P^2 & x_S = vr \\ v = 2y_P^2 & y_S = -u(r - v^2x) - y_Pv^3 \\ r = u^2z_P - 2v^2x_P & z_S = z_Pv^3. \end{cases}$$

— If $P \neq Q$ then

$$\begin{cases} u = y_Qz_P - y_Pz_Q & x_S = vr \\ v = x_Qz_P - x_Pz_Q & y_S = -u(r - x_Pz_Qv^2) - y_Pz_Qv^3 \\ r = u^2z_Pz_Q - v^2(x_Pz_Q + x_Qz_P) & z_S = z_Pz_Qv^3. \end{cases}$$

Remark 4: Bézout generalized for algebraic curves

Bézout theorem can be generalized for algebraic curves. Let C, D be two plane projective curves given by the equations $F(x, y, z) = 0$ and $G(x, y, z) = 0$ respectively. We say that the curves *share a common component* if F, G have a non constant common divisor. Any projective point $(x_0 : y_0 : z_0)$ satisfying $F(x_0, y_0, z_0) = 0$ and $G(x_0, y_0, z_0) = 0$ is defined as an *intersection point* of C and D. Any two algebraic curves without common component intersect in finitely many points. Let P be an intersection point of C and D. Then we can

define an intersection multiplicity at P, denoted here as $v_P(C, D)$. We are not going to give here the detailed definition of this intersection multiplicity, for more details see [Sut15].

Lemma 2.2 (Bézout)**.** *Let C, D be two plane projective curves over an algebraically closed field $\mathbb{F}$ of degree m and n respectively. Suppose that C and D share no common components. Let S be the set of intersection points $C(\mathbb{F}) \cap D(\mathbb{F})$. Then $\sum_{P \in S} v_P(C, D) = mn$.*

2.1.3 Riemann Roch and the group law

The set of points of an elliptic curve together with the operation defined above form an abelian group: the operation is commutative and associative, there is a neutral element (the point at infinity), and there exists a negative point for every point on the curve. In both a geometrical or an algebraic setting, it is direct to prove that the operation is commutative, but it is not trivial to prove that it is associative. In what follows we sketch three different proofs of associativity:

— a geometrical (grid) proof,

— a computational proof, and

— a proof based on the Riemann Roch theorem.

The first two proofs are interesting as context for elliptic curves, but they are not necessary to understand as a background to our formalization. The third proof, which is based on the Riemann–Roch theorem, allows to introduce several notions that reappear in the formalization presented in Chapter 3. The proof of associativity that we have formalized is based on the idea underlying the Riemann–Roch proof. However, the formalization of the Riemann–Roch theorem was out of the scope of this thesis. As a result, the parts of the proof which are direct consequences of the Riemann–Roch theorem are formalized in a more elementary way. This is explained in details in Chapter 3.

The geometrical grid proof

Let us recall the Bézout theorem as stated in [HIS14]:

Theorem 2.3 (Bézout)**.** *Let X and Y be two plane projective curves defined over a field $\mathbb{F}$ that do not have a common component (i.e. X and Y are defined by polynomials whose greatest common divisor is a constant). Then the total number of intersection points of X and Y with coordinates in an algebraically closed field $\mathbb{E}$ which contains $\mathbb{F}$, counted with multiplicity, is equal to the product of the degrees of X and Y .*

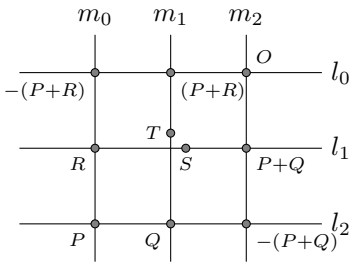


Figure 2.1 – The geometrical grid proof

The following lemma is a direct consequence of the Bézout theorem for cubics and allows to prove that the addition on elliptic curve points, as defined above, is a group law.

Lemma 2.4. *In a projective plane, let A_{ij} be the intersection points of the straight lines p_i and q_j where $1 \leq i,\ j \leq 3$, and the points A_{ij} are pairwise distinct. Suppose that all points A_{ij}, except perhaps A_{33} lie on a cubic. Then A_{33} also lies on this cubic.*

The sketch of the proof given here is partly from [Sut15] which is an adaptation of the proof initially given in [Cas91] and partly from [HIS14]. It only concerns the general case (meaning that in the diagram in Figure 2.1, the points are related only by the way the diagram is constructed and in no other way).

Let P, Q, R be three distinct non-zero points of an elliptic curve $\mathcal{E}$ over some field $\mathbb{F}$, that we consider algebraically closed.

- Let l_0 the line through P and Q. The third point of intersection with $\mathcal{E}$ is the point $-(P+Q)$.
- Let m_0 the line through P and R. The third point of intersection with $\mathcal{E}$ is the point $-(P+R)$.
- Let m_2 the line through $-(P+Q)$ and $P+Q$. The third point of intersection with $\mathcal{E}$ is the point $\mathcal{O}$.
- Let l_2 the line through $-(P+R)$ and $P+R$. The third point of intersection with $\mathcal{E}$ is the point $\mathcal{O}$.
- Let m_1 the line through Q and $P+R$. The third point of intersection with $\mathcal{E}$ is the point $S = -(Q+(P+R))$.
- Let l_1 the line through R and $P+Q$. The third point of intersection with $\mathcal{E}$ is the point $T = -(R+(P+Q))$.

It suffices to show that $S = T$. By the aforementioned Bézout theorem, the intersection point of m_1 and l_1 is on the curve $\mathcal{E}$. As a result, l_1 intersects the curve at Q, $P+R$ and A. Hence, $A = S$. Following the same reasoning for m_1, $A = T$ and therefore $T = S$ and the addition is associative.

The computational proof

Using the explicit Weierstrass formulas, which define addition in affine coordinates, we can give a more elementary proof of associativity, fully presented in [Fri98]. Despite what one might have thought, this direct proof is not trivial because it involves many special cases that have to be treated separately. Moreover, some of the explicit computations involved are hard and the verification took several hours on a computer. The proof presented in [Fri98] treats separately the following different cases:

— $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathcal{E} \setminus \{\mathcal{O}\}$. If $A \neq \pm B$, $B \neq \pm C$, $A + B \neq \pm C$ and $B + C \neq \pm A$, then $(A + B) + C \neq A + (B + C)$.

— $\mathbf{A}, \mathbf{B} \in \mathcal{E} \setminus \{\mathcal{O}\}$. If $A \neq -A$, $A \neq \pm B$, $A + A \neq \pm B$ and $A + B \neq \pm A$, then $(A + A) + B \neq A + (A + B)$.

— $\mathbf{A} \in \mathcal{E} \setminus \{\mathcal{O}\}$. If $A \neq -A$, $A + A \neq -(A + A)$, $(A + A) + A \neq \pm A$ and $A + A \neq \pm A$, then $(A + A) + (A + A) \neq A + (A + (A + A))$.

— After having proven several basic properties, such as the uniqueness of the neutral element and the cancellation rule, one can prove the following fact. Assume that

1. $A + B \neq C$ and $A \neq B + C$ or
2. $A = B$ or $B = C$ or $A = C$ or
3. $\mathcal{O} \in \{A, B, C, A + B, B + C, (A + B) + C, A + (B + C)\}$,

then $(A + B) + C = A + (B + C)$.

This is the most elementary and technical approach to proving associativity.

Théry et al [Thé07] present a formal proof that an elliptic curve is a group using the Coq proof assistant, similar to the one described here. The proof that the operation is associative relies heavily on case analysis and uses computer-algebra systems to deal with non-trivial computation.

The Riemann Roch theorem

Following Dummit's lecture notes *A Whirlwind Tour of Elliptic Curves* at NTS in 2012, we state the Riemann–Roch theorem for algebraic curves and we sketch the proof of associativity for the elliptic curve group law.

The Riemann–Roch theorem is a famous theorem of algebraic geometry giving the dimension of the space of functions on a curve, given conditions on their zeros and poles. To state Riemann–Roch, we first need to introduce a few notions: Let C be a smooth curve over a field $\mathbb{K}$.

1. The free abelian group generated by the points of C is called the *divisor group* and is denoted $\mathrm{Div}(C)$. A *divisor* D is an element of the form $D = \sum_{P \in C} n_P(P)$ with $n_P \in \mathbb{Z}$, all but finitely many of which are zero.

The *degree* of a divisor $D = \sum_{P \in C} n_P(P)$ is defined to be the sum of the coefficients $\deg(D) = \sum_{P \in C} n_P$. The subgroup of degree-zero divisors $\mathrm{Div}^0(C)$ is the kernel of the degree map.

2. Let f be a non-zero function on C. Let the divisor of f to be $\mathrm{div}(f) = \sum_{P \in C} \mathrm{ord}_P(f)(P)$ where $\mathrm{ord}_P(f)$ is the order of vanishing of f at P (if f has a zero at P, the order is the degree of the zero, and if f has a pole at P, the order is the opposite of the degree of the pole). This is a degree-zero divisor because functions on smooth projective curves have equal numbers of zeros and poles.
3. A divisor is *principal* if it is the divisor of some nonzero function. The principal divisors form a subgroup of $\mathrm{Div}^0(C)$ denoted $\mathrm{Prin}(C)$, and therefore we can quotient $\mathrm{Div}(C)$ by $\mathrm{Prin}(C)$ and form the group of divisors modulo principal divisors. This quotient group is called the *Picard group* and is denoted $\mathrm{Pic}(C)$. Two divisors are equivalent if they belong to the same class of the Picard group.
4. Let w be a nonzero differential form on C. We do not give a definition of differential forms here, for details see [Ful89]. We can associate a divisor to w by computing its zeros and the corresponding orders of vanishing. All nonzero differentials are in the same class of the Picard group, called the *canonical divisor* K_C.
5. A divisor D is called *effective*, denoted by $D \geq 0$, if all coefficients of D are greater than or equal to zero. This is extended to a partial order on $\mathrm{Div}(C)$ as follows:

$$\forall P, \sum_{P \in C} n_P(P) \geq \sum_{P \in C} m_P(P) \iff n_P \geq m_P.$$

6. For a divisor D we define $L(D)$ to be the finite-dimension vector space

$$L(D) = \{0\} \cup \{f \in \bar{\mathbb{K}}(C) \mid div(f) \geq -D\}.$$

We denote the dimension of $L(D)$ by $l(D)$.

Theorem 2.5 (Riemann–Roch)**.** *Let C be a smooth curve of genus g and K_C the canonical divisor of C. Let D be any divisor in $Div(C)$. Then $l(D) - l(K_C - D) = deg(D) - g + 1$.*

An elliptic curve is a smooth curve of genus 1, so in the case of an elliptic curve $\mathcal{E}$, the Riemann–Roch theorem gives $l(D) - l(K_C - D) = \deg(D)$ for any divisor $D \in \mathrm{Div}(\mathcal{E})$.

To prove that the elliptic curve law is associative, one has to prove that there exist an isomorphism between $\mathrm{Pic}^0(\mathcal{E})$ and $\mathcal{E}$: note that $\mathrm{Pic}^0(\mathcal{E})$, which is the Picard group of zero-degree divisors, is an abelian group (as it is a quotient of

abelian groups). Because of the Riemann–Roch theorem, every class of $\mathrm{Pic}^0(\mathcal{E})$ has a unique representative of the form $[(P) - (\mathcal{O})]$ where $P \in C$. Thus, there exists a bijection between $\mathrm{Pic}^0(\mathcal{E})$ and $\mathcal{E}$, which is also a morphism. By transport of structure, $\mathcal{E}$ is also an abelian group. The fact that every class of $\mathrm{Pic}^0(\mathcal{E})$ has a unique representative of the form $[(P) - (\mathcal{O})]$ is a direct consequence of the Riemann Roch theorem:

Existence Recall that an elliptic curve is a smooth curve of genus 1. From Riemann–Roch, we have that $l(D + (\mathcal{O})) = 1$: Remark that, in the case of elliptic curves, $g = 1 \implies 2g - 2 = 0$, so any divisor D with degree greater than zero satisfies $l(K_{\mathcal{E}} - D) = 0$.

Given a zero degree divisor D, $\deg(D + (\mathcal{O})) = 1 > 0 \implies l(K_{\mathcal{E}} - D - (\mathcal{O})) = 0$. Hence, $l(D + (\mathcal{O})) - l(K_{\mathcal{E}} - D - (\mathcal{O})) = \deg(D + (\mathcal{O})) \implies l(D + (\mathcal{O})) = 1$. Given a generator f of $L(D + (\mathcal{O}))$, we have $\mathrm{div}(f) > -D - (\mathcal{O})$ and $\deg(\mathrm{div}(f)) = 0$. So there exists a point $P \in \mathcal{E}$ such that $\mathrm{div}(f) = -D - (\mathcal{O}) + (P)$. As a result, $D \sim (P) - \mathcal{O}$.

Uniqueness To prove that the representative is unique, we have to prove that if $(P) \sim (Q)$ then $P = Q$: From Riemann–Roch, we have $l((Q)) = 1$. As above, $l((Q)) - l(K_{\mathcal{E}} - (Q)) = \deg((Q)) = 1 \implies l(K_{\mathcal{E}} - (Q)) = 0$. Now, $L((Q))$ contains all the constant functions (whose divisors are the zero divisor). So if we consider a generator g of $L((Q))$, then g must be a constant function (if not, then all constant functions are not included in $L((Q))$). If $(P) \sim (Q)$ then there exists a function $f \in \mathbb{K}(\mathcal{E})$ such that $\mathrm{div}(f) = (P) - (Q)$ and $f \in L((Q))$. By the above, f is constant and so $\mathrm{div}(f) = (P) - (Q) = 0$ which implies that $P = Q$.

Remark 5 : Our formal proof of associativity

We have formalized the above proof, with the exception of the fact that every class of $\mathrm{Pic}^0(C)$ has a unique representative of the form $[(P) - (\mathcal{O})]$. In the above description, this is a consequence of the Riemann–Roch theorem. In our formalization, the proof follows a more elementary path. This is because formalizing the Riemann–Roch theorem was out of reach in this work.

Remark 6 : Riemann–Roch and Weierstrass (cubic) forms

The Riemann–Roch theorem is very important for the theory of elliptic curves. The following proposition is a direct consequence of the Riemann-Roch theorem:

Lemma 2.6. *A plane projective curve C of genus 1 is a cubic given by the equation $E : y^2 + a_1xy + a_3y = x^3 + a_2x + a_4x + a_6$.*

Proof. Indeed, a property of the Riemann Roch theorem is that if $\deg(D) > 2g-2$ then $l(D) = \deg(D) - g + 1$. Let C be a curve of genus 1 over an algebraically

closed field $\mathbb{K}$. Let P be a point of C. Then, by the above property, $l(n(P)) = n$ for all $n = 1, 2, \ldots$, since $\deg(n(P)) = n > 2g - 2 = 0$ for all n.

— $l((P)) = 1$ and hence, $L((P)) = \mathbb{K}$ since it includes all the constant functions. Therefore $L((P)) = \mathbb{K}$ has base $\{1\}$.

— $l(2(P)) = 2$ and so a base for $L(2(P))$ will be $\{1, x\}$ where $\mathrm{ord}_P(x) = -2$ and $\mathrm{ord}_Q(x) \geq 0$ for all other points Q in C.

— $l(3(P)) = 3$ and so a base for $L(3(P))$ will be $\{1, x, y\}$ where $\mathrm{ord}_P(y) = -3$ and $\mathrm{ord}_Q(y) \geq 0$ for all other points Q in C.

— In the same way, $L(4(P))$ has base $\{1, x, y, x^2\}$ and $L(5(P))$ has base $\{1, x, y, x^2, xy\}$.

— In $L(6(P))$ the set $\{1, x, y, x^2, xy, x^3, y^2\}$ is linearly dependent (7 functions included). Indeed, both x^3 and y^2 have order -6 at P and non-negative order at all other points of C. In the linear relation, the coefficients of x^3 and y^2 cannot be zero, because if not every other term would have a different order at P. As a result, for appropriate constants, the linear dependence can be written in the form $y^2 + a_1xy + a_3y = x^3 + a_2x + a_4x + a_6$. □

2.2 Use of elliptic curves in cryptography

Elliptic curves are used in public key cryptography mainly as an alternative to traditional public-key cryptosystems such as RSA and finite field discrete logarithm based systems. Their use was proposed in 1985 independently by Miller [Mil85] and Koblitz [Kob87] and while their acceptance was not immediate, they were widely adopted in the 21st century. Elliptic curve cryptosystems present an efficiency and security advantage over finite field cryptosystems, known to be slow and vulnerable to number field sieve attacks with precomputation [ABD+15], two limitations that do not apply to elliptic curves, as far as currently known. Indeed, up to now and with the exception of some curves of special form [MOV93], there has not been found a generic attack for elliptic curves with a subgroup of large prime order, better than the Pollard's rho attack [Pol78] (which runs in exponential time). Therefore, when compared to standard finite field Diffie–Hellman or RSA, elliptic curve systems require much shorter keys to achieve the same security level.

Moreover, recent trends in protocol design indicate a shift towards the use of elliptic curves in preference to older asymmetric primitives. This is partly due to concerns about mass surveillance, which means that non-forward-secret primitives such as RSA encryption are no longer considered sufficient. Cryptographic libraries such as OpenSSL already implement dozens of standardized elliptic curves. However, concerns about backdoors in NIST standards [CNE+14] have led to the standardization of new elliptic curves such as Curve25519 and Curve448 [LH16],

and implementations of these relatively new curves are currently being developed and widely deployed. Verifying these fresh implementations of new elliptic curves was given as an open challenge from practitioners to academics at the Real World Cryptography workshop in 2015 [Kas15].

Elliptic Curve Cryptographic Schemes

Let $\mathcal{E}$ be an elliptic curve over some prime field $\mathbb{F}_p$. Let $P \in \mathcal{E}(\mathbb{F}_p)$ be a point of order r. Given an integer k in the range $[1, r-1]$ we call *scalar multiplication* by k, the sum of k copies of P:

$$[k]P = P + P + \cdots + P.$$

Evidently, the result $[k]P$ is a an element of the cyclic subgroup generated by P.

Some basic cryptographic schemes using elliptic curves are presented below. The description of these are taken from [Lon11]. In what follows, P is a point of $\mathcal{E}$ which generates the cyclic group $\langle P \rangle$ of prime order r. In the following schemes, the parameters $\mathcal{E}, P, p, r$ are public and everyone has access to them.

Elliptic Curve Diffie–Hellman key exchange

The elliptic curve Diffie–Hellman key exchange is a variant of the original Diffie–Hellman scheme [DH76]. Two correspondants Alice and Bob are trying to establish a shared key after exchanging some messages on a public channel. Alice chooses her secret key, which is an integer a in $[1, ..., r-1]$ and computes the associated multiple $Q_a = [a]P$. Bob too chooses his secret key, which is an integer b in $[1, ..., r-1]$ and computes the associated multiple $Q_b = [b]P$. Then they exchange the values Q_a and Q_b. Alice then uses her secret key to compute $[a]Q_b$, and Bob similarly computes $[b]Q_a$. They have both computed the shared key $K = [ab]P$. The ECDH scheme is depicted in Algorithm 1.

Algorithm 1: Elliptic Curve Diffie-Hellman key exchange (ECDH)

Input: $\mathcal{E}$, p, P, r
Alice: Choose a random integer $a \in [1, r-1]$.
Alice: Compute $Q_a = [a]P$ and send it to Bob.
Bob: Choose a random integer $b \in [1, r-1]$.
Bob: Compute $Q_b = [b]P$ and send it to Bob.
Alice: Upon reception of Q_b, compute $K = [a]Q_b$.
Bob: Upon reception of Q_a, compute $K = [b]Q_a$.
Output: shared key $Q = [ab]P$

Elliptic Curve ElGamal

The elliptic curve ElGamal scheme is an analog of the standard ElGamal crypto-scheme [Gam84]. Alice chooses a private key which is an integer k, and then publishes her public key $Q = [k]P$. Assume that Bob wants to send Alice a message m. First, he converts the message m into a point $M \in \mathcal{E}(\mathbb{F}_p)$. Then he chooses his ephemeral key to be a random integer d and he computes the two points $C_0 = [d]P$ and $C_1 = M + [d]Q$. He sends the two points (C_0, C_1) to Alice. To decrypt the message, Alice computes the point $M = C_1 - [k]C_0$ using her secret key k. She then converts the point M to the plaintext m. The elliptic curve ElGamal scheme is depicted in Algorithm 2 and 3.

Algorithm 2: Elliptic Curve El Gamal Encryption

Input: $\mathcal{E}$, p, P, r, public key Q and plaintext m.
Encode m as a point M in $\mathcal{E}(\mathbb{F}_p)$.
Choose a random integer $d \in [1, r-1]$.
Compute $C_0 = [d]P$.
Compute $C_1 = M + [d]Q$.
Return (C_0, C_1).
Output: ciphertext (C_0, C_1)

Algorithm 3: ElGamal Decryption

Input: $\mathcal{E}$, p, P, r, private key k and ciphertext (C_0, C_1)
Compute $M = C_1 - [k]C_0$.
Decode the point M to the plaintext m.
Return m
Output: plaintext m

In order to perform elliptic curve ElGamal, one needs to encode plaintext messages as points, which is not straightforward. Nevertheless, there have been several solutions to approach this problem [Ica09]. We do not discuss this here.

Elliptic curve digital signature algorithm (ECDSA)

The elliptic curve digital signature algorithm is the analogue of the Digital Signature Algorithm (DSA). In what follows, H denotes a hash function that is assumed to be collision resistant.

Suppose Alice wants to send a signed message m to Bob. Suppose also that Alice possesses a secret private key k. Then, she chooses a random integer d and computes the point $[d]P = (x_1, y_1)$. Setting $z = x_1$, she computes $s_0 = z$

$\pmod r$ and $s_1 = d^{-1}(H(m) + kz) \pmod r$, where $s_0 \neq 0$. Then, she sends the encrypted message to Bob together with the signature (s_0, s_1).

To verify the signature, Bob has first to recover the plaintext m. He first computes the hash $H(m)$ and $t = s_1^{-1} \pmod r$. Afterwards, he computes the integers $u = e^t \pmod r$ and $v = s_0 t \pmod r$ and the point $T = [u]P + [v]Q = (x_1, x_2)$. Setting $z = x_1$, if $s_0 = z \pmod r$, then the signature is verified. If not, or if $T = \mathcal{O}$, then the signature is rejected.

The ECDSA scheme is illustrated in Algorithms 4 and 5.

Algorithm 4: ECDSA signature generation

1 *Input*: $\mathcal{E}$, p, P, r, private key k and message m
2 Choose random integer $d \in [1, r-1]$.
3 Compute $[d]P = (x_1, y_1)$ and set $z = x_1$.
4 Compute $s_0 = z \pmod r$. If $s_0 = 0$ go to step 2.
5 Compute $e = H(m)$.
6 Compute $s_1 = d^{-1}(e + kz) \pmod r$. If $s_0 = 0$ go to step 2.
7 Return (s_0, s_1).
8 *Output*: Signature (s_0, s_1).

Algorithm 5: ECDSA signature verification

1 *Input:* $\mathcal{E}$, p, P, r, public key Q, message m and signature (s_0, s_1)
2 If $s_0, \notin [1, r-1]$ or if $s_1, \notin [1, r-1]$, then return (reject the signature)
3 Compute $e = H(m)$
4 Compute $t = s_1^{-1} \pmod r$.
5 Compute $u = et \pmod r$ and $v = s_0 t \pmod r$
6 Compute $T = [u]P + [v]Q = (x_1, x_2)$ and set $z = x_1$. If $T = \mathcal{O}$, then return (reject the signature).
7 If $s_0 = z \pmod r$, then return (accept the signature). Else return (reject the signature).
8 *Output*: reject or accept the signature

Elliptic curve discrete logarithm problem (ECDLP)

In all the above cryptosystems, the main operation performed on elliptic curve points is a scalar multiplication. The hardness of crypto-schemes based on scalar multiplication is based on the difficulty of solving the ECDH or the ECDLP.

Definition 2.7. *Let $\mathcal{E}$ be an elliptic curve over some prime field $\mathbb{F}_p$. Let $P \in \mathcal{E}(\mathbb{F}_p)$ be a point of order r and a, b two integers in the range $[1, r-1]$.*

Given the points $P, [a]P$ and $[b]P$, and without knowing the integers a, b, the ECDH is the problem of determining the point $[ab]P$.

Definition 2.8. *Let $\mathcal{E}$ be an elliptic curve over some prime field $\mathbb{F}_p$. Let $P \in \mathcal{E}(\mathbb{F}_p)$ be a point of order r. Given the point P and the point Q in the cyclic subgroup generated by P, the ECDLP is the problem of determining the integer k in the range $[1, r-1]$ such that $Q = [k]P$.*

The ECDLP is a separate problem from the ECDH, yet for the curves that we use in practice, they are considered equivalent [SVM04]. The ECDLP is considered to be harder that the DLP on some finite field or the integer factorization problem. This is because the index calculus attack [Adl79] to solve the DLP problem runs in subexponential time, while there is no analogue known until now for the ECDLP. Indeed, the fastest known algorithm to solve the ECDLP is Pollard's rho which runs in exponential time and is no better than the generic algorithm to solve DLP in any group.

Therefore, to achieve a satisfactory security level using RSA or standard DL-based systems one needs to use increasingly large keys, while using elliptic curve schemes one may achieve the same security level using much smaller keys. For example, to achieve a 128-bit level of security, one can use 256-bit keys using elliptic curve schemes or 3072-bit for RSA. The 128-bit security level refers to the length of keys in a symmetric cryptosystem in which case an attack by brute force would need 2^{128} steps to break the system. Concerning ECC and RSA, the estimates are based on the size of keys for which we achieve the same security level if we run the fastest attack algorithm for each case.

2.2.1 Algorithms for scalar multiplication

The main operation in all of the above schemes is the scalar multiplication. Since the introduction of elliptic curves in cryptography, there has been major research in speeding up algorithms for scalar multiplication for elliptic curve groups. Typically, there are three ways to optimize this operation: optimize the operations of the underlying finite field, choose an optimal representation of the curve and of the coordinate system (affine, projective, jacobian, Chudnovsky, mixed coordinate systems), and choose an efficient exponentiation algorithm. One should combine these three choices in order to achieve an efficient algorithm taking into consideration that the effects of the field arithmetic and the curve representation are not independent from the effects of the exponentiation algorithm.

In what follows, we present some algorithms that are used to speed up elliptic curve scalar multiplication.

Binary exponentiation

The use of addition chains [Bra39] is a classical way to speed up exponentiation in any group. The main interest of addition chains is reducing the total number of operations performed. For example, using the standard Square-and-Multiply algorithm, computing a^m in a multiplicative group requires at most $\log_2(m)$ squarings and as many multiplications, in the worst case scenario. In addition, in a group where squaring is cheaper than multiplication, to speed up exponentiation one can compute all even powers with squaring instead of multiplying. Based on this observation, numerous algorithms were developed for binary exponentiation, the most simple one being Square-and-Multiply.

Double-and-Add is the equivalent of the Square-and-Multiply algorithm for an additive group. Double-and-Add is used mostly because it presents an efficiency advantage, but it is not constant time which means it has to be modified when implemented to be side-channel resistant. It is mostly used when the scalar is not secret, for example for the verification of ECDSA signatures. Double-and-Add is illustrated by Algorithm 6. The description of the algorithm is taken from [Lon11].

Algorithm 6: Left-to-Right Double-and-Add

Input: $k = (k_{t-1}, k_{t-2}, ..., k_0)_2$ and $P \in \mathcal{E}(\mathbb{F}_p)$
Set $Q \leftarrow \mathcal{O}$.
For $i = t - 1$ downto 0 do
    $Q \leftarrow [2]Q$
    if $k_i = 1$ then $Q \leftarrow Q + P$
Return Q
Output: $[k]P$

When subtraction is cheap, which is always the case for elliptic curve groups, then scalar multiplication can be optimized using a signed digit binary form for the scalar. This form is not unique even when its weight is minimal, which is when it gives the most advantageous result. Among different signed digit binary representations, the non-adjacent form (NAF) is a canonical representation of minimal weight. The NAF representation [Rei60] of an integer is unique, and there are no zeros adjacent. For example, the NAF of 7 is $100\bar{1}$ and the NAF of 3190 is $10\bar{1}001000\bar{1}0\bar{1}0$. Here $\bar{1}$ stands for a coefficient -1 in the signed binary expansion. Algorithm 7 presents elliptic curve scalar multiplication using NAF.

Precomputation can also be exploited to speed up Double-and-Add by means of the Sliding Window Method. The window method decomposes the binary form of the scalar into zero and non-zero words (windows). In general, it is not mandatory that the length of the windows remain equal. Algorithm 8 presents elliptic curve scalar multiplication using the Sliding Window Method.

Algorithm 7: NAF Double-and-Add

Input: $k = (k_{t-1}, k_{t-2}, ..., k_0)_{NAF}$ and $P \in \mathcal{E}(\mathbb{F}_p)$
Set $Q \leftarrow \mathcal{O}$.
For $i = t-1$ downto 0 do
    $Q \leftarrow [2]Q$
    if $k_i = 1$ then $Q \leftarrow Q + P$
    if $k_i = -1$ then $Q \leftarrow Q - P$
Return Q
Output: $Q = [k]P$

Algorithm 8: Sliding Window Double-and-Add

Input: $k = (k_{t-1}, k_{t-2}, ..., k_0)_2$ and $P \in \mathcal{E}(\mathbb{F}_p)$
Compute and store $[w]P$ for all $w = 3, 5, 7, \ldots, 2^d - 1$.
Decompose k into zero and non zero windows F_i
    of length $L(F_i)$ for $i = 0, 1, 2, .., k-1$.
Set $Q = [F_{k-1}]P$
For $i = k-2$ downto 0 do
    $Q \leftarrow [2^{L(F_i)}]Q$
    if $F_i \neq 0$ then $Q \leftarrow Q + [F_i]P$
Return Q
Output: $Q = [k]P$

Except for the standard binary representation of the scalar, there has been other ideas proposed to speed up scalar multiplication, such as representations based on double based number systems [DJM98, DIM05].

Montgomery Ladder

A very nice algorithm to speed up scalar multiplication on elliptic curve groups is the Montgomery Ladder [Mon87], illustrated in Algorithm 9. It is particularly fast when the curve can be put into a Montgomery representation (i.e. the Weierstrass equation is replaced by $by^2 = x^3 + ax^2 + x$) over finite fields with odd characteristic. The Montgomery ladder is a generic group algorithm for exponentiation, but it presents an efficiency advantage for elliptic curves because it works only with computations on the x-coordinate of the point. To be more precise, given P and Q two points on an elliptic curve: (i) from x_P we can compute $x_{[2]P}$ and (ii) from x_P, x_Q and x_{P-Q} we can compute x_{P+Q}. Using these two operations, we can compute $x_{[k]P}$. For a detailed proof of the Montgomery algorithm see [Mon87]. The y coordinate can be computed efficiently, if needed, in the end of the routine [LD99, OS01]. In any case, many

cryptographic schemes, such as ECDH for example, do not really need a y coordinate. Since all computations can be executed only using the x-coordinate, a lot of field multiplications can be spared, which results to a significant efficiency advantage. Moreover, since the y coordinate is ignored until the end of the routine, fewer memory is needed.

The Montgomery Ladder presents also a security advantage against side channel attacks. This is because the conditional branching in the loop is highly regular: whatever the processed bit, an addition and a doubling always take place. As a result, the algorithm is constant time and therefore side–channel resistant.

Algorithm 9: Montgomery Ladder

```
Input: P, k = (k_{t-1}, ..., k_0)_2
Output: Q = [k]P
Initialization: R_0 ← O_G; R_1 ← P
For j = t − 1 downto 0 do
    if (k_j = 0) then
        R_1 ← R_0 + R_1; R_0 ← 2(R_0)
    else [if (k_j = 1)]
        R_0 ← R_0 + R_1; R_1 ← 2(R_1)
Return R_0
```

The Gallant-Lambert-Vanstone (GLV) algorithm

Apart from scalar multiplication algorithms for generic groups such as the ones presented in the previous section, there exist other types of algorithms that can only be applied on elliptic curves, mainly because they exploit the internal structure and properties of elliptic curve groups. Such an example is the algorithm proposed by C. Doche, T. Icart, and D.R. Kohel that uses isogeny decomposition to accelerate computation[DIK05]. Another example is the algorithm GLV initially proposed by Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone in [GLV01]. The GLV algorithm performs scalar multiplication on elliptic curves with efficiently computable endomorphisms.

The idea is the following: Let $\mathcal{E}$ be an elliptic curve over a prime field $\mathbb{F}_p$. We want to compute a multiple of $P \in \mathcal{E}$, say $[k]P$, with $k \in \mathbb{N}$. Suppose that there exists an efficiently computable endomorphism $\phi : \mathcal{E} \to \mathcal{E}$, which one can compute using only a few field operations, and which acts as a multiplication on $\langle P \rangle$; i.e. $\forall Q \in \langle P \rangle, \phi(Q) = [\lambda]Q$, for some $\lambda \in \mathbb{N}$ (or equivalently $\phi(P) \in \langle P \rangle$). For example, if the prime of the base field satisfies the condition $p \equiv 1(\mod 4)$ and i is a square root of -1 in $\mathbb{F}_p$, then any curve of the form $y^2 = x^3 + ax$ has

an explicit and very efficient endomorphism:

$$\phi(x, y) = (-x, iy).$$

To compute $\phi(x, y)$ one has to perform only one field multiplication. In this case, $\lambda = \sqrt{-1}$. The integer λ that characterizes the endomorphism ϕ is one of the eigenvalues of ϕ on $\langle P \rangle$; that is, one of the roots modulo N of the characteristic polynomial of ϕ. Further details are given in Chapter 4. Given the existence of an efficiently computable endomorphism ϕ, computing $[k]P$ breaks down to computing $[k]P = [k_1]P + [k_2]\phi(P)$ with $k = k_1 + k_2\lambda$. This can be computed efficiently, using a multi-exponentiation algorithm, if k_1 and k_2 are relatively short. But if λ is large enough (if $\lambda > \sqrt{N}$ with N the order of the cyclic group $\langle P \rangle$) then we can always find k_1 and k_2 relatively small (roughly $\sqrt{N}$).

More precisely, GLV is the composition of three independent algorithms:

1. The computation of the endomorphism ϕ.
2. **The multiexponentiation algorithm**. The algorithm, depicted in Algorithm 10, was used in [GLV01]; it is an extension of the sliding window binary algorithm presented above. Note that if the window size $w = 1$, then we get a 2-dimensional analogue of the Double-and-Add algorithm. There are various alternatives for efficient multiexponentiation and some of them are discussed in [Str64].

Algorithm 10: Simultaneous sliding window exponentiation in an additive group

Input: $w \in \mathbb{N}$, $w \neq 0$, $u = (u_{t-1}, ...u_1, u_0)_2$, $v = (v_{t-1}, ...v_1, v_0)_2$, P, Q.
Compute $iP + jQ$ for all $i, j \in [0, 2^{w-1}]$.
Write $u = (u^{d-1}, ..., u^1, u^0)_2$ and $v = (v^{d-1}, ..., v^1, v^0)_2$ where each u^i and v^i is a bitstring of length w and $d = \frac{t}{w}$.
Set $R \leftarrow 0$.
For i from $d - 1$ downto 0 do
    $R \leftarrow 2^w R$
    $R \leftarrow R + (u^i P + v^i Q)$
Return R
Output: $R = [u]P + [v]Q$

3. **The scalar decomposition**. In the decomposition of the scalar $k = k_1 + \lambda k_2$, the integers k_1 and k_2 are not unique. To compute k_1, k_2, one needs to find two linearly independent vectors $v_1 = (x_1, y_1), v_2 = (x_2, y_2)$ of $\mathbb{Z}^2$ which satisfy $x_1 + \lambda y_1 \pmod N \equiv x_2 + \lambda y_2 \pmod N \equiv 0 \pmod N$. For example, two such vectors are $v_1 = (N, 0)$ and $v_2 = (\lambda, -1)$. Then, any

vector v in the lattice $\langle v_1, v_2 \rangle$ results in the following simple decomposition $(k_1, k_2) = (k, 0) - v$. However, the GLV algorithm is efficient if k_1 and k_2 have roughly half the bitlength of k, which requires v to be close to $(k, 0)$. Such a v can be easily computed given a short basis for $\langle v_1, v_2 \rangle$, and this basis can be easily precomputed using lattice basis reduction algorithms. The standard technique for decomposition proposed in [GLV01] is based on the Extended Euclidean algorithm and is presented in details in Chapter 4.

The main advantage of GLV is efficiency: assuming that k_1 and k_2 have half the length of the original scalar k, then roughly half of the doublings will be eliminated if we use multi-scalar multiplication techniques as the one presented above. As a result, GLV is particularly fast and useful especially in the case where the base point is variable.

However, the main disadvantage that GLV presents is that it can only be implemented on curves with efficiently computable endomorphisms. Unfortunately, finding such curves turns out to be highly nontrivial and that is why in 2009 Galbraith, Lin and Scott proposed a modified version of GLV, named the GLS algorithm in their article [GLS09]. GLS solves the problem of finding such curves in the following way: starting with any elliptic curve over a prime field, first we extend the curve to the quadratic extension field and then use an efficiently computable homomorphism which arises from the Frobenius map on the quadratic twist of the curve. Furthermore, the Q-curve construction presented in [Smi16] is a generalization of the GLS algorithm.

Since its publication on 2001, significant research has been done to optimize performance of GLV [FLS15], to analyze its security properties and its applicability to different settings [BCHL13, LS14].

GLV is an algorithm particularly interesting to formalize, first because it is an algorithm that is actually used in real-life cryptographic implementations and secondly, because the mathematics involved are not trivial: besides two generic algorithms (multiexponentiation and shortest vector for decomposition), the use of endomorphisms requires formal theory for several non-trivial properties of elliptic curves. We do this in Chapter 4.

2.2.2 Use of different coordinate systems

Significant research has been going on lately concerning different coordinate systems in order to provide resistance against side channel attacks and allow for more efficient implementations.

As explained previously, the elementary representation of elliptic curve points uses affine coordinates (x, y). However, addition formulas based on affine coordinates demand the computation of field inversions, which is particularly expensive over finite fields. To avoid field inversions, one can use projective

coordinates of the form $(X : Y : Z)$. The equivalence between affine and projective representation is explained in detail in Chapter 3.

A special case of projective coordinates are Jacobian coordinates: A point in Jacobian coordinates is the equivalence class $(X : Y : Z) = \{(\lambda^2 X, \lambda^3 Y, \lambda Z) \mid \lambda \in \mathbb{F}_p^*\}$. In this case, a Jacobian point $(X : Y : Z)$, with $Z \neq 0$, corresponds to the affine point $(\frac{X}{Z^2}, \frac{Y}{Z^3})$. The curve equation becomes $Y^2 = X^3 + aXZ^4 + bZ^6$ and the point at infinity is the point $\mathcal{O} = (1 : 1 : 0)$.

There are many other variants of coordinate systems such as mixed coordinates [CMO98] and modified Jacobian and Chudnovsky coordinates [CC86]. For more details see [Lon11]. The most common form of elliptic curves over prime fields in cryptographic settings is the short Weierstrass form $y^2 = x^3 + ax + b$ with $a, b \in \mathbb{F}_p$. The projective form of this equation using Jacobian or homogeneous projective coordinates has been accepted as a standard by NIST and IEEE [Lon11, Nat99]. Yet, there has been significant research going on new optimized curve forms. These forms, which are only beginning to be standardized for some applications, may present important efficiency or security advantages such as fast arithmetics or side channel resistance. For more details on different curve forms see [Gal12].

To sum up, different curve forms and their corresponding coordinate system may be a better choice depending on the scalar multiplication algorithm used. In general, when implementing elliptic curve scalar multiplication, one has to choose the algorithm together with the curve and the coordinate system, taking into account all the dependences between the three. Moreover, one has to bear in mind the underlying field arithmetic together with the efficiency and security requirements of his own implementation. As a result, elliptic curve scalar multiplication implementation is a complex task and most of the time verifying that an implementation is actually correct is far from trivial.

2.3 Coq and its SSREFLECT extension

Proof assistants are programs allowing the interactive development and automatic verification of programs and mathematical statements proofs. Coq [The10] belongs to a large family of proof assistants including NuPrl, PVS, HOL, Isabelle, Mizar, Lego.

Coq is the result of more than 30 years of active research, starting with the work of Thierry Coquand and Gérard Huet [CH85] in 1984 at INRIA. The architecture of Coq is based on two layers: the kernel and the proof engine. The proof engine provides the tools, or *tactics*, allowing the interactive construction of proofs. Coq comes with a set of predefined tactics, and a language for the users to write their own tactics. The kernel is the core engine of Coq. It checks that a proof constructed by the proof engine rely on valid logical reasoning. As

such, the kernel guarantees the correctness of Coq and its proof engine. Proofs and statements are expressed in a language called *Gallina*, based on an extension of the *Calculus of Inductive Constructions* [The10, PP89], a dependently typed polymorphic lambda calculus.

2.3.1 Propositions and Types

Types are a central notion in Coq. Indeed, in the Coq language every valid expression comes with a type. Types determine if an expression is well formed or not: rules for building expressions are accompanied by *typing rules* that show the relation between the type of the whole expression and the type of its parts. For example, assume we declare a variable `a` of type `nat` (we say that a is an inhabitant of the type `nat`), which stands for the type of natural numbers. The constant 8 being also an inhabitant of `nat`, we can deduce that `8 + a` has type `nat`. On the contrary, the expression `a + false`, where `false` is a constant of type `bool`, is not well-formed. Such an expression is forbidden by the typing rules.

There exist a wide variety of types in Coq, as well as type constructors. For instance, pairs in Coq are defined as a higher-order datatype `prod`: from two types A and B, one can construct the type $A \times B$ of pairs (a, b) where a is of type A and b is of type B.

```
Inductive prod (A B : Type) : Type := pair : A -> B -> A x B
```

Coq comes with a special type named **`Prop`** which stands for the type of propositions.

The *Curry-Howard isomorphism* describes the relation between proofs and programs: the relation between a program (or expression) and its type is the same as the relation between a proposition and its proof. For example, assume that one wants to prove that $P \Rightarrow A$ under the given list of assumptions (or *environment*) E. In minimal propositional logic, if a proof of A under the assumptions $E \cup \{P\}$ is known then one may derive a proof a $P \Rightarrow A$ under E, as stated by the following rule:

$$\frac{E, P \vdash A}{E \vdash P \Rightarrow A}$$

where $E \vdash A$ stands for A is valid under the assumptions in E. This rule is tightly related to the Coq typing rule for the formation of functions, as given below:

$$\frac{E, x : P \vdash f(x) : A}{E \vdash ((x : P) \mapsto f(x)) : P \to A}$$

The rule works as follows: if when assuming a variable x of type P one can deduce that $f(x)$ is of type A, then the function $(x : P) \mapsto f(x)$ is of type $P \to A$. By removing the variables and expressions from the second one, we

come back to the first rule. The Curry-Howard isomorphism establishes this fact: *The relation between a program (i.e. a function) and its type is the same as the relation between a proposition and its proof.* Hence, an expression $e : T$ can be either interpreted as a program e of type T or a proof e of a proposition T. See [Gil04] for further explanations.

Dependent Types are important

A type that is parametrized by values is called a *dependent* type. Such examples are arrays of size n, binary trees of depth p, but also logical formulas. This has to do with the fact that quantifications can be used to form new types. An example taken from [AM16] demonstrates exactly this functionality: Here a new type is constructed for the existential statement:

```
ex : forall A : Type, (A -> Prop) -> Prop.
```

The type constructor `ex` is parametrized by a type `A` and a predicate on `A`. This statement shows that in Coq we can construct types that play the roles of propositions which may be useful sometimes. Furthermore, this functionality is very convenient because it allows the construction of more complex datatypes such as the type of matrices:

```
matrix : Type -> nat -> nat -> Type.
mulmx : forall R : Type, forall m n p : nat,
  matrix R m n -> matrix R n p -> matrix R m p.
```

In this case, the type `matrix` is parametrized by two natural numbers that correspond to its size. This allows matrix multiplication to be constructed in such a way that only compatible matrices can be multiplied. It also reveals the size of the output matrix, in terms of the size of the input matrices.

2.3.2 Coq by example

We here exemplify the syntax of Coq and introduce its proof engine using minimal intuitionistic propositional logic. Our goal is not to give an exhaustive definition of the Coq language, but to help the reader follow the formalization described later at Chapter 3 and 4. For a complete introduction to the Coq system, see [BC04]. A reader with experience in using proof assistants is invited to skip this section.

Assume that we want to prove the following tautology: $(P \implies P \implies Q) \implies P \implies Q$, where P and Q are propositions. First, we declare two propositional symbols:

```
Parameter P Q : Prop.
```

In Coq, our statement will be expressed as follows, where `->` denotes the logical implication:

```
Goal (P -> (P -> Q)) -> (P -> Q).
```

We will use the two following tactics:

— **move=>** which introduces a hypothesis to the environment,

— **apply** which applies a hypothesis to the goal.

The final proof will be as follows:

```
Goal (P -> (P -> Q)) -> (P -> Q).
Proof.
  move=> HPQ.
  move=> HP.
  apply: HPQ.
  apply: HP.
  apply: HP.
Qed.
```

We now give all the intermediate steps. After inputting the statement, one sees the following:

```
 P : Prop
 Q : Prop
============================
 (P -> P -> Q) -> P -> Q
```

Above the =-line lies the environment with all the declarations and definitions, and bellow the =-line is the current goal that has to be proved. Using **move=>** `HPQ`, the system introduces the head hypothesis with name `HPQ`. The new goal is `P -> Q`:

```
 P : Prop
 Q : Prop
 HPQ : P -> P -> Q
============================
 P -> Q
```

Likewise, using **move=>** `HP` we introduce the second hypothesis `HP` and are left to prove the following goal:

```
 P : Prop
 Q : Prop
 HPQ : P -> P -> Q
 HP : P
============================
 Q
```

Next, we use the tactic **apply** to apply the hypothesis `HPQ` to our new goal.

Remark

Proofs in Coq go backwards: the users start from the conclusion and apply tactics simplifying the goal up to a point where it appears as an hypothesis in the environment. For instance, the `apply:` tactic does a *modus ponens* in the reverse way: if one has to prove the goal `B` and has a proof `H` of `A -> B`, then the tactic `apply: H` will transform the goal `B` to `A`.

In our case, applying `HPQ` generates two new sub-goals requiring a proof of `P`. The first goal comes from the first hypothesis of `HPQ`, whereas the second comes from the second one:

```
 P : Prop
 Q : Prop
 HPQ : P -> P -> Q
 HP : P
============================
 P

subgoal 2 is:
 P
```

Applying hypothesis `HP` solves the first goal. We then move to the second sub-goal:

```
 P : Prop
 Q : Prop
 HP : P -> P -> Q
 HPQ : P
============================
 P
```

Applying once more `HP` closes the proof and Coq displays the message `Proof completed`. At that point, the proof is not checked yet. By inputting `Qed`, we ask the proof engine to send the constructed proof to the kernel of Coq. Only after this step we know that we have a formal proof of our statement. There exist several tactics which allow the user to perform case analysis, proofs by induction, first-order reasoning, automatic proofs search, as well as more powerful tactics or user defined tactics coming from external libraries.

Remark:

In the SSREFLECT language, the tactics `move=>` and `apply:` correspond to the Coq tactics `intro` and `apply` respectively. To be more precise, the tactics are just `move` and `apply`, while the symbols `=>` and `:` are called tacticals, i.e.

tactic modifiers, such that one same tactic may cope with a wide range of similar situations. For more details, see [AM16].

2.3.3 Functions and Equality

In Coq, there exist two ways to denote a function. For example, for $f : x \in \mathbb{N} \mapsto x^2$, one can write `(fun (n:nat) : nat => n * n)` or use a global level definition to give a name to the function:

```
Definition sqr (n:nat) : nat := n * n.
```

The argument (along its type) of `sqr` is denoted by `(n:nat)`. The second `:nat` denotes that `sqr n` is of type `nat`. If we typecheck the `sqr` function, we obtain `sqr:nat->nat`, meaning that `sqr` is a function from $\mathbb{N}$ to $\mathbb{N}$.

A function in Coq is a function from the computer-science point of view, i.e. an algorithm or a *computable function*. On the contrary, from a mathematics point of view usually a function $f : A \to B$ is a subset of $A \times B$ - its graph. Take for example, the functions $f(x, y) = (x + y)(x - y)$ and $g(x, y) = x^2 - y^2$. These functions are considered equal in mathematics since their graphs are extensionally equal: given the same input, they always produce the same output. Yet, the definitions of the functions are not equal, meaning that as algorithms they are not the same. In that intensional sense f and g are not the same and so they are not considered equal in Coq.

Coq primitive equality is the Leibniz one: $x = y$ if for any predicate P, $P(x)$ implies $P(y)$, which does not capture the extensional equality. As such, $\forall x.\, f(x) = g(x)$ *does not imply* $f = g$.

2.3.4 Inductive types

In Coq one can define inductive data-types. An inductive type represents the set of expressions built by a finite number of applications of its constructors. For example, a simple inductive type is an enumerated type which represents a finite fixed set, as the one for booleans:

```
Inductive bool : Type := true | false.
```

This definition produces two elements of type `bool`: the two constructors `true` and `false`.

Another example of inductive types is the natural numbers. $\mathbb{N}$ is defined (as in Peano arithmetic) as:

```
Inductive nat : Type := 0 | S of nat.
```

The type `nat` has two constructors: `0` which stands for zero and `S` for the successor function.

For every inductive definition, an inductive principle [CP88, Tar55] is generated by Coq allowing the user to develop proofs by induction, or define function by recursion. For example, lists of elements of type `A` are defined as:

```
Inductive list (A: Type) : Type := nil | cons of A & list A.
```

where `nil` is the empty list and `cons` is the function that concatenates an element of type `A` with a list of type `list A`.

If one wishes to prove a statement on lists by induction, she needs to prove the following facts:

1. that the statement stands for the empty list, and
2. that if the statement stands for a list `L`, then for any elements `a`, the statement stands for the concatenation `cons a L`.

Let's take a look at the following example from [Ber08]. Suppose we want to prove the following statement, using induction on natural numbers:

```
Lemma addn0 : forall n, n + 0 = n.
```

The following two lemmas will be useful:

```
Lemma add0n : forall n : nat, 0 + n = n.
Lemma addnS : forall n m : nat, S n + m = S (n + m)
```

After inputting the statement, one sees the following:

```
forall n : nat, n + 0 = n
```

Using `move=> n`, the system introduces the natural number n in the environment. The new goal is

```
 n : nat
=============
 n + 0 = n
```

Next we use the tactic `elim: n => [|n ih]` to perform induction on the natural number n. Two subgoals are generated, the first one concerns the case where $n = 0$ and the second one the general case where n is any natural number different from 0:

```
 2 subgoals
 =============
 subgoal 1 is:
 0 + 0 = 0

 subgoal 2 is:
 S n + 0 = S n
```

Applying lemma `add0n`, the first goal disappears. Now the second goal looks like:

```
 n : nat
 ih : n + 0 = n
 =============
  S n + 0 = S n
```

We observe that Coq has generated the induction hypothesis `ih` for the natural number `n`, and we have to prove the statement for its successor `S n`. By rewriting lemma `addSn`, we obtain:

```
  n : nat
  ih : n + 0 = n
 =============
  S (n + 0) = S n
```

Now by rewriting the induction hypothesis the goal turns to

```
  n : nat
  ih : n + 0 = n
 =============
  S n = S n
```

Using the tactic **`reflexivity`** finishes the proof and Coq displays the message **`Proof`** `completed`. Indeed, **`reflexivity`** is a tactic that finishes the proof when the goal looks like `e = e`. The entire proof script is

```
Lemma addn0 : forall n, n + 0 = n.
Proof.
move=> n.
elim: n => [|n ih].
rewrite add0n.
rewrite addSn.
rewrite ih.
reflexivity.
Qed.
```

Coq allows us to do proofs by case analysis using the tactic **`case`**. For example, given a boolean statement `b : bool` we can perform case analysis to prove a statement about `b`. Let us prove the following statement to demonstrate how **`case`** works: Negation is involutive.

```
Lemma negbK (b:bool): ~~ (~~ b) = b.
```

The proof script will be the following:

```
Lemma negbK (b:bool): ~~ (~~ b) = b.
Proof. by case: b. Qed.
```

Let's examine the intermediate steps. First, after inputting the statement, one sees the following: `~~ (~~ b) = b`. Next we use the tactic **`case`**`: b` to perform

case analysis on the boolean `b`. Two subgoals are generated, the first one concerns the case where $b = true$ and the second one the general case where $b = false$:

```
b : bool

2 subgoals
============================
subgoal 1 is:
 ~~ ~~ true = true

subgoal 2 is:
~~ ~~ false = false
```

In both goals we first apply the tactic **`simpl`** to simplify, and the goal is transformed to `true = true` (resp. `false = false`) which is then resolved by the tactic **`reflexivity`**. This is done automatically if we use the **`by`** tactical which closes the proof script. The proof is completed.

2.3.5 The Ssreflect extension

The Small Scale Reflection is a set of extensions for Coq developed to support proof methodology for algebra. It comes with a set of tactics and a library. Ssreflect was first designed for the proof of the Four Colour theorem [Gon07] but has afterwards evolved to prove the Feit-Thompson theorem [GAA+13]. The Ssreflect library contains theory about groups, algebraic structures, linear algebra, polynomials and matrices, representation and character theory [Bih10, Bih09, Gon11, GMR+07]. In this thesis, we do not explain in details the Ssreflect methodology. For an extended introduction to Ssreflect, see [AM16, GM10, Gar11].

Boolean Reflection The logic of Coq is constructive, which implies that the excluded middle principle does not hold in all cases. More precisely, the excluded middle holds and can be proved for some property, only if this property is decidable: if one can write a function in Coq that outputs a boolean value and tests if the property holds or not. In the Mathematical Components library, usually all decidable properties are formalized as boolean tests (so in most cases we can consider that the excluded middle holds). A significant (and very interesting) issue in formalizing mathematics is that a structure or a statement can be represented in many different ways. When the principle of excluded middle holds for some property, there are two ways to express this property: using boolean values or logical connectives. Boolean values have the advantage that case analysis can easily be performed, while logical connectives give statements (in the **`Prop`** sort) that can be destructed to provide subformulas (e.g. from

conjunctive or disjunctive facts) or witnesses of existential statements. Note that to formalize a property in `bool`, one needs to implement the decision procedure that decides if the property holds. SSREFLECT proposes the *boolean reflection* methodology which allows interchanging between equivalent statements in **`Prop`** or `bool` [AM16]: One expresses the same statement in **`Prop`** and `bool`, and then has to prove a lemma (called view lemma) stating the equivalence between the **`Prop`** and the `bool` statements. For more details on the Small Scale Reflection methodology, see [AM16, GM10]. We here give the example, extracted from the SSREFLECT library, for defining what is a type whose equality predicate is decidable (i.e. whose equality predicate can be reflected using a boolean equality):

```
Record eqSpec (T: Type) := EqSpec {
  eq: T -> T -> bool;
  eqAxiom: forall (x y : T), (eq x y) = true <-> x=y
}.

Record eqType : Type := EqType {
  base : Type ;
  mixin : eqSpec base
}.
```

The record `eqSpec` defines the specifications of a type with equality, while the record `eqType` packs the base type (the carrier) with the `eqSpec` specifications.

Last, we would like to highlight that proof irrelevance is particularly useful for Σ-types `sT := {x : T | P x}` of elements of type `T` satisfying the boolean statement `P`. In that case, two inhabitants of the type `sT` are equal if their first projections are equal. This is not necessarily true in the case that `P` is not a boolean statement, because the proofs that `x` satisfies `P x` may not be the same. The Σ-types for which the above property holds are called subtypes.

Algebraic Hierarchy and Canonical Structures Between algebraic structures, inheritance and sharing structure is very common: For example, a ring is an abelian group under addition, which means that rings and abelian groups share some theory. When formalizing algebraic structures, it is very important to implement the sharing of mathematical structures in a way that enables inheritance and maximizes sharing in order to avoid repeating the same proofs again and again.

Algebraic structures in SSREFLECT are usually described as a carrier set along with a number of functions and a set of properties the functions have to satisfy. SSREFLECT uses bundled representation schemes which support multiple inheritance between algebraic structures, and allows the encoding of the algebraic hierarchy.

More precisely, an algebraic structure is usually implemented by a record with two fields: the carrier set and the class of the structure. The class contains the signature (i.e. the name and type of the operations) and proofs of the properties that the operations satisfy. For a detailed description of the algebraic structures in SSREFLECT see [Coh12].

For example, the zmodType, which stands for an abelian group, is defined as the subtype of eqType as follows:

```
Record zmodSpec (V : Type) : Type := ZmodSpec {
  zero : V;
  opp : V -> V;
  add : V -> V -> V;
  _ : associative add;
  _ : commutative add;
  _ : left_id zero add;
  _ : left_inverse zero opp add
}.

Record zmodType : Type := ZmodType {
  base :> eqType;
  mixin : ZmodSpec base
}.
```

In this case the specifications of zmodType is given by zmodSpec and the record zmodType packs the above specifications with an eqType. As a subtype of eqType, zmodType inherits the functions and properties of eqtype - i.e. it has a computable equality.

In mathematics, when one considers an element of a certain algebraic structure, she implicitly considers an element of the carrier set of the structure. For example, given a certain group `G : zmodType`, an element `x : G` is indeed an element `x : ZmodType.sort G`, where `ZmodType.sort G` is the projection to the carrier set of `G`. So for example, for two elements `x y : G` writing `x + y` poses no problem because it is typable. If we remove notations, the statement

```
forall x y : G, x + y = 0
```

is expanded to

```
forall x y : Zmodule.sort G, @add G x y = zero G
```

Nevertheless, problems may arise in other cases where one does not manipulate directly the defined structure. For example, integers form an abelian group and therefore share the zmodType structure. But when writing

```
forall x y : int, x + y = y + x
```

an error arises because `int` is not of type `zmodType`. To address this problem, unification can be aided by Coq's *Canonical Structures* mechanism. For a detailed use of Canonical Structures in SSREFLECT, see [Coh12].

Quotient Types When working with algebraic structures, one often has to manipulate quotients and functions defined on them. In that case, different (not equal) terms may represent the same conceptual object.

Manipulating quotients is difficult in Coq, because of type theory. In mathematics, algebraic quotients are sets, for which structure is transported from the base set to the quotient. An element x of the base set characterizes its equivalence class of the quotient set, so one can implicitly consider x as an element of the base set and an element of the quotient set. However when it comes to Coq, defining a quotient type as a subtype of the base type does not allow free interchanges between the two as they are two different types. In type theory there are two options proposed about dealing with quotients: setoids and forging quotient types i.e. define types where each element is one equivalence class of the quotient. For more details, see [Coh12]. Moreover, in Coq, quotient types are not primitive types as it would make type-checking indecidable [SvdW11].

SSREFLECT deals with this problem by restricting the quotient types to decidable ones. A quotient type U in SSREFLECT is represented by a packed structure that binds together:

1. the base type T,
2. a function $\pi : T \to U$, called *canonical surjection*, which is the embedding of T in U, and
3. a function $\mathrm{repr} : U \to T$ s.t. for a class C in U, $\mathrm{repr}(C)$ gives a representative in T for C

s.t. the composition of the canonical surjection with the representative function $\pi \circ \mathrm{repr}$ is the identity function. A quotient type `Q` is an instance of the following interface:

```
Structure quotType (T : Type) := QuotType {
  quot_sort :> Type;
  quot_class : quot_class_of quot_sort
}.

Record quot_class_of (T Q : Type) := QuotClass {
  repr : Q -> T;
  pi : T -> Q;
  reprK : forall x, pi (repr x) = x
}.
```

Hence, to define a quotient type, one has to be able to give a function which computes the representative for every class. A quotient type `Q` of base type `T` and quotient structure `qT` results in the following equivalence for elements of `T`: Two elements `x y : T` are equivalent if `pi qT x = pi qT y`. SSREFLECT allows for the following more intuitive notation `x = y \%[mod qT]` of the above.

Given an arbitrary type `R`, a function `f : T -> R` is compatible with the quotient type `Q` if it stays constant on each equivalence class. In that case it has a lifting which means that we can define a corresponding function `g : Q -> R` on the quotient type `Q`, such that `pi f x = g pi x` for all `x : T`. The canonical surjection `pi` is a morphism for `f`.

The proof of the Picard theorem requires the definition of functions over quotients. A common method, when working with quotients, is to define a function on the non-quotiented set and prove that the function respects the quotient, i.e. that for any x, $f(x) = f(\mathrm{repr}(\pi(x)))$.

3

A formal library for elliptic curves

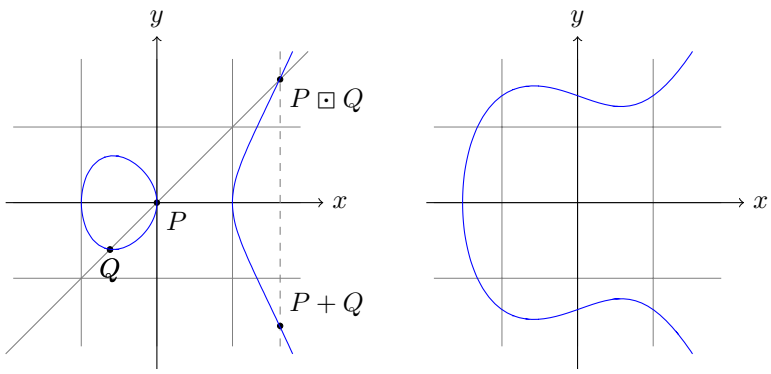
In this chapter, we present a formal proof of the Picard theorem for elliptic curves: There exists an isomorphism between the Picard group of divisors and the group of points of an elliptic curve. An important consequence of this proposition is the associativity of the elliptic curve group operation. This development has resulted in more than 15000 lines of code and is available at https://github.com/strub/glv. It includes formal theory about Weierstrass curves, the field of rational functions on a curve, theory about free groups, divisors of rational functions on curves and isomorphic representations on different coordinate systems. This result, has been published at the article *A Formal Library for Elliptic Curves in the Coq Proof Assistant* at the International Theorem Proving conference 2014.

3.1 Elliptic curves definitions

An elliptic curve is a special case of a projective algebraic curve that can be defined as follows:

Definition 3.1. *Let $\mathbb{K}$ be a field. Using an appropriate choice of coordinates, an* elliptic curve *$\mathcal{E}$ is a plane cubic algebraic curve $\mathcal{E}(x, y)$ defined by an equation of the form:*

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



(a) $y^2 = x^3 + x$ (b) $y^2 = x^3 - x + 2$

Figure 3.1 – Catalog of Elliptic Curves Graphs

where the a_i's are in $\mathbb{K}$ and the curve has no singular point (i.e. no cusps or self-intersections). The set of points, written $\mathcal{E}(\mathbb{K})$, is formed by the solutions (x, y) of $\mathcal{E}$ augmented by a distinguished point $\mathcal{O}$ (called point at infinity*):*

$$\mathcal{E}(\mathbb{K}) = \{(x, y) \in \mathbb{K} \mid \mathcal{E}(x, y)\} \cup \{\mathcal{O}\}.$$

Figure 3.1 provides graphical representations of such curves in the real plane.

When the characteristic of $\mathbb{K}$ is different from 2 and 3 and with an appropriate change of coordinates, the equation $\mathcal{E}(x, y)$ can be simplified into its *Weierstrass* form:

$$y^2 = x^3 + ax + b.$$

Moreover, such a curve does not present any singularity if $\Delta(a, b) = 4a^3 + 27b^2$ — the curve's discriminant — is not equal to 0. Our work lies in this setting.

The parametric type `ec` represents the points on a specific curve. It is parameterized by a `K : ecuFieldType` — the type of fields with characteristic not in $\{2, 3\}$ — and a `E : ecuType` — a record that packs the curve parameters a and b along with a proof that $\Delta(a, b) \neq 0$. An inhabitant of the type `ec` is a point of the projective plane (represented by the type `point`), along with a proof that the point is on the curve.

Note that in this setting, the type `point` formalizes the projective plane as the set of affine finite points, together with the point at infinity. Nevertheless, in the next section we present a more general formalization of the projective plane using a three coordinate system and a proof that those two representations are isomorphic.

```
Record ecuType := { A : K; B : K; _ : 4 * A^3 + 27 * B^2 != 0 }.
```

```
Inductive point (K : Type) : Type :=
| EC_Inf : point K
| EC_In : K -> K -> point K.

Notation "(| x , y |)" := (@EC_In _ x y)

Definition oncurve (p : point K) := (
  match p with
  | EC_Inf => true
  | EC_In x y => y^+2 == x^+3 + A * x + B
  end).

Inductive ec : Type := EC p of oncurve p.
```

Using standard SSREFLECT methodology, the type `point` is equipped with the structure of an `eqType`, `choiceType` and `countType`. As a subtype of `point`, the type `ec` inherits the same structures.

The points of an elliptic curve can be equipped with a structure of an abelian group. We give here a geometrical construction of the law. Let P and Q be points on the curve $\mathcal{E}$ and l be the line that goes through P and Q (or that is tangent to the curve at P if $P = Q$). By the Bézout theorem, counting multiplicities, l intersects $\mathcal{E}$ at a third point, denoted by $P \boxdot Q$. The sum $P + Q$ is the opposite of $P \boxdot Q$, obtained by taking the symmetric of $P \boxdot Q$ with respect to the x axis. Figure 3.1 highlights this construction. To sum up:

1. $\mathcal{O}$ is defined to be the neutral element: $\forall P. P + \mathcal{O} = \mathcal{O} + P = P$,
2. the opposite of a point (x_P, y_P) (resp. $\mathcal{O}$) is $(x_P, -y_P)$ (resp. $\mathcal{O}$), and
3. if three points are collinear, their sum is equal to $\mathcal{O}$.

This geometrical definition can be translated into an algebraic setting, obtaining polynomial formulas for the definition of the law. Having such polynomial formulas leads to the following definitions:

```
Definition neg (p : point K) :=
if p is (|x, y|) then (|x, -y|) else EC_Inf.

Definition add (p1 p2 : point K) :=
  let p1 := if oncurve E p1 then p1 else EC_Inf in
  let p2 := if oncurve E p2 then p2 else EC_Inf in

  match p1, p2 with
    | EC_Inf, _ => p2 | _, EC_Inf => p1

    | (|x1, y1|), (|x2, y2|) =>
      if x1 == x2
```

```
    then if (y1 == y2) && (y1 != 0)
         then
           let s := (3 * x1^+2 + E#a) / (2 * y1) in
           let xs := s^+2 - 2 * x1 in
             (| xs, - s * (xs - x1) - y1 |)
         else
           EC_Inf
    else
      let s := (y2 - y1) / (x2 - x1) in
      let xs := s^+2 - x1 - x2 in
        (| xs, - s * (xs - x1) - y1 |)
  end.
```

Note that these definitions do not directly work with points on the curve, but instead on points of the projective plane (points that do not lie on the curve are projected to $\mathcal{O}$).

We link back this algebraic definition to its geometrical interpretation. First, we define a function `line` that, given two points P, Q on the curve, returns the triplet (u, v, c) that characterizes the equation $ux + vy + c = 0$ of the line (PQ) intersecting the curve at P and Q (resp. the equation of the tangent to the curve at P if $P = Q$). We then show that, if (PQ) is not parallel to the y axis (i.e. is not intersecting the curve at $\mathcal{O}$), then (PQ) is intersecting $\mathcal{E}$ exactly at P, Q and $-(P + Q) = P \boxdot Q$ as defined algebraically. This is the main part of the proof of the lemma `add0`, which implies the computational manipulation of polynomial formulas.

The function `line` is defined using polynomial formulas as below:

```
Definition line (p q : K * K) : K * K * K :=
  let: (x1, y1) := p in
  let: (x2, y2) := q in

  match oncurve E (|x1, y1|) && oncurve E (|x2, y2|) with
  | false => (0, 0, 0)
  | true =>
      if x1 == x2
      then if (y1 == y2) && (y1 != 0)
           then
             let s := (3 * x1^+2 + E#a) / (2 * y1) in
               (1, -s, y2 - s * x2)
           else
             (0, 1, x1)
      else
        let s := (y2 - y1) / (x2 - x1) in
          (1, -s, y2 - s * x2)
```

```
  end.
```

We then prove that these operations are internal to the curve and lift them to $\mathcal{E}$. We further prove that they satisfy all the properties of an abelian group except associativity.

```
Lemma zero0 : oncurve E EC_Inf.
Notation zeroec := (EC zero0).

Lemma neg0 : forall p, oncurve E p -> oncurve E (neg p).
Definition negec := [fun p : ec E => EC (neg0 [oc of p])].

Lemma add0 (p q point K): oncurve E (add p q).
Definition addec := [fun p1 p2 : ec E => EC (add0 p1 p2)].

Lemma addNe : left_inverse zeroec negec addec.
Lemma add0e : left_id zeroec addec.
Lemma addeC : commutative addec.
```

As pointed out before, we have defined the operations on points of the projective plane and then lifted them to points on the curve. Moreover, properties and lemmas concerning the operations are proven for elements of type `point` in order to be as general as possible. This is a structural difference between this development and the development [TH07] where addition is defined directly on elliptic curve points. Our choice allows the manipulation of operations on points, without demanding to prove a priori that they belong on the curve.

3.2 The Picard group of divisors

From now on, let $\mathcal{E}$ be a smooth elliptic curve with equation $y^2 = x^3 + ax + b$ over the field $\mathbb{K}$. We assume that $\mathbb{K}$ is not of characteristic 2, nor 3. Related to this curve, we assume two Coq parameters `K : ecuFieldType` and `E : ecuType K`. We now move to the construction of the *Picard group* $\mathrm{Pic}(\mathcal{E})$. This construction is split into several steps:

1. we start by constructing two objects: the field of rational functions $\mathbb{K}(\mathcal{E})$ over $\mathcal{E}$ and the group of $\mathcal{E}$-divisors $\mathrm{Div}(\mathcal{E})$, i.e. the set of formal sums over the points of $\mathcal{E}$. From $\mathrm{Div}(\mathcal{E})$, we construct $\mathrm{Div}^0(\mathcal{E})$ which is the subgroup of zero-degree divisors.
2. Then, we attach to each rational function $f \in \mathbb{K}(\mathcal{E})$ a divisor $\mathrm{Div}(f)$ (called *principal divisor*) that characterizes f up to a scalar multiplication. This allows us to define the subgroup $\mathrm{Prin}(\mathcal{E})$ of $\mathrm{Div}(\mathcal{E})$, namely the *group of principal divisors*. The quotient group $\mathrm{Div}^0(\mathcal{E})/\mathrm{Prin}(\mathcal{E})$ forms the *Picard*

group.

3.2.1 The field of rational functions $\mathbb{K}(\mathcal{E})$

The construction of the field of rational functions presents two key points that need to be up-fronted:

1. Polynomials that take the same values on the same curve points are considered equivalents and are identified. This leads to the definition of the ring $\mathbb{K}[\mathcal{E}]$.
2. From the evaluation of rational functions on all points of an elliptic curve arises the problem of evaluating fractions with zero denominators and the problem of evaluating fractions at the point at infinity. This induces the concept of the *order* of vanishing of a function at a point.

The ring $\mathbb{K}[\mathcal{E}]$

We denote the ring of bivariate polynomials over $\mathbb{K}$ by $\mathbb{K}[x, y]$.

Considering bivariate polynomials with variables x and y, we are interested on their evaluation on curve points. On the curve, the variables x and y are not independent, they are related by the curve equation $y^2 = x^3 + ax + b$. So, two polynomials whose evaluation is the same on all curve points will be considered equivalent. For example, the polynomials y^2 and $x^3 + ax + b$ are equivalents. More generally, if we replace an occurrence of y^2 by $x^3 + ax + b$, we obtain an equivalent polynomial and two polynomials whose difference is a multiple of $y^2 - (x^3 + ax + b)$ are equivalent. That leads directly to the following definition:

Definition 3.2. *The ring* $\mathbb{K}[\mathcal{E}]$ *of* polynomials over the curve *is defined as the quotient ring of* $\mathbb{K}[x, y]$ *by the prime ideal* $\langle y^2 - (x^3 + ax + b)\rangle$. *The field* $\mathbb{K}(\mathcal{E})$ *is defined as the field of fractions over the integral domain* $\mathbb{K}[\mathcal{E}]$.

In other words, $\mathbb{K}[\mathcal{E}]$ is defined as the quotient of $\mathbb{K}[x, y]$ by the following equivalence relation $\sim$:

$$p \sim q \text{ if and only if } \exists k \in \mathbb{K}[x, y] \text{ such that } p - q = k(y^2 - x^3 - ax - b).$$

Since the polynomials y^2 and $x^3 + ax + b$ are identified in $\mathbb{K}[\mathcal{E}]$, we can associate, to any equivalence class of $\mathbb{K}[\mathcal{E}]$, a canonical representative of the form $p_1 y + p_2$ ($p_1, p_2 \in \mathbb{K}[x]$), obtained by iteratively substituting y^2 by $x^3 + ax + b$. As such, instead of going through the path of formalizing ideals and ring quotients, we give a direct representation of $\mathbb{K}[\mathcal{E}]$ solely based on `{poly K}`, the type of univariate polynomials over `K`:

```
Inductive ecpoly := ECPoly of {poly K} * {poly K}.
Notation "[ecp p1 *Y + p2]" := (ECPoly p1 p2).
Coercion ecpoly_val (p : ecpoly) := let: ECPoly p := p in p.
```

The type `ecpoly` is simply a copy of `{poly K} * {poly K}`, an element `([ecp p1 *Y + p2] : ecpoly)` representing the class of the polynomial $p_1 y + p_2 \in \mathbb{K}[\mathcal{E}]$. We explicitly define the addition and multiplication, that are compatible with the one induced by the ring quotient.

For instance:

$$\begin{aligned}(p_1 y + p_2)(q_1 y + q_2) &= p_1 q_1 y^2 + (p_1 q_2 + q_1 p_2) y + p_2 q_2 \\ &= (p_1 q_2 + q_1 p_2) y + (p_1 q_1 (x^3 + ax + b) + p_2 q_2)\end{aligned}$$

leads to:

```
Notation XPoly := 'X^3 + A *: 'X + B.
Definition dotp p q : {poly K} := p.2 * q.2 + (p.1 * q.1) * Xpoly.
Definition one := [ecp 0 *Y + 1].
Definition mul p q :=
  locked [ecp (p.1 * q.2 + p.2 * q.1) *Y + (dotp p q)].
```

where `.1` and `.2` resp. stand for the first and second projections.

Unfolding the definitions we prove that the operations satisfy the properties of a commutative ring.

Remark. For the construction of the ring $\mathbb{K}[\mathcal{E}]$, we could have used a more general approach, relying on the general definition of a ring quotient by an ideal — a basic construction of commutative algebra. We would have obtained for free that $\mathbb{K}[\mathcal{E}]$ is an integral domain as the ring quotient by a prime ideal. Moreover, the quotient library of Ssreflect is built s.t. it would have been possible to choose our canonical representatives and to stick to the $y \cdot p(x) + q(x)$ representation. The only extra work would have been to link the ring operations inherited from generic construction to the ones directly defined on the canonical representatives. For example, for the multiplication, this would amount to prove that:

$$\begin{aligned}&[(y \cdot p_1 + q_1)] \cdot [y \cdot p_2 + q_2)] = \\ &\qquad [y \cdot (p_1 q_2 + q_1 p_2) + (p_1 q_1 (x^3 + ax + b) + p_2 q_2)],\end{aligned}$$

where $[x]$ denotes the class of x in the quotient.

However, the Ssreflect library comes with very few results on commutative algebra — for example, even the definition of what a ring ideal is was missing at that time! Constructing a mathematical component for commutative algebra was out of scope of this work and we hence decided to go to the elementary construction we just presented. Therefore the proofs are less abstract. For example, to prove that $\mathbb{K}[\mathcal{E}]$ is an integral domain becomes a little more technical, since it requires proving explicitly that

$$(p_1 y + p_2)(q_1 y + q_2) = 0 \implies p_2 q_2 = 0 \vee p_1 q_1 = 0.$$

Yet, after demonstrating the lemma `idomainAxiom` we are able to equip the type `ecpoly` with an `integralDomain` structure.

```
Lemma idomainAxiom: forall p q, p * q = 0 -> (p == 0) || (q == 0).
```

The proof is based on the following idea: We can prove by contradiction that the polynomial $y^2 - (x^3 + ax + b)$ is irreducible and therefore that $\mathbb{K}[\mathcal{E}]$ is an integral domain: Indeed, if we can factor $y^2 - (x^3 + ax + b) = (y - p(x))(y - q(x))$ then p and q have necessarily the same degree, which is absurd since their product is a polynomial of degree 3.

We then use the `fraction` [Coh13] library to built the `{fraction ecpoly}` type representing $\mathbb{K}(\mathcal{E})$, the field of fractions over $\mathbb{K}[\mathcal{E}]$.

Evaluation of polynomials of $\mathbb{K}[\mathcal{E}]$ on points is naturally defined on the canonical representatives. Note that while we are interested on evaluation on curve points, evaluation is defined on all finite points in order to remain as general as possible. Nevertheless, several lemmas concerning evaluation are true only concerning points that belong to the curve.

```
Definition eceval p (x y : K) := p.1.[x] * y + p.2.[x].
Notation "p .[ x , y ]" := (eceval p x y).
```

To pursue with the definition of the *order* in the following section, three notions are needed: the *conjugate*, the *norm* and the *degree*.

Definition 3.3 (Conjugate)**.** *Let $p \in \mathbb{K}[\mathcal{E}]$ of canonical representative $p_1(x)y + p_2(x)$. The* conjugate *of p, written $\bar{p}$, is defined as $\bar{p} = -p_1(x)y + p_2(x)$.*

Definition 3.4 (Norm)**.** *Let $p \in \mathbb{K}[\mathcal{E}]$ of canonical representative $p_1(x)y + q_1(x)$. The* norm *of p, written $n(f)$, is defined as $n(p) = p\bar{p}$.*

Remark that $n(p) = p_2(x)^2 - p_1(x)^2(x^3 + ax + b)$ is a polynomial on x. The *degree* of $n(p)$ is simply defined to be the degree of the polynomial $n(p) = p_2(x)^2 - p_1(x)^2(x^3 + ax + b)$ on x. For example the degree of y is 2. The definitions of the *conjugate*, *norm* and *degree* are then simply translated from their textbook counterpart on inhabitants of the type `ecpoly`:

```
Definition conjp p : {ecpoly E} := [ecp -p.1 *Y + p.2].
Definition normp p : {poly K} := dotp p (conjp p).
Definition degree p := size (normp p).
```

Lemmas concerning properties of all these notions are straightforward such as:

```
Lemma normpE p : normp p = (p.2)^+2 - (p.1)^+2 * (Xpoly E).

Lemma degree_mul_id p q, p * q != 0 ->
```

```
  degree (p * q) = (degree p + degree q).-1.

Lemma degree_add_max (p q : ecpoly E): degree p != degree q ->
  degree (p + q) = maxn (degree p) (degree q).

Lemma norm_eq0 p : (normp p == 0) = (p == 0).
```

The *degree* of a rational function $\frac{f}{g} \in \mathbb{K}(\mathcal{E})$ is defined as $\deg(\frac{f}{g}) = \deg(f) - \deg(g)$. It is easy to see that it is well defined and stable on fractions. This definition is simply translated from its textbook counterpart in Coq:

```
Definition fdegree_r (f : {ratio {ecpoly E}}) :=
  if \n_f == 0 then 0 else degree \n_f - degree \d_f.

Definition fdegree :=
  lift_fun1 {fraction {ecpoly E}} fdegree_r.
```

Order and evaluation of rational functions

In complex analysis, the *zeros* and *poles* of functions, and their *order* of vanishing are notions related to analytic functions and their Laurent expansion; while in abstract algebra, they refer to algebraic varieties and discrete valuation rings [Ful89]. For our formalization, we follow the more elementary definitions given in [Gui10]. More precisely, the evaluation of a function $f \in \mathbb{K}(\mathcal{E})$ at a point $P = (x_P, y_P) \in \mathcal{E}$ is defined as follows:

Definition 3.5. *A rational function $f \in \mathbb{K}(\mathcal{E})$ is said to be regular at $P = (x_P, y_P)$ if there exists a representative g/h of f such that $h(x_P, y_P) \neq 0$. If f is regular at P, the evaluation of f at P is the value $f(P) = \frac{g(x_P,y_P)}{h(x_P,y_P)}$, which is independent of the representative of f. If f is not regular at P, then P is called a pole of f and the evaluation of f at P is defined as $f(P) = \infty$.*

Evaluation for polynomials is extended at the point at infinity:

Definition 3.6. *Let $f \in \mathbb{K}(\mathcal{E})$ and let $\frac{n}{d}$ be a representative of f. Then,*

$$f(\mathcal{O}) = \begin{cases} 0 & \textit{when } \deg(n) < \deg(d) \\ \infty & \textit{when } \deg(n) > \deg(d) \\ \alpha_n/\alpha_d & \textit{when } \deg(n) = \deg(d) \end{cases}$$

where α_n (resp. α_d) is the coefficient of the higher degree term of n (resp. d).

Remark. Being not analytic, such definitions are difficult to formalize as-is in Coq. For example, concerning evaluation at finite points, one has to provide

a representative matching the definition and it is not clear how to find one. Moreover, the function is defined on fractions which means that it has to be stable on a fraction class. Those two details have to be taken into account in order to formalize it in Coq.

Fortunately, using *uniformizers*, which is a notion from algebraic geometry, we can computationaly decompose every fraction of $\mathbb{K}(\mathcal{E})$ in a canonical form which allows us compute the evaluation at some point, as defined above. Furthermore, this canonical form, which we will call *decomposition*, allows us compute the *order* of vanishing of the function at that point, which is essential in what follows. We have chosen this way to proceed and to define evaluation.

In what follows, we explain in details how the following extra notions allow us to decompose any rational function in some canonical representative:

Definition 3.7. *A function $u \in \mathbb{K}(\mathcal{E})$ is called a uniformizer at $P \in \mathcal{E}(\mathbb{K})$ if i) $u(P) = 0$, and ii) every non-zero function $f \in \mathbb{K}(\mathcal{E})$ can be written in the form $f = u^v g$ with $g(P) \neq 0, \infty$ and $v \in \mathbb{Z}$.*

The exponent v is independent from the choice of the uniformizer and is called the order *of f at P, a quantity denoted by* $\mathrm{ord}_f(P)$.

Lemma 3.1. *There exists a uniformizer for every point on the curve.*

To get an intuition of the previous definitions, one can make a parallel with the notion of multiplicity for roots of univariate polynomials or with the notion of zeros and poles in $\mathbb{K}(x)$.

For instance, let us first consider the ring of polynomials $\mathbb{K}[x]$. Let p be a polynomial in $\mathbb{K}[x]$ and r be an element of $\mathbb{K}$. We can factorize p as $p = (x-r)^m q$ such that $m \in \mathbb{N}$ and $q(r) \neq 0$. The exponent m is the multiplicity of p at r. The multiplicity of r is 1 for the polynomial factor $(x - r)$. Evaluation and multiplicity are closely related: r is a root of p iff $m > 0$.

In an analogous way, we can consider the field of fractions $\mathbb{K}(\mathcal{E})$. Let P be in $\mathcal{E}$ and f in $\mathbb{K}(\mathcal{E})$. Then, one can always write f in the form $f = u^v g$ with $v \in \mathbb{Z}$ uniquely defined and P neither a zero nor a pole of g ($g(P) \neq 0, \infty$). The exponent v is the order of f at P. (Here, the function u corresponds to the polynomial factor $(x - r)$ for univariate polynomials) If $v > 0$ then P is a zero for f, and if $v < 0$ then P is a pole for f.

An interesting fact, used in algebraic geometry, is that the order is a discrete valuation for $\mathbb{K}(\mathcal{E})$, which makes $\mathbb{K}[\mathcal{E}]$ a discrete valuation ring — a property that we demonstrate later:

— for all $P \in \mathcal{E}$ and for all non zero $f_1, f_2 \in \mathbb{K}[\mathcal{E}]$,

$$\mathrm{ord}_{f_1 * f_2}(P) = \mathrm{ord}_{f_1}(P) + \mathrm{ord}_{f_2}(P),$$

— for all $P \in \mathcal{E}$ and for all non zero $f_1, f_2 \in \mathbb{K}[\mathcal{E}]$,

$$\mathrm{ord}_{f_1 + f_2}(P) \geq \max\{\mathrm{ord}_{f_1}(P), \mathrm{ord}_{f_2}(P)\}.$$

The proof of Lemma 3.1 is constructive and gives all the necessary material to define the notions of the uniformizer and the order. Indeed, the problem of evaluating a rational function at a given point is reduced to finding a proper decomposition of the rational function, based on the following property:

Lemma 3.2. *Let $f \in \mathbb{K}(\mathcal{E})^*$ and P be a point on the curve. Let u be a uniformizer at P and $u^d g$ a decomposition of f respecting the conditions of Definition 3.7. Then:*

$$f(P) = \begin{cases} 0 & \text{when } d > 0 \\ \infty & \text{when } d < 0 \\ g(P) & \text{when } d = 0. \end{cases}$$

Computing such a decomposition, for every rational function f and point on the curve P, gives a decomposition $u^o \cdot \frac{n}{d}$ where u and o are constructively defined and $n(P) \neq 0$ and $d(P) \neq 0$ if P is finite, or $\deg(n) = \deg(p)$ otherwise. We first define a family of rational functions $\{u_P\}_P$ s.t. for any point of the curve P, u_P will be a uniformizer at P:

$$u_P = \begin{cases} x - x_P & \text{when } P = (x_P, y_P) \text{ and } y_P \neq 0 \\ y & \text{when } P = (x_P, 0) \\ \frac{x}{y} & \text{when } P = \mathcal{O}. \end{cases}$$

The family of uniformizers is translated in our setting as:

```
Definition unifun_fin (x y : K) : ecpoly :=
    if y == 0
    then [ecp 1 *Y + 0 ]
    else [ecp 0 *Y + ('X - x)].

  Definition unifun (P : point K) :=
    match P with
    | (| x, y |) => (unifun_fin x y)
    | EC_Inf => 'X / [ecp 1 *Y + 0]
    end.
```

To demonstrate how we obtain the decomposition of a function, let us consider the first case of a finite point $P = (x_P, y_P)$ with $y_P \neq 0$. In this case, the uniformizer u_P is equal to $(x - x_P)$. Let $f \in \mathbb{K}[\mathcal{E}]^*$ of representative $p(x) + yq(x)$. We want to compute $\mathrm{ord}_P(f)$ and the decomposition $f = u^d \cdot \frac{n}{d}$.

P is not a zero of $f(x, y)$

Then $f(x, y) = (x - x_P)^0 \cdot \frac{f(x,y)}{1}$.

P is a zero of $f(x, y)$ and $p(x_P) \neq 0$ or $q(x_P) \neq 0$

Let μ be the multiplicity of x_P in $\mathrm{norm}(f)$, and $r(x)$ a element of $\mathbb{K}[x]$ s.t. $\mathrm{norm}(f) = (x - x_P)^\mu r(x)$. Then:

$$f = (x - x_P)^\mu \left(\frac{r(x)}{p(x) - yq(x)}\right)$$

with $r(x_P) \neq 0$ and $p(x_P) - y_P \cdot q(x_P) \neq 0$.

P is a zero of $f(x, y)$ and $p(x_P) = q(x_P) = 0$

Let μ_p (resp. μ_q) be the multiplicity of x_P in p (resp. in q), and μ as:

$$\mu = \begin{cases} \mu_q & \text{if } \mu_p = 0 \\ \mu_p & \text{if } \mu_q = 0 \\ \min(\mu_p, \mu_q) & \text{otherwise.} \end{cases}$$

Let $p_1(x)$ and $q_1(x)$ be the elements of $\mathbb{K}[x]$ s.t. $p(x) = (x - x_P)^\mu p_1(x)$ and $q(x) = (x - x_P)^\mu q_1(x)$. Then:

$$f = (x - x_P)^\mu (p_1(x) + yq_1(x))$$

with $p_1(x_P) \neq 0$ or $q_1(x_P) \neq 0$. We are back to the previous case.

The function `poly_ordreg` returns the decomposition at finite points with $y \neq 0$. It takes the coordinates of a point (x, y) and a polynomial of type `ecpoly` and returns a triple `(o, n, d)` such that `p = (unifun (|x, y|))^o * (n // d)`:

```
Definition poly_ordreg
  (x y : K) (p : ecpoly) : nat * (ecpoly * ecpoly)
:=
  let: (d, (pp1, pp2)) := mudiv_join x p.1 p.2 in
  let: p' := [ecp pp1 *Y + pp2] in

  if p'.[x, y] == 0 then
    let d' := \mu_x (normp p') in
    let g := (normp p') / ('X - x)^d' in
      ((d + d'), ([ecp 0 *Y + g], (conjp p')))
  else
    (d, (p', 1)).
```

In the same way, we define a decomposition function, called `poly_ordspec`, for finite points with $y = 0$ and `poly_orderinf` for the point at infinity. Finally, the function `poly_order` computes the decomposition for all points of the projective plane, making sure that points that do not belong to the curve are projected to the triplet $(0, 0, 1)$ as a convention.

```
Definition poly_orderfin (x y : K) (f : ecpoly) :=
    if y == 0 then poly_ordspec x f else poly_ordreg x y f.
```

```
Definition poly_orderinf (p:ecpoly) : int * (ecpoly * ecpoly) :=
    let d := (degree p).-1 in
    (-d, ('X^+d * p, [ecp 1 *Y + 0] ^+ d)).

Definition poly_order
  (p : ecpoly) (ecp : point K) : int * (ecpoly * ecpoly)
:=
    if (p == 0) || ~~(oncurve ecp) then (0, (0, 1)) else
      if ecp is (| x, y |) then
        let: (n, (g, h)) := poly_orderfin x y p in (n, (g, h))
      else poly_orderinf p.
```

We continue by proving that the above decomposition is correct and unique. The correctness of the decomposition is expressed by the predicate `uniok`. Note that for the infinite point, the property that the point at which the decomposition occurs is not a zero of `n` or `d` has been replaced by `degree n == degree d`.

```
Definition uniok_fin (u f : {fraction ecpoly}) x y o (n d : ecpoly) :=
 [&& f == u ^ o * (n // d), n.[x, y] != 0 & d.[x, y] != 0].

Definition uniok_inf (u f : {fraction ecpoly}) o (n d : ecpoly) :=
 [&& f == u ^ o * (n // d), n // d != 0 & (degree n == degree d)].

Definition uniok (ecp : point K) o (n d : ecpoly) :=
  if ecp is (| x, y |)
  then uniok_fin x y o n d
  else uniok_inf o n d.

Lemma poly_order_correct:
  forall (f : ecpoly) (p : point), f != 0 -> oncurve p ->
    let: (o, (g1, g2)) := poly_order f p in
      uniok (unifun p) f p o g1 g2.

Lemma uniok_uniq:
  forall f p, f != 0 -> oncurve p ->
    forall o1 o2 n1 n2 d1 d2,
        uniok (unifun p) f p o1 n1 d1
      -> uniok (unifun p) f p o2 n2 d2
      -> (o1 == o2) && (n1 // d1 == n2 // d2).
```

Proving that all the lifted functions are stable by quotienting allows us to lift all the proved properties over $\mathbb{K}[\mathcal{E}]$ to $\mathbb{K}(\mathcal{E})$. For instance, the order on `{fraction ecpoly}` is defined as:

```
Definition orderf (f : {ratio ecpoly}) p : int :=
```

```
  if \n_f == 0 then 0 else
    (poly_order \n_f p).1 - (poly_order \d_f p).1.

Definition order f p :=
 (lift_fun1 {fraction ecpoly} (orderf^~ p)) f.

Lemma order_correct (n d : ecpoly) ecp o1 o2 n1 n2 d1 d2:
      oncurve ecp
   -> uniok (unifun ecp) n ecp o1 n1 d1
   -> uniok (unifun ecp) d ecp o2 n2 d2
   -> [&& order (n // d) ecp == o1 - o2
        & uniok (unifun ecp) (n // d) ecp (o1 - o2) (n1 * d2) (n2 * d1)].
```

Then, we prove that the order is a discrete valuation, making $\mathbb{K}[\mathcal{E}]$ a discrete valuation ring:

```
Lemma order_mul f1 f2 ecp: (f1 * f2) != 0 ->
 order (f1 * f2) ecp = order f1 ecp + order f2 ecp.

Lemma order_add_leq f g ecp: f * g != 0 -> f + g != 0
 -> order (f + g) ecp >= Num.min (order f ecp) (order g ecp).
```

We can then formalize Definition 3.5 by a simple case analysis over the *order*, relying on the decomposition of rational functions: Given, a rational function `f` and a point P, the function `decomp` returns the (n, d) part of the function's decomposition $f = u^d \cdot \frac{n}{d}$. The function `leadc` computes the leading coefficient of the highest-power term of a polynomial. Next, `ecreval` is the extension of `eceval` for all points on the curve (including the infinite point) and finally `eval` is the evaluation for elements of the field of fractions $\mathbb{K}(\mathcal{E})$.

```
Definition decomp
  (f : {fraction ecpoly}) (ecp : point K) : (ecpoly * ecpoly)
:=
  let f := repr f in
      if (\n_f == 0) || ~~(oncurve ecp) then (0, 1) else
        let: (n1, d1) := (poly_order \n_f ecp).2 in
        let: (n2, d2) := (poly_order \d_f ecp).2 in
          (n1 * d2, n2 * d1).

Definition leadc (p : ecpoly) :=
  if (degree [ecp p.1 *Y + 0] > degree [ecp 0 *Y + p.2])
  then lead_coef p.1
  else lead_coef p.2.

Definition ecreval (p : ecpoly) ecp :=
```

```
  if oncurve ecp then
    if ecp is (|x, y|) then p.[x, y] else leadc p
  else 0.

Definition eval (f : {fraction ecpoly}) ecp : (gproj K) :=
  match order f ecp with
  | _.+1 => 0 | Negz _ => GP_Inf | 0 =>
      (ecreval (decomp f ecp).1 ecp / ecreval (decomp f ecp).2 ecp)
  end.
```

Note that the evaluation returns an element of the projective line, formalized by the parametric type `gproj`:

```
Inductive gproj (G : Type) : Type := GP_Finite of G | GP_Inf.
```

Addition and multiplication are defined naturally on the projective line, with the difference that multiplication of 0 with ∞ is defined to be 0 as a convention.

The key lemma of this section is the following:

Lemma 3.3. *A rational function $f \in \mathbb{K}(\mathcal{E})$ has a finite number of poles and zeros. Moreover, assuming that $\mathbb{K}$ is algebraically closed, $\sum_{P \in \mathcal{E}}(\mathrm{ord}_P(f)) = 0$.*

This lemma is central to the construction of the isomorphism between an elliptic curve and its Picard group, as it will be described later. The proof is based on the fact that a finite point on curve $P = (x_P, y_P)$ is a zero of a polynomial $p \in \mathbb{K}[\mathcal{E}]$ if and only if x_P is a root of the norm n_p of p. Indeed,

$$\begin{aligned} n_p(x_P) = 0 &\implies p(x_P, y_P)\bar{p}(x_P, y_P) = 0 \\ &\implies p(x_P, y_P) = 0 \vee \bar{p}(x_P, y_P) = 0 \end{aligned}$$

since in $\mathbb{K}$ there are no divisors of zero. In other words, if x_P is a root of the norm n_p then, any (x_P, y_P) such that $y_P^2 = x_P^3 + ax_P + b$ is either a zero of p or of its conjugate $\bar{p}$. The norm, as a univariate polynomial, has a finite number of roots, hence a polynomial $p \in \mathbb{K}[\mathcal{E}]$ has a finite number of zeros too. Let p/q be a representation of a rational function $f \in \mathbb{K}(\mathcal{E})$. Then f has at most as many zeros as p and as many poles as the zeros of q, hence f has a finite number of zeros and poles.

To compute the zeros of a polynomial $p \in \mathbb{K}[\mathcal{E}]$, one needs to compute the roots $x_1, x_2, \cdots$ of its norm n_p and then find the corresponding y_i such that $(x_i, y_i) \in \mathcal{E}$ and $p(x_i, y_i) = 0$. The function `ecroots` formalizes this procedure:

```
Definition ecroots f : seq (K * K) :=
  let forx := fun x =>
  let sqrts := roots ('X ^+ 2 - (Xpoly.[x])%:P) in
```

```
  [seq (x, y) | y <- sqrts & f.[x, y] == 0]

  in undup (flatten ([seq forx x | x <- roots (normp f)])).

Lemma ecroot_normp f x y : oncurve (|x, y|) ->
    ((normp f).[x] == 0) = ((f.[x, y] == 0) || (f.[x, -y] == 0)).
```

It relies on the function `roots` that returns the roots of an univariate polynomial. Note that for all polynomials $f \in \mathbb{K}[\mathcal{E}]$, if (x, y) is a zero of f, then $(x, -y)$ is a zero of $\bar{f}$. Using the decomposition of f, we can easily demonstrate that $\mathrm{ord}_P(f) = \mathrm{ord}_{-P}(\bar{f})$. Then,

$$\sum_{P \in \mathcal{E} | P \text{ finite}} \mathrm{ord}_P(f) = \sum_{-P \in \mathcal{E} | -P \text{ finite}} \mathrm{ord}_{-P}(\bar{f}) = \sum_{P \in \mathcal{E} | P \text{ finite}} \mathrm{ord}_P(\bar{f}).$$

Since the order is multiplicative, we have $\mathrm{ord}_P(n_f) = \mathrm{ord}_P(f) + \mathrm{ord}_P(\bar{f})$. Hence,

$$\begin{aligned}\sum_{P \in \mathcal{E} | P \text{ finite}} \mathrm{ord}_P(n_f) &= \sum_{P \in \mathcal{E} | P \text{ finite}} \mathrm{ord}_P(f) + \sum_{P \in \mathcal{E} | P \text{ finite}} \mathrm{ord}_P(\bar{f}) \\ &= 2 \cdot \sum_{P \in \mathcal{E} | P \text{ finite}} \mathrm{ord}_P(f).\end{aligned}$$

Now, $\mathbb{K}$ being algebraically closed, the norm of f can be decomposed as $n_f(x) = (x - x_1)^{k_1}(x - x_2)^{k_2} \cdots (x - x_r)^{k_r}$. We distinguish two cases: i) if (x_i, y_i) is a regular point ($y_i \neq 0$), then $\mathrm{ord}_{(x_i, y_i)}(n_f) = k_i$, and ii) if (x_i, y_i) is a special point ($y_i = 0$) then $\mathrm{ord}_{(x_i, y_i)}(n_f) = 2 \cdot k_i$. As a result,

$$\begin{aligned}\sum_{P \in \mathcal{E} | P \text{ finite}} \mathrm{ord}_P(n_f) &= \sum_{(x,y) \in \mathcal{E}} \mathrm{ord}_{(x,y)}(n_f) \\ &= \sum_{\substack{(x,y) \in \mathcal{E} \\ 0 < y}} \mathrm{ord}_{(x,y)}(n_f) + \sum_{\substack{(x,y) \in \mathcal{E} \\ 0 < y}} \overbrace{\mathrm{ord}_{(x,-y)}(n_f)}^{\mathrm{ord}_{(x,y)}(n_f)} + \sum_{(x,0) \in \mathcal{E}} \mathrm{ord}_{(x,0)}(n_f) \\ &= 2 \cdot \sum_i k_i.\end{aligned}$$

As a result, $\sum_{(x,y) \in \mathcal{E}} \mathrm{ord}_{(x,y)}(f) = \deg(f)$, and $\sum_{P \in \mathcal{E}} \mathrm{ord}_{(x,y)}(f) = 0$.

3.2.2 Principal Divisors

From now on, we assume that $\mathbb{K}$ is algebraically closed. Principal divisors are introduced as a tool for describing the zeros and poles of rational functions on an elliptic curve:

Definition 3.8 (Principal divisors)**.** *Given $f \in \mathbb{K}(\mathcal{E})$, $f \neq 0$, the* principal divisor $\mathrm{Div}(f)$ *of f is defined as the formal (finite) sum:*

$$\mathrm{Div}(f) = \sum_{P \in \mathcal{E}} (\mathrm{ord}_P(f))(P).$$

Note that $\mathrm{Div}(f)$ *is well defined because a rational function has only finitely many zeros and poles. We write* $\mathrm{Prin}(\mathcal{E})$ *for the set of principal divisors.*

The set $\mathrm{Prin}(\mathcal{E})$ forms a subgroup of $\mathrm{Div}(\mathcal{E})$, the set of formal sums over $\mathcal{E}$, a notion that we define now.

Definition 3.9. *A* divisor *on an elliptic curve* $\mathcal{E}$ *is a formal sum of points*

$$D = \sum_{P \in \mathcal{E}} n_P(P),$$

where $n_P \in \mathbb{Z}$ *and only finitely many of them are nonzero. In other words, a divisor is any expression taken in the free abelian group generated over* $\mathcal{E}(\mathbb{K})$*. The* domain *of* D *is* $\mathrm{dom}(D) = \{P \mid n_P \neq 0\}$ *and its* degree *is* $\deg(D) = \sum_{P \in E} n_P$*. For any point* P*, the* coefficient *of* P *in* D *is* $\mathrm{coeff}(P, D) = n_P$*.*

We write $\mathrm{Div}(\mathcal{E})$ *for the set of divisors on* $\mathcal{E}$*, and* $\mathrm{Div}^0(\mathcal{E})$ *its subgroup composed of divisors of degree* 0*.*

The set of divisors on $\mathcal{E}$ is an abelian group. The sum of two divisors is defined as the point wise addition $\sum_{P \in \mathcal{E}} a_k(P_k) + \sum_{P \in \mathcal{E}} b_k(P_k) = \sum_{P \in \mathcal{E}} (a_k + b_k)(P_k)$ whereas the zero divisor is the unique divisor with all its coefficient set to 0.

Based on the `quotient` libraries of SSREFLECT, we develop the theory of free abelian groups. Let `T` be a type. We first define the type of *pre-free group* as the collection of all sequences `s` of type `int * T` s.t. no pair of the form `(0, _)` can appear in `s` and for any `z : T`, a pair of the form `(_, z)` can appear at most once in `s`.

```
Definition reduced (D : seq (int * T)) :=
    (uniq [seq zx.2 | zx <- D])
  && (all [pred zx | zx.1 != 0] D).

Record prefreeg : Type := mkPrefreeg {
  seq_of_prefreeg : seq (int * T);
  _ : reduced seq_of_prefreeg
}.
```

The intent of `prefreeg` is to give a unique representation of a free-group expression, up to the order of the coefficients. For instance, if $D = k_1x_1 + \cdots + k_nx_n$ (with all the x_i's pairwise distinct and all the k_i's in $\mathbb{Z}^*$), then the `reduced` sequence `s = [:: (k_1, x_1), ..., (k_n, x_n)]`, or any sequence equal up to a permutation to `s`, is a valid representation of D. The type `freeg` of free-groups is then obtained by quotienting `prefreeg` by the `perm_eq` equivalence relation (where `perm_eq xs ys` is true if the lists `xs ys` are equal up to permutation).

From there, we equip the type `freeg` with a group structure (the operation is noted additively), and define all the usual notions related to free groups (domain, coefficient, degree, ...). For instance, assume `G : zmodType` (G is a $\mathbb{Z}$-module) and `f : T -> G`. Then, `f` defines a unique group homomorphism from `freeg` to `G` that can be defined as follows:

```
Definition prelift (D : seq (int * T)) : G :=
  \sum_(x <- D) (f x.2) * x.1.

Definition lift (s : prefreeg T) : G := prelift s.
Definition fglift (D : {freeg T}) := lift (repr D).
```

One can check that the `fglift` function defines the homomorphism

$$\sum_{(z,x)\in D} z f(x)$$

The coefficient `coeff` and degree `deg` functions can be then defined as:

```
Definition coeff (t : T) (D : {freeg T}) :=
  fglift (fun x => (x == t)) D.

Definition deg (D : {freeg K}) : int :=
  fglift (fun x => 1) D.
```

The Group of Principal Divisors

One can easily check that $\mathrm{Prin}(\mathcal{E})$ is a subgroup of $\mathrm{Div}^0(\mathcal{E})$. Indeed, $\forall f, g \in \mathbb{K}(\mathcal{E})$ i) $\deg(\mathrm{Div}(f)) = 0$ by Lemma 3.3, and ii) since the order function is multiplicative ($\mathrm{ord}_p(f/g) = \mathrm{ord}_p(f) - \mathrm{ord}_p(g)$), we have $\mathrm{Div}(f/g) = \mathrm{Div}(f) - \mathrm{Div}(g)$. Moreover, it is now clear that the coefficients associated in $\mathrm{Div}(f)$, to each point P, is the order of the function f at P, highlighting the fact that a divisor wraps up the zeros and poles of f.

Formally, we define principal divisors for polynomials on the curve with the function `ecdivp`:

```
Definition ecdivp (f : ecring) : {freeg (point)} :=
  \sum_(p <- ecroots f)
      << (order f (p.1, p.2)) * (p.1, p.2) >>
    + << order f EC_Inf * EC_Inf >>.
```

where `<< z * P >>` stands for the divisor $z(P)$ and the function `ecroots` takes a polynomial of $\mathbb{K}[\mathcal{E}]$ and returns the list of its finite zeros as explained in the previous section.

Next, we lift the definition of principal divisors to $\mathbb{K}(\mathcal{E})$, prove its correctness and recast the key Lemma 3.3 (deg_ecdiv_eq0):

```
Notation "\n_f" := (numerator f).
Notation "\d_f" := (denominator f).

Definition ecdiv_r (f : {ratio ecring}) :=
  if \n_f == 0 then 0 else (ecdivp \n_f) - (ecdivp \d_f).

Definition ecdiv := lift_fun1 {fraction ecring} ecdiv_r.

Lemma ecdiv_coeffE (f : {fraction ecring}) p:
  coeff p (ecdiv f) = order f p.

Lemma deg_ecdiv_eq0 (f : {fraction ecring}): deg (ecdiv f) = 0.
```

3.2.3 Divisor of a line

Before moving to the definition of the Picard group, we characterize the divisors of some specific rational functions. These divisors will later help formalize the construction of the Picard group:

Definition 3.10. *A line $l \in \mathbb{K}(\mathcal{E})$ is any rational function of the form $l(x, y) = ax + by + c$ with $a, b, c \in \mathbb{K}$ not all zero.*

For instance, if (PQ) is the line intersecting the curve at P and Q, then we know that (PQ) intersects $\mathcal{E}$ at exactly three points (counting multiplicities): P, Q and $P \boxdot Q$. Assuming that P, Q and $P \boxdot Q$ are all finite, these three points are the unique zeros of the rational function l associated to (PQ) and $\mathrm{Div}(l) = (P) + (Q) + (P \boxdot Q) - 3(\mathcal{O})$. If (PQ) is the line tangent at P which passes from Q, then $\mathrm{Div}(l) = 2(P) + (Q) - 3(\mathcal{O})$. The line tangent at P which passes from no other point on the curve, is a special case (P is an inflexion point) and its divisor is $\mathrm{Div}(l) = 3(P) - 3(\mathcal{O})$. Moreover, the above relation still holds when one or several of these three points are equal to $\mathcal{O}$: For instance, $\mathrm{Div}(x - x_P) = (P) + (-P) - 2(\mathcal{O})$, where $x - x_P$ is the line intersecting $\mathcal{E}$ at P, $-P$ and $\mathcal{O}$.

3.2.4 The Picard Group

Since principal divisors form a subgroup of zero degree divisors, we can define the quotient of zero-degree divisors modulo principal divisors, called the Picard group or the divisor class group:

Definition 3.11. *The Picard group $\mathrm{Pic}(\mathcal{E})$ is the group quotient $\mathrm{Div}(\mathcal{E})/\mathrm{Prin}(\mathcal{E})$. The Picard group of zero degree $\mathrm{Pic}^0(\mathcal{E})$ is the group quotient $\mathrm{Div}^0(\mathcal{E})/\mathrm{Prin}(\mathcal{E})$.*

Note that the degree is well defined on the divisor class group since if $D_1 = D_2 + \mathrm{Div}(f)$ then $\deg D_1 = \deg D_2 + \deg(\mathrm{Div}(f)) = \deg D_2 + 0 = \deg D_2$.

In other words, $\mathrm{Pic}(\mathcal{E})$ is defined as the quotient of $\mathrm{Div}(\mathcal{E})$ by the following equivalence relation $\sim$:

$$D_1 \sim D_2 \text{ if and only if } \exists f \in \mathbb{K}(\mathcal{E}) \text{ such that } \mathrm{Div}(f) = D_1 - D_2.$$

We formalize this notion as follows:

```
Definition ecdeqv D1 D2 :=
  (exists f : {fraction ecring}, ecdiv f = D1 - D2).
Notation "D1 :~: D2" := (ecdeqv D1 D2).
```

The Picard group that is of interest to our development is $\mathrm{Pic}^0(\mathcal{E})$. We do not give a direct construction of $\mathrm{Pic}^0(\mathcal{E})$ but instead prove that any class of $\mathrm{Pic}^0(\mathcal{E})$ can be represented by a divisor of the form $(P) - (\mathcal{O})$.

The construction of this representative is based on a procedure called *Linear Reduction*. Assume that P and Q are two finite points of $\mathcal{E}(\mathbb{K})$. We know that the divisor of the line l intersecting $\mathcal{E}$ at P and Q is $\mathrm{Div}(l) = (P)+(Q)+(P\boxdot Q)-3(\mathcal{O})$. Likewise, the divisor of the line l' intersecting $\mathcal{E}$ at $P\boxdot Q$ and $-(P\boxdot Q)$ $(= P+Q)$ is $\mathrm{Div}(l') = (P+Q) + (P \boxdot Q) - 2(\mathcal{O})$. Hence,

$$\begin{aligned}\mathrm{Div}(l/l') &= \mathrm{Div}(l) - \mathrm{Div}(l') \\ &= (P) + (Q) - (P+Q) - (\mathcal{O}),\end{aligned}$$

and $(P) + (Q) \sim (P+Q) + (\mathcal{O})$.

Iterating this procedure, we can reduce any divisor of the form:

$$(P_1) + \cdots + (P_n) - (Q_1) - \cdots - (Q_k) + r(\mathcal{O})$$

to an equivalent one $(P) - (Q) + r'(\mathcal{O})$, with $r' \in \mathbb{Z}$. Using one more time the same construction, one can show that $(P) - (Q) + n'(\mathcal{O})$ is equivalent to $(P-Q) + n''(\mathcal{O})$ where $n', n'' \in \mathbb{Z}$:

Indeed, the divisor of the line l_1 intersecting $\mathcal{E}$ at P and $-P$ is $\mathrm{Div}(l_1) = (P) + (-P) - 2(\mathcal{O})$. Likewise, the divisor of the line l_2 intersecting $\mathcal{E}$ at Q, $-P$ and $P-Q$ is $\mathrm{Div}(l_2) = (Q) + (-P) + (P-Q) + 3(\mathcal{O})$. Hence,

$$\begin{aligned}\mathrm{Div}(l_1/l_2) &= \mathrm{Div}(l_1) - \mathrm{Div}(l_2) \\ &= (P) - (Q) - (P-Q) + (\mathcal{O}),\end{aligned}$$

and $(P) - (Q) \sim (P-Q) + (\mathcal{O})$.

The `lr` function formally defines the linear reduction procedure:

```
Definition fgpos (D : {freeg K}) :=
  \sum_(p <- dom D | coeff p D > 0) coeff p D.
Definition fgneg (D : {freeg K}) :=
  \sum_(p <- dom D | coeff p D < 0) -(coeff p D).

Definition lr_r (D : {freeg point}) :=
  let iter p n := iterop _ n + p EC_Inf in
    \sum_(p <- dom D | p != EC_Inf) (iter p `|coeff p D|).

Definition lr (D : {freeg point}) : point :=
  let: (Dp, Dn) := (fgpos D, fgneg D) in
    lr_r Dp - lr_r Dn.

Lemma ecdeqv_lr D: all oncurve (dom D) ->
  D :~: << lr D >> + << deg D - 1 *g EC_Inf >>.
```

where `(Dp, Dn) := (fgpos D, fgneg D)` is the decomposition of D into its negative and positive parts.

The lemma `ecdeqv_lr` states that any divisor is equivalent to a divisor of the form $(P) + (\deg D - 1)(\mathcal{O})$. In the context of $\mathrm{Pic}^0(\mathcal{E})$, this means that any class contains a divisor of the form $(P) - (\mathcal{O})$ (recall that $\mathrm{Pic}^0(\mathcal{E})$ is a group quotient of $\mathrm{Div}^0(\mathcal{E})$ — the divisors of degree 0). In the next section, we end the construction of the Picard group by proving that at most one such representative can be found in each class of $\mathrm{Pic}^0(\mathcal{E})$.

Remark. Linear Reduction obviously presents an algorithmic aspect: it is a function that iterates a procedure on a divisor and produces an equivalent one. This was the first approach of formalizing an abstract elliptic curve algorithm, using our formalization. The algorithm of linear reduction could possibly be extended to (an abstract version of) the Miller algorithm [Mil86] to compute Weil pairings on elliptic curves.

Moreover, linear reduction as well as addition on elliptic curve points, they both reflect the geometrical vision that characterizes our development. While our development takes place in an algebraic setting based on the algebra libraries of Ssreflect the use of lines to define addition and to reduce divisors preserve the geometric aspect of elliptic curves that one finds in almost all mathematics textbooks.

3.3 Linking $\mathrm{Pic}(\mathcal{E})$ to $\mathcal{E}(\mathbb{K})$

In this section, we finish our formal construction of the Picard group and prove the existence of an isomorphism between $\mathrm{Pic}^0(\mathcal{E})$ and $\mathcal{E}(\mathbb{K})$. We start by

defining a canonical representative for the classes of $\mathrm{Pic}^0(\mathcal{E})$:

Lemma 3.4. *For every class of* $\mathrm{Pic}^0(\mathcal{E})$*, there exists a unique representative of the form* $(P) - (\mathcal{O})$ *with* $P \in \mathcal{E}(\mathbb{K})$*.*

The proof of this proposition works by contradiction: if not true, it allows us to construct an extension of $\mathbb{K}(x)$ of degree 2 that is isomorphic to itself. From Section 3.2.4, we already know that each class of $\mathrm{Pic}^0(\mathcal{E})$ contains one such representative. Assume now that $(P)-(\mathcal{O})$ and $(Q)-(\mathcal{O})$ are two representatives of a class of $\mathrm{Pic}^0(\mathcal{E})$ with $P \neq Q$. Such an assumption allows us to find a rational function $h \in \mathbb{K}(\mathcal{E})$ s.t. every rational function $f \in \mathbb{K}(\mathcal{E})$ can be expressed as a polynomial fraction of h. This implies that $\mathbb{K}(\mathcal{E})$ and $\mathbb{K}(x)$ are isomorphic. However, since $\mathbb{K}(\mathcal{E})$ is a field extension of degree 2 of $\mathbb{K}(x)$, such an h cannot exist. Hence, $P = Q$:

```
Lemma lr_uniq: << p >> :~: << q >> -> p = q.
```

Formalizing the above proposition turns out to be quite technical. Assuming the existence of a function $h \in \mathbb{K}(\mathcal{E})$, whose divisor is $(P) - (Q)$, we have to show that every rational function $f \in \mathbb{K}(\mathcal{E})$ can be expressed as a polynomial fraction of h (a fact that leads finally to a contradiction).

More precisely, for all points $R \in \mathcal{E}$, we consider the function $h_R = h - h(R)$ and we compute its divisor. For a fixed point $S \in \mathcal{E}$, h_S has the following properties:

— it is not a constant fraction, and

— its divisor is $(S) - (Q)$ if $S \neq Q$.

Then using function decomposition, any rational function such that Q is not in the domain of its divisor can be expressed as a polynomial fraction of h, which is a property that can be generalized for all rational functions in $\mathbb{K}(\mathcal{E})$.

Using the previous result, we can express x and y as polynomial fractions of h. But on the curve $\mathcal{E}$, these functions satisfy the curve equation $y^2 = x^3 + ax + b$, which contradicts the following proposition, whose proof is based on another technical result:

Lemma 3.5. *Let* $\mathcal{E} : y^2 = x^3 + ax + b$ *be a smooth Weierstrass curve on some field* $\mathbb{K}$*. If* $r, s \in \mathbb{K}(x)$ *satisfy* $r^2 = s^3 + as + b$*, then* r, s *are constant fractions.*

As will be explained below, this part of the proof is a direct consequence of the Riemann-Roch theorem. The Picard group $\mathrm{Pic}^0(\mathcal{E})$ can now be formally defined as the set of divisors of the form $(P) - (\mathcal{O})$. It remains to prove the existence of a bijection between $\mathrm{Pic}^0(\mathcal{E})$ and $\mathcal{E}(\mathbb{K})$. Namely, the function

$$\begin{aligned} \phi : \mathcal{E}(\mathbb{K}) &\to \mathrm{Pic}^0(\mathcal{E}) \\ P \quad &\mapsto [(P) - (\mathcal{O})] \end{aligned}$$

From the results of Section 3.2.4:

$$\begin{aligned}\phi(P_1) - \phi(P_2) &= [(P_1) - (\mathcal{O})] - [(P_2) - (\mathcal{O})] = [(P_1) - (P_2)] \\ &= [(P_1 - P_2) - (\mathcal{O})] = \phi(P_1 - P_2).\end{aligned}$$

In our formalization, we directly use the linear reduction function `lr` in place of ϕ^{-1}. For instance, we prove that `lr` commutes with the curve operations and maps $(P) - (\mathcal{O})$ to $P \in \mathbb{K}(\mathcal{E})$:

```
Lemma lrB: forall (D1 D2: {freeg point},
   deg D1 = 0 -> all oncurve (dom D1) ->
   deg D2 = 0 -> all oncurve (dom D2) ->
   lr (D1 - D2) = lr D1 - lr D2.

Lemma lrpi: forall p : point,
  oncurve p -> lr (<<p>> - <<EC_Inf>>) = p.
```

This allows us to transport the structure from $\mathrm{Pic}^0(\mathcal{E})$ to $\mathcal{E}(\mathbb{K})$, proving that $\mathcal{E}(\mathbb{K})$ is a group.

Remark. A part of the formalization of lemma 3.5 turned out to be quite technical, the proof involving matrices and polynomials, and being more than 800 lines of code. This part mainly involved a detailed translation of the pencil-and-paper proof into Ssreflect code while no structures were defined and no design decisions were taken. Nevertheless, it led to the development of theory about evaluation of univariate polynomial fractions and about composition of polynomial fractions with rational functions of $\mathbb{K}(\mathcal{E})$. It also led to the formalization of some additional lemmas concerning polynomials and matrices. This technical part of the proof could have been avoided if we had used the Riemann Roch theorem. Indeed, the existence and the uniqueness of the canonical representative of a Picard class is a direct consequence of the Riemann Roch theorem, as was detailed in Chapter 2. Nevertheless, to use Riemann Roch we ought to have a formal proof of the theorem, which was a goal far too ambitious, although undeniably interesting for this particular research project. Our purpose was to develop elliptic curve theory in order to formalize elliptic curve algorithms used in cryptographic schemes. As a result, the formalization of Riemann Roch was out of the scope of this project.

During this formalization, there were many times where we needed to manipulate large polynomial formulas, like the parts concerning the group law equations equations, the decomposition of rational functions, or the large technical part concerning the uniqueness of the Picard group representation. Those parts turned out to be technical in Ssreflect, mainly because of the lack of automation. Indeed, Ssreflect is not a mathematics software designed to

compute polynomial formulas or resolve equations in some ring, such as Sage or Maple for example, but a theorem prover to formalize algebra proofs. However, we chose not to use any of these software because they come with no formal guarantee of their correctness. Our development is completely based on Coq and no results have been admitted nor verified tools using untrusted external tools. In that context, Pierre-Yves Strub developed an interface between the `ring` tactic of Coq and the ring structures of SSREFLECT which allowed us to simplify the proof. Nevertheless, we would like to stress out the necessity for tactics that would provide more automation in SSREFLECT because it can be frustrating and discouraging to the developer to confront this kind of difficulties.

Conclusion

In this section, we presented a formal proof of the Picard theorem for elliptic curves, which allowed the formalization of theory about elliptic curves in affine form, rational functions and divisors. Using this theorem, we have proven that the elliptic curve operation defined by polynomial formulas is a group law. Using the above, we will demonstrate that we can represent elliptic curves in different coordinate systems (projective in our case) and that this representation is equivalent. By transport of structure, the curve in projective coordinates also possesses the structure of an abelian group.

3.4 The Projective Plane

Recall that in cryptography, elliptic curves are represented in different coordinate systems in order to accelerate computation. The reason is that addition formulas based on affine coordinates require the computation of field inversions, which is particularly expensive over prime fields. To avoid field inversions, an elliptic curve is often represented using a projective coordinate system. In this section, we present a formalization of elliptic curves in projective coordinates and we demonstrate that this representation is isomorphic to the affine representation described in the previous section.

A projective plane is a geometric concept that expands the concept of an Euclidean plane. In an ordinary Euclidean plane, any two lines intersect in exactly one point, except for parallel lines that intersect in no point. A projective plane can be comprehended as an Euclidean plane together with separate points (called points at infinity) on which parallel lines intersect. Intuitively, a projective plane is a plane where any two lines intersect in exactly one point. In what follows, we start by giving the classical geometrical definitions and move to a more algebraic construction of a projective plane over a field.

The geometrical definition of a projective plane is the following:

Definition 3.12. *A projective plane* $(\mathbb{P}, \mathbb{L})$ *is a non empty set* $\mathbb{P}$ *whose elements are called points, together with a set* $\mathbb{L}$ *whose elements are non-empty subsets of* $\mathbb{P}$ *called lines, satisfying three axioms:*

1. *For any two distinct points* $p_1, p_2 \in \mathbb{P}$*, there exists exactly one line* $l \in \mathbb{L}$ *such that both* $p_1 \in l$ *and* $p_2 \in l$*.*
2. *There exists a set of four points, such that given any set of three of these points, no line exists that contains all three points.*
3. *Any two lines intersect in exactly one point.*

We say that a point p is *on* a line l if $p \in l$. A set of points is said to be *collinear* if there exists a line such that all points are on this line. Using this terminology, we can write the above axioms in a simpler way:

1. Two distinct points determine exactly one line.
2. There exists a set of four points, no three of which are collinear.
3. Any two lines intersect in exactly one point.

Construction from fields

Let $\mathbb{K}$ be a field and $\mathbb{K}^3$ the 3 dimensional vector space over $\mathbb{K}$. Then we can define the projective plane $\mathbb{P}$ over $\mathbb{K}$ as follows.

— The points of $\mathbb{P}$ are the lines of $\mathbb{K}^3$ through the origin $(0, 0, 0)$:

$$\mathbb{P} = \{\alpha v \mid v \in \mathbb{K}^3 \setminus (0, 0, 0), \alpha \in \mathbb{K}\}.$$

— The lines of $\mathbb{P}$ are the planes of $\mathbb{K}^3$ through the origin $(0, 0, 0)$:

$$\mathbb{L} = \{\alpha v + \beta w \mid v, w \in \mathbb{K}^3 \setminus (0, 0, 0), \alpha, \beta \in \mathbb{K}\}.$$

The fact that the three axioms are satisfied follows from some elementary linear algebra:

1. In $\mathbb{K}^3$, two lines with exactly one point in common are contained in exactly one plane. Hence, two distinct projective points are on exactly one projective line and the first axiom is satisfied.
2. The projective points $(1 : 0 : 0)$, $(0 : 1 : 0)$, $(0 : 0 : 1)$ and $(1 : 1 : 1)$ satisfy the second axiom.
3. In $\mathbb{K}^3$, any two planes passing through the origin, intersect in exactly one line passing through the origin too. Hence, any two projective lines intersect in exactly one projective point.

To avoid confusion, from now on, we will denote v (resp. (x, y, z)) for a vector of $\mathbb{K}^3$ and $[v]$ (resp. $(x : y : z)$) for a projective point.

A non-zero vector in $\mathbb{K}^3$ determines a line in $\mathbb{K}^3$ passing through the origin. Therefore we can use the non zero vectors of $\mathbb{K}^3$ to represent the points of $\mathbb{P}$.

Two non-zero vectors v and w represent the same projective point of $\mathbb{P}$ if they are on the same line in $\mathbb{K}^3$, i.e. $[v] = [w]$ iff $v = \alpha w$ for some $\alpha \in \mathbb{K}^*$. Let $v = (x, y, z)$ be a non zero vector of $\mathbb{K}^3$. If $z \neq 0$ then $[v] = [z^{-1}v] = (\frac{x}{z} : \frac{y}{z} : 1)$, and so there exists a unique representative of the class of the form $(x', y', 1)$. Hence, there exists a 1-to-1 map between the set of projective points with $z \neq 0$ and $\mathbb{K}^2$. If $z = 0$, then x and y cannot both be zero because v is a non zero vector. Moreover, if $x \neq 0$ then $[v] = [x^{-1}v] = (\frac{x}{x} : \frac{y}{x} : 0)$ and so there exists a unique representative of the class of the form $(1, y', 0)$. So, there exists a 1-to-1 map between the set of projective points with $z, x \neq 0$ and $\mathbb{K}$. In the same way, when $z = 0$ and $x = 0$, we have $[v] = [y^{-1}v] = (0 : \frac{y}{y} : 0) = (0 : 1 : 0)$. Hence, there exists an isomorphism between the set of projective points $\mathbb{P}$ and the set $\mathbb{K}^2 \cup \mathbb{K} \cup \{(0 : 1 : 0)\}$.

It may have become obvious to the reader by now that starting from the geometrical definition of the projective plane (over some field $\mathbb{K}$) we have arrived to construct an algebraic quotient structure:

Definition 3.13 (Projective plane)**.** *The projective plane $\mathbb{P}^2$ over $\mathbb{K}$ is the quotient $\mathbb{P}^2 = \mathbb{K}^3 \setminus (0, 0, 0)/\sim$ where $(x, y, z) \sim (x', y', z')$ if and only if there exists a $\lambda \in \mathbb{K}^*$ such that $(x', y', z') = (\lambda x, \lambda y, \lambda z)$.*

Elliptic curve on projective plane

A curve of the projective plane is the set of projective points $(x : y : z)$ whose coordinates are a solution of a homogeneous equation $f(x, y, z) = 0$. A curve represented by a homogeneous equation $f(x, y, z) = 0$ is *smooth* or *non-singular* if the partial derivatives of f with respect to x, y, z do not all vanish simultaneously on the curve.

As introduced in Chapter 2, an elliptic curve $\mathcal{E}$ over some field $\mathbb{K}$ is defined as a smooth projective plane curve over $\mathbb{K}$ of the form

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

The above form is called a Generalized Weierstrass form. If the characteristic of the field $\mathbb{K}$ is different from 2 and 3, then the elliptic curve can be put in the short Weierstrass form

$$y^2z = x^3 + axz^2 + bz^3.$$

The condition that the curve is smooth becomes $\Delta = 4a^3 - 27b^3 \neq 0$.

In projective coordinates, addition is defined by the following (geometrical) rules:

1. The negative of a projective point $(x : y : z)$ is the point $(x : -y : z)$.
2. The zero element is the point $(0 : 1 : 0)$, called the point at infinity.
3. The sum of three points that belong to the same projective line is zero.

Like in the affine setting, those rules can be translated into polynomial formulas, which are stable by quotienting. For all $P = (x_P : y_P : z_P) \in \mathcal{E}$ and $Q = (x_Q : y_Q : z_Q) \in \mathcal{E}$, let $S = P + Q = (x_S : y_S : z_S)$ be the sum of the two points.

— If $P = Q$ then

$$\begin{cases} u = 3x_P^2 + az_P^2 & x_S = vr \\ v = 2y_P^2 & y_S = -u(r - v^2x) - y_Pv^3 \\ r = u^2z_P - 2v^2x_P & z_S = z_Pv^3. \end{cases}$$

— If $P \neq Q$ then

$$\begin{cases} u = y_Qz_P - y_Pz_Q & x_S = vr \\ v = x_Qz_P - x_Pz_Q & y_S = -u(r - x_Pz_Qv^2) - y_Pz_Qv^3 \\ r = u^2z_Pz_Q - v^2(x_Pz_Q + x_Qz_P) & z_S = z_Pz_Qv^3. \end{cases}$$

Lemma 3.6. *Let $\mathbb{K}$ be a field of characteristic different from $2, 3$. Let $\mathcal{E}_{\text{proj}}$ be the projective curve $\mathbb{K}$ defined by the equation $Y^2Z = X^3 + aXZ^2 + bZ^3$ and let $\mathcal{E}_{\text{aff}}$ be the affine curve $\mathbb{K}$ defined by the equation $y^2 = x^3 + ax + b$ together with a separate point $\mathcal{O}$. We assume that the condition $\Delta = 4a^3 - 27b^3 \neq 0$ holds.*

Then the map $\phi : \mathcal{E}_{\text{proj}} \to \mathcal{E}_{\text{aff}}$ with $\phi(x : y : z) = (\frac{x}{z}, \frac{y}{z})$ when $z \neq 0$ and $\phi(0 : 1 : 0) = \mathcal{O}$ is an isomorphism of the curves $\mathcal{E}_{\text{proj}}$ and $\mathcal{E}_{\text{aff}}$.

A Formalization of the projective plane. We formalize the construction of projective planes from fields as explained above, and we prove the equivalence of curves in affine and projective coordinates.

Let $\mathbb{K}$ be a field, then the projective plane is defined $\mathbb{K}^3 \setminus (0, 0, 0)$ quotiented by the equivalence relation $(x, y, z) \sim (x', y', z')$ if and only if $(x', y', z') = (\lambda x, \lambda y, \lambda z)$ for some $\lambda \in \mathbb{K}^*$.

To formalize the quotient we have used the standard quotient methodology of Ssreflect which was introduced in Chapter 2. The first step is to define the equivalence relation, by the boolean predicate `lineq`, which is defined in all generality for uplets of all sizes. Next, we restrict `lineq` to triplets with the function `projeq`:

```
Definition lineq p1 p2 : bool :=
  let P := [pred i : 'I_n | tnth p1 i != 0] in
  match [pick i : 'I_n | P i] with
  | None => p1 == p2
  | Some i =>
        [tuple tnth p1 j / tnth p1 i | j < n]
      == [tuple tnth p2 j / tnth p2 i | j < n]
```

```
  end.

Definition projeq (p1 p2 : K * K * K) : bool :=
  let: (x1, y1, z1) := p1 in
  let: (x2, y2, z2) := p2 in
  lineq [tuple x1; y1; z1] [tuple x2; y2; z2].
```

Given as input two n-uplets $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$, the function `lineq` first picks an $a_i \neq 0$ if it exists. Then it checks if for all j in the range $[1, \ldots, n]$ $\frac{a_j}{a_i} = \frac{b_j}{b_i}$. If all $a_i = 0$ then the function checks if all $b_i = 0$ too. In other words, the predicate `lineq` holds for $(a_1, a_2, \cdots, a_n)$ and $(b_1, b_2, \cdots, b_n)$, if it exists a $\lambda \in \mathbb{K}$ such that $(a_1, a_2, \cdots, a_n) = (\lambda b_1, \lambda b_2, \cdots, \lambda b_n)$, as is stated by the following view lemma:

```
Lemma lineqP p1 p2:
  reflect
    (exists2 x, x != 0 & forall i, tnth p1 i = x * (tnth p2 i))
    (lineq p1 p2).

Lemma projeqP (p1 p2 : K * K * K):
  let: (x1, y1, z1) := p1 in
  let: (x2, y2, z2) := p2 in
  reflect
    (exists2 l, l != 0 & [&& x1 == l * x2, y1 == l * y2 & z1 == l * z2])
    (projeq p1 p2).
```

Note that the relation `projeq` is defined on triplets of $\mathbb{K}^3$. Using the lemma `projeqP`, we prove that `projeq` is indeed an equivalence relation:

```
Lemma projeq_refl: reflexive projeq.
Lemma projeq_sym: symmetric projeq.
Lemma projeq_trans: transitive projeq.
```

In the following, we consider that `K` is a fieldType, which is the type of fields in Ssreflect. We define the type `prepoint` to formalize exactly $\mathbb{K}^3 \setminus (0, 0, 0)$ which is the base type of our quotient. An inhabitant of the type `prepoint` is a triplet of $\mathbb{K}^3$ along with a proof that the triplet is different from $(0, 0, 0)$. Next, we restrict the equivalent relation to elements of $\mathbb{K}^3 \setminus (0, 0, 0)$ and we call the restriction `ppequiv`:

```
Inductive prepoint: Type :=
| PrePoint (t : K * K * K) of t != (0, 0, 0).

Definition ppequiv (p1 p2 : prepoint) := projeq p1 p2.
```

We quotient the type prepoint by the relation projeq to obtain the type of projective points {ppoint K}. The function Point allows us to turn any triplet of $\mathbb{K}^3$ into an element of type prepoint. Indeed, if $p \neq (0,0,0)$ then the function returns p together with a proof that it is different from $(0,0,0)$ while if $p = (0,0,0)$ it returns by convention the triplet $(0,0,1)$.

```
Definition Point (K : fieldType) (p : K * K * K) :=
  insubd (PrePoint (@zero_proof K)) p.
```

Subsequently, we define the following notations for elements of the base type prepoint and of the quotient type {ppoint K}:

```
Notation "(| x , y , z |)" := (Point (x, y, z)).
Notation "<[ x : y : z ]>" := (\pi_{ppoint _} (|x, y, z|)).
```

We have left for future work the formalization of the isomorphism between the projective plane over $\mathbb{K}$ and $\mathbb{K}^2 \cup \mathbb{K} \cup \{(0:1:0)\}$ and we move directly to the definition of elliptic curves in projective coordinates.

Formalizing elliptic curves in projective coordinates. Given an affine curve E, we first define the function pponcurve, that decides if a projective point satisfies the equation of the curve in projective coordinates. In our case, we first explicitly define the function (pponcurve_r) on elements of the base type prepoint K and then lift it to the quotient type {ppoint K}:

```
Variable K : ecuFieldType.
Variable E : ecuType K.
Local Notation a := (E#a).
Local Notation b := (E#b).

Definition pponcurve_r (p : prepoint K) :=
  let: (x, y, z) := val p in
    y^+2 * z == x^+3 + a*x*z^+2 + b*z^+3.

Lemma pponcurve_mod_eq (p q : prepoint K):
  ppequiv p q -> pponcurve_r q = pponcurve_r p.

Definition pponcurve := lift_fun1 {ppoint K} pponcurve_r.
```

Following the same steps as in the affine part of the development, we use the pponcurve function to declare a type for projective points on the curve: inhabitants of the type ec_proj are projective points (of type {ppoint K}) that satisfy the curve equation.

```
Inductive ec_proj : Type :=
    | EC_proj : forall p : {ppoint K}, pponcurve p -> ec_proj

Lemma oncurve_ec_proj (p : ec_proj): pponcurve p.
```

Formalizing the isomorphism between affine and projective forms. To prove Lemma 3.6, we explicitly define the map between affine and projective coordinates and we prove that it is a bijection and a morphism for the curve operation.

Let $\mathcal{E}_{\text{aff}}$ be the affine Weierstrass curve defined by the equation $y^2 = x^3+ax+b$ and $\mathcal{E}_{\text{proj}}$ be the projective Weierstrass defined by the equation $\{(X : Y : Z)|Y^2Z = X^3 + aXZ^2 + bZ^3\}$. Then the map $f : \mathcal{E}_{\text{proj}} \to \mathcal{E}_{\text{aff}}$, $f(X : Y : Z) = (\frac{X}{Z}, \frac{Y}{Z})$ if $Z \neq 0$ and $f(X : Y : Z) = \mathcal{O}$ if $Z = 0$, is a bijection between $\mathcal{E}_{\text{aff}}$ and $\mathcal{E}_{\text{proj}}$. In SSREFLECT, instead of defining the map f and f^{-1} on points on the curve, the definitions apply directly on elements of type `point K` and `ppoint K`. This is done deliberately for two reasons: Many of the lemmas expressing properties about the maps are valid even for points that are not on the curve, so we try to stay as general as possible. Moreover, we avoid having to prove in every step that the point is on the curve, therefore making the definitions and the proofs shorter. Note that the function `p2a` (which corresponds to $f : \mathcal{E}_{\text{proj}} \to \mathcal{E}_{\text{aff}}$ as denoted above) is first defined on the base type `prepoint K` and then lifted to the quotient type `{ppoint K}`.

```
Definition a2p (p : point K) : {ppoint K} :=
 if p is (|x, y|) then <[ x : y : 1 ]> else <[ 0 : 1 : 0 ]>.

Definition p2a_r (p : prepoint K) : point K :=
 if p.2 == 0 then EC_Inf else (| p.1.1 / p.2, p.1.2 / p.2 |).

Definition p2a := lift_fun1 {ppoint K} p2a_r.
```

Next, we demonstrate that for all points on the curve, the maps f, f^{-1} cancel each others. As a result, when restricted on the curve, f is a bijection.

```
Lemma a2pK: cancel a2p p2a.
Lemma p2aK: {in pponcurve E, cancel p2a a2p}.
Lemma bij_a2p: {on pponcurve E, bijective a2p}.
```

Like in the affine case, given the curve in projective coordinates we define addition and doubling using polynomial formulas. Note that addition is defined in three steps:

— the function `padd_t` defines addition on triplets of $\mathbb{K}^3$ using the polynomial

formulas,

— the function padd_tr restrains padd_t on $\mathbb{K}^3 \setminus (0,0,0)$ i.e. on elements of type prepoint assuring that outside the curve the function returns $(0,1,0)$ as a convention, and

— after demonstrating that the function padd_tr is stable by the quotient, meaning that $\forall p' \sim p$ and $\forall q' \sim q$, we have $p' + q' \sim p + q$, we can lift it to the quotient type {ppoint K}.

```
Definition pdouble_t (p : K * K * K): K * K * K := nosimpl (
let: (x, y, z) := p in
  if (y == 0) || (z == 0) then (0, 1, 0) else
    let u := 3 * x^+2 + a * z^+2 in
    let v := 2 * y * z in
    let r := u^+2 * z - 2 * v^+2 * x in
    let xs := v * r in
    let ys := -u * (r - v^+2 * x) - y * v^+3 in
    let zs := z * v^+3 in
    (xs , ys , zs)).
```

```
Definition padd_t (p q : K * K * K): K * K * K := nosimpl (
  let: (xp, yp, zp) := p in
  let: (xq, yq, zq) := q in
    if zp == 0 then q else
    if zq == 0 then p else
    if xp / zp == xq / zq then
    if yp / zp == yq / zq then pdouble_t p else (0, 1, 0)
    else
      let u := yq * zp - yp * zq in
      let v := xq * zp - xp * zq in
      let r := u^+2 * zp * zq - v^+2 * (xp * zq + xq * zp) in
      let xs := v * r in
      let ys := -u * (r - xp * zq * v^+2) - yp * zq * v^+3 in
      let zs := zp * zq * v^+3 in
      (xs , ys , zs)).

Definition padd_tr (p q : prepoint K): K * K * K :=
  if pponcurve_r E p && pponcurve_r E q then
    padd_t p q
  else
    (0, 1, 0).

Definition padd_r (p q : prepoint K) := Point (padd_tr p q).
Definition padd := lift_op2 {ppoint K} padd_r.
```

Addition as defined is stable by the quotient and therefore it is a canonical morphism for the class. Some technical details:

— Concerning `double_t`: The function separates three cases: when $y = 0$ we consider that the point is not on the curve ; when $z = 0$ and if the point lies on the curve, it is the infinite point ; when $y \neq 0$ and $z \neq 0$ and if the point lies on the curve, there exists a representative of the form $(x', y', 1)$ and it satisfies the curve equation.

— Concerning `padd_t`: The function separates cases: If one of the points has $z = 0$ then (if it is on the curve) it corresponds to the point at infinity, so the result is directly the other point. Note that points that do not belong to the curve are eliminated by `padd_tr` later on. In the case that $z \neq 0$ and the two points belong to same equivalence class, the function calls `double_t`.

In the same way, we proceed for the definition of the negative of a point:

```
Definition popp_tr (p : prepoint K) : K * K * K :=
  let: (x, y, z) := val p in (x, -y, z).

Definition popp_r (p : prepoint K) := Point (popp_tr p).
Definition popp := lift_op1 {ppoint K} popp_r.
```

Last but not least, we demonstrate the final lemma of this section, stating that the function p2a is an isomorphism between the two curve forms:

```
Lemma oncurve_p2a (p : {ppoint K}): pponcurve E p -> oncurve E (p2a p).

Lemma isomorph (p q : {ppoint K}):
    pponcurve E p -> pponcurve E q
  -> (p \- q) = a2p (p2a p \- p2a q).
```

Based on the isomorphism that allows using all the theory of curves in affine coordinates, we were able to prove several properties of the opposite and the addition on projective points, such as:

```
Lemma poppK: forall q :{ppoint K}, popp (popp q) = q.

Lemma pponcurve_popp (p : ec_proj E): pponcurve E (popp p).

Lemma pponcurve_padd_ppoint: forall (p1 p2 : {ppoint K}),
pponcurve E p1 -> pponcurve E p2 -> pponcurve E (padd E p1 p2).

Lemma pponcurve_padd (p1 p2 : ec_proj E): pponcurve E (padd E p1 p2).
```

Since we have demonstrated that the sum of two points always lies on the curve, then we can lift the definitions of addition and the opposite to elements of type `ec_proj E`. Applying the isomorphism lemma and by transport of structure, the addition in projective coordinates satisfies all group properties and therefore we can equip `ec_proj E` with a $\mathbb{Z}$-module structure.

```
Definition ecp_zero := EC_proj (pponcurve0 E).
Definition ecp_opp (p : ec_proj E) := EC_proj (pponcurve_popp p).
Definition ecp_add (p1 p2 : ec_proj E) := EC_proj (pponcurve_padd p1 p2).

Lemma ecpC : commutative ecp_add.
Lemma ecp0e : left_id ecp_zero ecp_add.
Lemma ecpNe : left_inverse ecp_zero ecp_opp ecp_add.
Lemma ecpNe : left_inverse ecp_zero ecp_opp ecp_add.
Lemma ecpA : associative ecp_add.

Definition ecp_zmodMixin := ZmodMixin ecpA ecpC ecp0e ecpNe.
Canonical ecp_zmodType := Eval hnf in ZmodType (ec_proj E) ecp_zmodMixin.
```

3.5 Related work

Hurd et al. [HGF06] formalize elliptic curves in higher order logic using the HOL-4 proof assistant. Their goal is to create a "gold-standard" set of elliptic curve operations mechanized in HOL-4, which can be used afterwards to verify ec-algorithms for scalar multiplication. They define datatypes to represent elliptic curves on arbitrary fields (in both projective and affine representation), rational points and the elliptic curve group operation, although they do not provide a proof that the operation indeed satisfies the group properties. In the end, they state the theorem that expresses the functional correctness of the ElGamal encryption scheme for elliptic curves.

Smith et al. [SD08] use the Verifun proof assistant to prove that two representations of an elliptic curve in different coordinate systems are isomorphic. Their theory applies to elliptic curves on prime fields. They define data structures for affine and projective points and the functions that compute the elliptic curve operations in affine and Jacobian coordinates. In their formalization there is no datatype for elliptic curves, an elliptic curve is a set of points that satisfy a set of conditions. They define the transformation functions between the two systems of coordinate and prove that for elliptic curve points the transformation functions commute with the operations and that both representations of elliptic curves in affine or Jacobian coordinates are isomorphic.

Théry [Thé07] present a formal proof that an elliptic curve is a group using the Coq proof assistant. The proof that the operation is associative relies heavily

on case analysis and requires handling of elementary but subtle geometric transformations and therefore uses computer-algebra systems to deal with non-trivial computation. In our development, we give a different proof of the associativity of the elliptic curve group law: we define an algebraic structure (the Picard group of divisors) and proceed to prove that the elliptic curve is isomorphic to this structure. Our formalization is more structural than [Thé07] in the sense that it involves less computation and the definition of new algebraic structures.

As in [HGF06] and [SD08] we wish to develop libraries that will enable the formal analysis of elliptic curve algorithms and our proofs follow textbook mathematics. As in [Thé07], we give a formal proof of the group law for elliptic curves. Nevertheless, the content of our development is quite different from the related work. To the extent of our knowledge this is the first formalization of divisors and rational functions of a curve, which are objects of study of algebraic geometry. Such libraries may allow the formalization of non-trivial algorithms that involve divisors (such as the Miller algorithm for pairings [Mil86]), isogenies (such as [BJ03], [DIK06]) or endomorphisms on elliptic curves (such as the GLV algorithm for scalar multiplication [GLV01]).

4

A formalization of the GLV algorithm

In this chapter, we present a formal proof of correctness of the GLV algorithm [GLV01] for scalar multiplication on an elliptic curve group. This proof uses theory from the elliptic curve library that was presented in Chapter 3. The development includes over 5k lines of code and is available at https://github.com/strub/glv.

The GLV algorithm was initially presented in 2000 in the article [GLV01] by Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. It presents an important efficiency advantage and this is the reason why it was adopted, studied and implemented in many versions, up to now. A first complexity analysis of GLV is presented in [GLV01] and a further analysis in [SCQ02]. Since its publication on 2001, significant research has been made to optimize performance of GLV [FLS15], to analyze its security properties and its applicability to different settings.

The idea behind GLV is the following: Let us consider an elliptic curve $\mathcal{E}$ over some prime field $\mathbb{F}_p$. Suppose that given a random point $P \in \mathcal{E}$ we can somehow compute easily a (non-trivial) multiple of P, say $[\lambda]P$. Then when asked to compute another multiple $[k]P$, we can break it down to $[k]P = [k_1]P + [k_2]([\lambda]P)$, with k_1 and k_2 having half the size of the initial k. Then we can use a fast double-multiplication algorithm (multi-exponentiation), which presents an important efficiency advantage compared to computing $[k]P$ directly.

However, computing the multiple $[\lambda]P$ of a given point P implies that the curve has an efficiently computable endomorphism ϕ which acts as scalar

multiplication $\phi = [\lambda]$; i.e. $\forall Q \in \langle P \rangle, \phi(Q) = [\lambda]Q$, for a certain $\lambda \in \mathbb{N}$. Under certain conditions (which usually hold in a cryptographic setting), all endomorphisms act as a multiplication on the cyclic subgroup $\langle P \rangle$. Note that by efficiently computable endomorphism, we mean that it can be computed by executing only a few field operations. For example, if the prime of the base field satisfies the condition $p = 1 \pmod 4$ and i is a square root of -1 in $\mathbb{F}_p$, then any curve of the form $y^2 = x^3 + ax$ has an explicit and very efficient endomorphism:

$$\phi(x, y) = (-x, iy).$$

To compute $\phi(x, y)$, one has to perform only one field operation. In this case, $\lambda = \sqrt{p-1}$. The integer λ that characterizes the endomorphism ϕ is one of the roots of the characteristic polynomial of ϕ.

The main inconvenience of GLV is that it requires finding curves with computable endomorphisms, which turns out to be highly nontrivial. This is the reason why on 2009 Galbraith, Lin and Scott proposed a modified version of GLV, named the GLS algorithm in [GLS09]. GLS solves the problem of finding curves with computable endomorphisms in the following way: starting with any elliptic curve over a prime field, first it takes the extension of the curve over the quadratic extension field. Then it uses an efficiently computable homomorphism which arises from the Frobenius map on the quadratic twist of the curve. In addition, a generalization of the GLS algorithm is presented with the Q-curve construction in [Smi16].

The GLV algorithm is especially interesting to formalize, first because of its use in cryptographic implementations and secondly, because of the mathematics involved: besides two generic algorithms (multiexponentiation and decomposition of the scalar), the endomorphisms part demanded formal theory for non trivial properties of elliptic curves. Our formalization follows the description of GLV as in [GLV01]

We have not been concerned by any optimized versions nor the GLS algorithm on the quadratic extension field. Note that our formal proof stays in an abstract mathematical level, i.e. the correctness of GLV is proven as a mathematical high-level property but we do not provide any low-level implementation that would be running efficiently on a machine. This part is left for future work.

GLV is composed by three independent sub–algorithms: multi–exponentiation, decomposition of the scalar and computing an endomorphism on an elliptic curve. Therefore we divide our development in three corresponding parts:

1. a formal proof of the algorithm of double-exponentiation on an abstract group,
2. a formal proof of a decomposition algorithm, based on the Extended Euclidean Algorithm, and

3. a formal proof that any endomorphism on an elliptic curve acts as multiplication on a cyclic subgroup of points.

The first two algorithms are formalized exactly as presented in [GLV01] and are completely independent from the elliptic curve development presented in the Chapter 3. The proof of the third part is an extension of the elliptic curve development.

Remark. In the third part, we do not formally prove how to extract the λ given the endomorphism. Yet, we give the formal proof that given an endomorphism ϕ, there exists always a $\lambda \in \mathbb{Z}$ such that $\phi(P) = [\lambda]P$.

4.1 The multi-exponentiation algorithm

Double-exponentiation is based on a well-known method for exponentiation in groups: the sliding window technique [Gor98]. The algorithm depends on a parameter w, a small positive integer (called the window or the block size) in the sense that the binary representation of the exponent is split into binary blocks of size w. First, a precomputation stage takes place, where a table of group elements is computed. Then, an evaluation stage takes place where the final result is computed, using the table of auxiliary values. This is the main idea of sliding window methods for single exponentiation, but sliding windows techniques for double exponentiation work in a similar way: In the precomputation stage, input group elements are combined with each other and then, at the evaluation stage, all exponents are computed simultaneously.

The upcoming description of the algorithm is taken directly from the article [GLV01]. In the following, $(u_{t-1}, \ldots, u_1, u_0)_2$ denotes the binary representation of the integer u and w is the window size.

Algorithm 11: Simultaneous sliding window exponentiation in an additive group

Input: $w \in \mathbb{N}^*$, $u = (u_{t-1}, \ldots u_1, u_0)_2$, $v = (v_{t-1}, \ldots, v_1, v_0)_2$, P, Q.
Compute $iP + jQ$ for all $i, j \in [0, 2^{w-1}]$.
Write $u = (u^{d-1}, \ldots, u^1, u^0)_2$ and $v = (v^{d-1}, \ldots, v^1, v^0)_2$
    where each u^i and v^i is a bitstring of length w and $d = \frac{t}{w}$.
Set $R \leftarrow 0$.
For i from $d-1$ downto 0 do
    $R \leftarrow 2^w R$
    $R \leftarrow R + (u^i P + v^i Q)$
Return R
Output: $R = uP + vQ$

Proof. The proof of correctness is done by induction on the number of blocks d. Let P and Q be elements of an additive group.

1. Suppose that $d = 1$. Then in the first round (i.e. the only round) of the algorithm we have:
   — $R \leftarrow 0$
   — $R \leftarrow 2^w R = 0$
   — $R \leftarrow R + (uP + vQ) = uP + vQ$
2. For all u and v integers, we suppose that the algorithm is correct for a split of u, v in $k \in \mathbb{N}^*$ blocks. Let $u = (u_{t-1}, \ldots u_1, u_0)_2$, $v = (v_{t-1}, \ldots v_1, v_0)_2$ be the binary representation of the two integers and $u = (u^k, \ldots, u^1, u^0)_2$, $v = (v^k, \ldots, v^1, v^0)_2$ their splitting into $k+1$ blocks. Let $u' = (u^{k-1}, \ldots, u^1, u^0)_2$ and $v' = (v^{k-1}, \ldots, v^1, v^0)_2$ be the two integers that derive from the k first blocks of u and respectively v. By the induction hypothesis, the result of the algorithm with input the two integers u', v' decomposed into k blocks, as above, will be $u'P + v'Q$.

   Now suppose the algorithm is called on input u and v both split into $k + 1$ blocks. At the k-th iteration of the algorithm, the output will be $R \leftarrow u'P + v'Q$. We have to show that if we run once more the loop of the algorithm (the last step) the result will be $uP + vQ$. Indeed, $R = 2^w(u'P+v'Q)+(u^kP+v^kQ) = (2^wu'+u^k)P+(2^wv'+v^k)Q = uP+vQ$. Hence, the proof by induction is completed. □

According to the algorithm description, the first step is to put the two integers u and v into binary form. The function `nat_to_bin` gives the binary form of an integer, starting from the least significant bit. Reversely, the function `bin_to_nat` given a sequence of bits, computes the corresponding integer.

```
Fixpoint nat_to_bin_aux (n a : nat) : seq bool :=
  if a is a.+1 then
    if n is 0 then [::] else
      (odd n) :: (nat_to_bin_aux n./2 a)
  else [::].

Definition nat_to_bin (n : nat) := nat_to_bin_aux n n.

Fixpoint bin_to_nat (l : seq bool) : nat :=
  if l is x :: xs then
    (x + 2 * (bin_to_nat xs))
  else 0.
```

The function `bin_to_nat` when applied to a integer gives a boolean sequence that finishes always with 1, since the MSB comes in the end of the sequence. So we have easily proven the following:

```
Lemma cancelN n : bin_to_nat (nat_to_bin n) = n.
```

However, the inverse statement `cancelB` is not always true:

```
Lemma cancelB n : nat_to_bin (bin_to_nat u) = u.
```

More precisely, in the case that `u` is a boolean sequence that finishes with `false`, that statement is not correct. Therefore we define the function `norm_bin1`, that is designed in order to drop all zeros in the end of a boolean sequence:

```
Definition epurate (b : bool):=
  if b then [:: true] else [::].

Fixpoint norm_bin1 (u : seq bool) : seq bool :=
  if u is x :: xs then
    let v := norm_bin1 xs in
    if x is y :: ys then x :: v else epurate v
  else [::].
```

Hence, we are able to prove the lemma `cancelB` and several other properties.

Next, we define the function `block`, that given a sequence of bits, extracts the `i`-th block of size `w`.

```
Definition block (u : seq bool) (w i : nat) : seq bool :=
  mkseq (fun k => nth false u (w*i + k)) w.
```

Following closely the description of the algorithm, we define the function `n_blocks` which computes the number of blocks of size w that will result by the splitting into blocks.

```
Definition n_blocks (u : seq bool) (w : nat) : nat :=
  if size u is 0 then 0 else (((size u).-1) / w).+1.
```

Finally, we have all the elements needed to define the algorithm of double exponentiation, on an abstract additive group `G`:

```
Definition algoG (w : nat) (u v : seq bool) (P Q : G) :=
  let d := maxn (n_blocks u w) (n_blocks v w) in
    foldl (fun (R : G) (i : nat) =>
      let R0 := R *+ 2 ^ w in
        R0 + (P *+ bin_to_nat (block u w i)) +
          (Q *+ bin_to_nat (block v w i)))
      0 (rev (iota 0 d)).
```

First, we extract the number of blocks d which is the number of times the loop is executed. The loop iteration is performed via `foldl` that iterates the operations of the loop exactly as many as d times. Here, we use the standard notation `*+` of SSREFLECT for scalar multiplication. Next we prove that the algorithm is correct, by induction on the number of blocks, as described previously:

```
Lemma algo_correct (P Q : G) (n m w : nat) : w != 0 ->
  algoG w (nat_to_bin n) (nat_to_bin m) P Q = P *+ n + Q *+ m.
```

Note that our formal proof of correction does not regard the precomputation part. More precisely, in this abstract version of the algorithm, we do not separate the precomputation part as in the mathematical description.

4.2 The decomposition algorithm

In this section, we describe the formalization of a decomposition algorithm proposed in [GLV01]. The algorithm takes as input two integers k, λ and $n \in \mathbb{N}$ and outputs two integers k_1, k_2 satisfying $k = (k_1 + k_2\lambda) \mod n$. Here k_1, k_2 will be the arguments of the double exponentiation function and n is the order of the point, which is the input of the GLV scalar multiplication algorithm. In real cryptographic settings, n is a prime number. One may observe that we can easily find such k_1, k_2: indeed $k_1 = k$ and $k_2 = 0$ satisfy the above equation. Since k_1, k_2 are the arguments of double-exponentiation, they are supposed to be the shortest possible (i.e. their binary form should be the shortest possible). The algorithm proposed returns k_1, k_2 half the size of the initial k. An informal analysis is presented in [GLV01]. Our formalization concerns only a proof of correction and we have not formalized any results concerning the size of the output.

Let $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_n$ defined by $f(i, j) = (i + j\lambda) \mod n$. The function f is a morphism for $\mathbb{Z} \times \mathbb{Z}$, i.e. it satisfies $f(x_1 + x_2, y_1 + y_2) = f(x_1, y_1) + f(x_2 + y_2)$ and $f(-x, -y) = -f(x, y)$.

The algorithm aims to output a short vector $u = (k_1, k_2)$ of $\mathbb{Z} \times \mathbb{Z}$ such that $f(u) = k$. Using vector space vocabulary, we recall that a vector $v = (x, y)$ is short if it has small Euclidean norm: $|v| = \sqrt{x^2 + y^2}$. The algorithm can be split in two separate steps:

— first, we find two linearly independent short vectors v_1, v_2 of $\mathbb{Z} \times \mathbb{Z}$ satisfying $f(v_1) = f(v_2) = 0$.

— Next, we find a vector v in the integer lattice generated by v_1, v_2 that is close to the vector $(k, 0)$.

Since f is a morphism, it follows that $u = (k, 0) - v$ is a short vector satisfying $f(u) = f(k, 0) - f(v) = k - 0 = k$. As noticed in [GLV01], both subproblems can

be approached using lattice reduction algorithms, yet the algorithm proposed in the article is more efficient and simple to implement.

Finding v_1, v_2. In order to find linearly independent $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$ such that $f(v_1) = f(v_2) = 0$ we use the Extended Euclidean Algorithm (EEA): Applying the EEA to find the greatest common divisor of n and λ we get a sequence of (s_i, t_i, r_i) satisfying the relation

$$s_i n + t_i \lambda = r_i \text{ for } i = 0, 1, 2, \ldots$$

with the initial $(s_0, t_0, r_0) = (1, 0, n)$ and $(s_1, t_1, r_1) = (0, 1, \lambda)$ and with $r_i \geq 0, \forall i$.

For all i we have $f(r_i, -t_i) = r_i - \lambda t_i = s_i n = 0 \pmod{n}$. Hence, we can choose $v_1 = (r_{i_1}, -t_{i_1})$ and $v_2 = (r_{i_2}, -t_{i_2})$ for some i_1, i_2 in the range of i; more precisely we will choose them in order for v_1, v_2 to have the smallest euclidean norm. Let m be the greatest index such that $r_m \geq \sqrt{n}$. We choose $v_1 = (r_{m+1}, -t_{m+1})$ and v_2 the shortest between $(r_{m+2}, -t_{m+2})$ and $(r_m, -t_m)$. The fact that the two vectors are linearly independent is assured by the following well known properties of EEA:

— $r_i > r_{i+1} \geq 0$, for all $i \geq 0$

— $|t_i| > |t_{i+1}|$, for all $i \geq 0$

Reasoning by contradiction, if v_1 and v_2 are not linearly independent, (without loss of generality we assume that $v_2 = (r_m, -t_m)$), then $\frac{r_{m+1}}{r_m} = \frac{|t_{m+1}|}{|t_m|}$ which is absurd since $\frac{r_{m+1}}{r_m} < 1$ while $\frac{|t_{m+1}|}{|t_m|} > 1$.

Computing v_1, v_2 only depends on n and λ and not on the scalar k, so they can be precomputed in advance.

Finding v. We wish to find a vector $v = av_1 + bv_2$ where $a, b \in \mathbb{Z}$ and v close to the vector $(k, 0)$. We consider $(k, 0)$, v_1 and v_2 as vectors of $\mathbb{Q} \times \mathbb{Q}$. Then we can find $\alpha, \beta \in \mathbb{Q}$ such that $(k, 0) = \alpha v_1 + \beta v_2$. Suppose $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$. To find α, β we have to solve the linear system

$$\begin{cases} x_1\alpha + x_2\beta = k \\ y_1\alpha + y_2\beta = 0. \end{cases}$$

This system has a solution since the vectors v_1, v_2 are linearly independent as it was shown above. We round up α, β to the nearest integers, say respectively a, b, and we define $v = av_1 + bv_2$. The vector $u = (k, 0) - v$ is the short vector satisfying $f(u) = k \pmod{n}$.

Formalization. The formalization follows closely the mathematical description given above and presents very few differences. First, we define the recursive function `eea`, which formalizes the extended euclidean algorithm. Given two integers `a` and `b`, `eea` returns the list of (r_i, u_i, v_i) such that $r_i = u_i a + v_i b$ (as

produced by the euclidean algorithm). Note that if the output list is $[(r_1, u_1, v_1) :: (r_2, u_2, v_2) :: \cdots :: (r_N, u_N, v_N)]$ then $r_1 < r_2 < \cdots < r_N$.

```
Fixpoint eea_rec r' (u' v' : int) (acc : seq (nat * int * int)) n :=
  if n is n.+1 then
    if r' == 0 then Some acc else
      let: (r, u, v) := head (0, 0, 0) acc in
      let: (q, m) := (r / r', r %% r') in
      eea_rec m (u - q * u') (v - q * v') ((r', u', v') :: acc) n
  else None.

Definition eea (a b : nat) : seq (nat * int * int) :=
  if a == 0 then [:: (b, 0, 1)] else
    odflt [::] (eea_rec b 0 1 [:: (a, 1, 0)] (maxn a b).+1).

Lemma eea_mod a b v :
  let: (r, x, y) := v in
    (r, x, y) \in (eea a b) -> r - y * b = x * a.
```

Looking closer at the auxiliary function `eea_rec`, we remark the extra parameter `n` which decreases with every recursive call. This parameter is necessary to define the `eea_rec` function because of the problem of function termination in Coq. Indeed, if we try writing the function without the `n` parameter, Coq displays the error message:

```
Error: Cannot guess decreasing argument of fix.
```

This is because, in Coq, recursive calls should be made on strict sub-terms in order to ensure that functions terminate. In our case, the procedure of the extended euclidean algorithm cannot be defined in some Coq function as–is because Coq cannot see directly the decreasing argument. As a result, it is necessary to introduce the extra (fuel) parameter `n` to explicitly make sure that the function will terminate in all cases, after the execution of at most $n + 1$ iterations. In the main function, `eea` n is naturally initialized as the maximum of the input integers a and b.

Following the mathematical description, we choose $v_1 = (r_{m+1}, -t_{m+1})$ and v_2 the shortest between $(r_{m+2}, -t_{m+2})$ and $(r_m, -t_m)$, where m is the greatest index such that $r_m \geq \sqrt{n}$. Consequently, we filter the `eea` sequence to get all triplets that contain $r \geq \sqrt{n}$, the result sequence being `eea_sqrr`:

```
Definition filter_sqrr (n : nat) (rs : seq (nat * int * int)) :=
  [seq t <- rs | let: (r, x, y) := t in (n <= r ^2)].

Definition eea_sqrr (n l : nat) := filter_sqrr n (eea n l).
```

Note that the sequence `eea_sqrr` is already sorted concerning r, so the triplet of index m, as denoted in the mathematical description, is just the first element of the output sequence. Given the first element of `eea_sqrr`, the function `index_hd` returns its index m in the sequence `eea`, and the function `base` returns the two vectors $v_1 = (r_{m+1}, -t_{m+1})$ and v_2 the shortest between $(r_{m+2}, -t_{m+2})$ and $(r_m, -t_m)$.

```
Definition index_hd (n l : nat) :=
  index (head (0, 0, 0) (eea_sqrr n l)) (eea n l).

Definition base (n l : nat) :=
  let m := index_hd n l in
  if (m <= 1) then
    let: (r0, x0, y0) := nth (0, 0, 0) (eea n l) 0 in
    let: (r1, x1, y1) := nth (0, 0, 0) (eea n l) 1 in
    ((r0, -y0),(r1, -y1))
  else
    let: (r1, x1, y1) := nth (0, 0, 0) (eea n l) m.-1 in
    let: (r0, x0, y0) := nth (0, 0, 0) (eea n l) m in
    let: (r2, x2, y2) := nth (0, 0, 0) (eea n l) m.-2 in
    if (r0 ^+ 2 + y0 ^+ 2 <= r2 ^+ 2 + y2 ^+ 2)
    then ((r1, -y1),(r0, -y0))
    else ((r1, -y1),(r2, -y2)).
```

We verify that the result vectors v_1, v_2 satisfy $f(v_1) = f(v_2) = 0$ under the conditions $n, \lambda \neq 0$. Those two restrictions are always satisfied in reality because n is usually a prime number, while λ is the eigenvalue of the endomorphism and therefore it cannot be zero.

```
Lemma base_modn_fst n l :
  let (v1, v2) := base n l in
  v1.1 + v1.2 * l = 0 %[mod n]

Lemma base_modn_snd n l : n != 0 -> l != 0 ->
  let (v1, v2) := base n l in
  v2.1 + v2.2 * l = 0 %[mod n].
```

Arriving at the second part, we have to find a vector close to $(k, 0)$ in the integer lattice generated by the base $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$. First we have to find the $\alpha, \beta \in \mathbb{Q}$ from the linear system

$$\begin{cases} x_1\alpha + x_2\beta = k \\ y_1\alpha + y_2\beta = 0, \end{cases}$$

and then their approximation a, b in $\mathbb{Z}$. Therefore, we define the elementary

approximation function approxZ: given two integers n and m and to compute the approximation of $\frac{n}{m}$, first we isolate the signs and then operate on the approximation of $\frac{|n|}{|m|}$. Let the euclidean division of $|n|$ by $|m|$ be $|n| = |m|q + r$. If $\frac{r}{|m|} > \frac{1}{2}$ we round up to the next integer, else to the previous one.

```
Definition approx (n m : nat) :=
  let r := n %% m in
  let q := n %/ m in

  if (2*r <= m) then q else q.+1.

Definition approxZ (n m : int) :=
  (sgz n) * (sgz m) * (approx (absz n) (absz m)).
```

Next, we directly compute the solution (a, b) of the linear system applying Cramer's rule and round up using approxZ.

```
Definition cramer_coefs n l k :=
  let (v1, v2) := base n l in
  let D := v1.1 * v2.2 - v1.2 * v2.1 in
  (approxZ (k * v2.2) D, approxZ (- k * v1.2) D).
```

We have all the elements in order to compute the final *decomposition* vector $(k_1, k_2) = (k, 0) - (av_1 + bv_2)$:

```
Definition decomp (n l k : nat) : int * int :=
  let (a, b) := cramer_coefs n l k in
  let (v1, v2) := base n l in
  let k1 := k - (a * v1.1 + b * v2.1) in
  let k2 := - (a * v1.2 + b * v2.2) in
  (k1, k2).
```

Unfolding the explicit definitions of this section and the properties of the mod function of Ssreflect, it is straightforward to prove that the output of the decomposition is correct:

```
Lemma correct_decomp n l k :
  let (k1, k2) := decomp n l k in
    n != 0 -> l != 0 -> k = (k1 + k2 * l) %[mod n].
```

4.3 Computing the endomorphisms

In this section, we give a formal proof of the fact that an endomorphism on an elliptic curve acts as scalar multiplication on a subgroup of curve points. This is true under certain conditions which always hold in a cryptographic setting:

— the scalar multiplication is performed on a cyclic group generated by a point of prime order, and

— the square of the order of the generator point does not divide the cardinal of the curve.

We have formalized the following theorem:

Theorem 4.1. *Let $\mathcal{E}$ be an elliptic curve defined on some finite field $\mathbb{K}$ and let $\phi : \mathcal{E} \to \mathcal{E}$ be an endomorphism of $\mathcal{E}$. Let G point of $\mathcal{E}$ of prime order n, such that n^2 does not divide the order of $\mathcal{E}$. Let $\langle G \rangle = \{[k]G \mid k = 0, 1, \ldots, n-1\}$ be the cyclic subgroup generated by G. Then, there exists an integer $\lambda \in [1, \ldots, n-1]$ such that $\forall A \in \langle G \rangle$, $\phi(A) = [\lambda]A$.*

The first step to the proof, was to formalize the subgroup of n-torsion points with coordinates on $\mathbb{K}$:

$$\mathcal{E}[n] = \{Q \in \mathcal{E} \mid [n]Q = \mathcal{O}\}.$$

Remark. In mathematical literature, the set of n-torsion points *with coordinates on* $\mathbb{K}$ is usually denoted as $\mathcal{E}[n]_{(\mathbb{K})}$ to underline the fact that points have coordinates on $\mathbb{K}$, while $\mathcal{E}[n]$ is used for n-torsion points with coordinates on the algebraic closure of $\mathbb{K}$. In this section, for the shake of simplicity, we are using the notation $\mathcal{E}[n]$ instead of $\mathcal{E}[n]_{(\mathbb{K})}$ for n-torsion points with coordinates on the field $\mathbb{K}$.

In what follows, we assume that `K` is a finite field of characteristic different from 2 and 3, `E` is an elliptic curve in short Weierstrass form defined over `K` and `n` is a natural number:

```
Variable K : finECUFieldType.
Variable E : ecuType K.
Variable n : nat.
```

The subgroup of n-torsion points is defined as follows: An inhabitant of the type `torsion` is an elliptic curve point (represented by the type `ec`), along with a proof that n-times that point is equal to the point at infinity. The function `tgpoint` returns the first projection of an element of type `torsion` i.e. the point itself, without the proof. `torsion` is a subtype of `ec` and is directly equipped with the structure of an `eqType`, a `ChoiceType` and a `finType`:

```
Record torsion := Torsion { tgpoint :> ec E; _ : tgpoint *+ n == 0 }.
```

Addition is defined as the restriction of the elliptic curve addition on elements of type `torsion`. The zero element of the torsion subgroup is the point at infinity, called `tg0`. To formally define addition (`tgadd`) and the opposite (`tgopp`) on the n-torsion subgroup, first we prove that the operations are internal. Next we prove that the operations and the point at infinity as defined, satisfy the group properties. Consequently, we can equip `torsion` with the structure of a $\mathbb{Z}$-module represented by the type `zmodType`.

Subsequently, we assume all the cryptographic conditions of the theorem; in our setting, `G` corresponds to the point G, the generator of the cyclic group. The order of point `G` is n and so it is of type `ntorsion`, which is just a notation for elements of type `torsion E n`. The variable `phi` defines a random curve homomorphism and the last condition `crypto_order` denotes that n^2 does not divide the order of the curve. We make use of the standard SSREFLECT notations for cardinals: `#|S|` denotes the cardinal of a finite set `S`.

```
Variable K : finECUFieldType.
Variable E : ecuType K.
Variable n : nat.
Hypothesis prime_n : prime n.
Notation ntorsion := (torsion E n).
Variable G : ntorsion.
Hypothesis GNz : G != 0.
Variable phi : {additive (ec E) -> (ec E)}.
Hypothesis crypto_order : ~~ ( (n ^ 2) %| #|[set: ec E]| ).
```

The next step is to prove that in this particular setting $\langle G \rangle = \mathcal{E}[n]$; in other words that that n-torsion points form a cyclic subgroup. We break it down to two parts: i) $\langle G \rangle \subseteq \mathcal{E}[n]$, and ii) $\mathcal{E}[n] \subseteq \langle G \rangle$. The first lemma is stated in Coq as follows:

```
Goal forall r, (G *+ r) *+ n = 0.
```

Using the theory in the `cyclic` module of the Mathematical Components we can prove the above lemma as follows:

Lemma 4.2. $\langle G \rangle \subseteq \mathcal{E}[n]$

Proof. Let $A \in \langle G \rangle$. Then there exists $a \in \mathbb{Z}_n$ such that $A = [a]G$. Hence,

$$[n]A = [n]([a]G) = [a]([n]G) = [a]\mathcal{O} = \mathcal{O}.$$

So, $A \in \langle G \rangle \implies A \in \mathcal{E}[n]$. □

We continue with the second inclusion lemma, $\forall P \in \mathcal{E}[n], P \in \langle G \rangle$ with stated in Coq like as follows:

```
Goal forall Q, tgpoint Q \in <[tgpoint G]>%g.
```

The proof is done by contradiction:

Lemma 4.3. $\mathcal{E}[n] \subseteq \langle G \rangle$.

Proof. By contradiction, suppose $\exists Q \in \mathcal{E} \setminus \langle G \rangle$. Then Q is not a multiple of G, which means that G and Q are linearly independent points of order n, where n is a prime number. Hence, the subgroup $\langle G, Q \rangle$ generated by G and Q has order n^2. But, $\langle G, Q \rangle$ is a subgroup of $\mathcal{E}$ and by the Lagrange theorem, its order divides the order of $\mathcal{E}$. This is a contradiction, since we have assumed that n^2 does not divide the order of $\mathcal{E}$. □

The proof of the above lemma necessitated the proof of several intermediate lemmas concerning torsion points and a more general finite group result:

```
Goal forall (gT : finGroupType) (x y : gT),
  #[x] = n -> #[y] = n -> commute <[x]> <[y]> ->
    x \notin <[y]> -> prime n -> ((n^2) %| #|[set: gT]|).

Lemma order_torsion (P : ntorsion) : P != 0 -> #[tgpoint P] = n.

Lemma commute_cycles (A B : ntorsion) :
  commute <[tgpoint A]> <[tgpoint B]>.
```

Here we use the standard Ssreflect notations:

`<[x]>` = the cycle (cyclic group) generated by x
`#[x]` = the order of x, i.e., the cardinal of <[x]>.

Finally we are able to demonstrate that ϕ acts as multiplication with λ when restricted on the group generated by G:

Lemma 4.4. $\forall P \in \langle G \rangle, \phi(P) = [\lambda]P$.

Proof. ϕ can be restricted to $\mathcal{E}[n] = \langle G \rangle$ since $\forall Q \in \mathcal{E}[n]$ we have also that $\phi(Q) \in \mathcal{E}[n]$. Indeed, since ϕ is a morphism we have $[n]\phi(Q) = \phi([n]Q) = \phi(\mathcal{O}) = \mathcal{O}$. Consequently if $Q \in \langle G \rangle$ then $\phi(Q) \in \langle G \rangle$ too.

$$\phi(G) \in \langle G \rangle \text{ implies that } \exists \lambda \in \mathbb{Z}_n \text{ such that } \phi(G) = [\lambda]G.$$

Therefore, for an arbitrary point $A = [a]G$ of $\langle G \rangle$, we have that

$$\phi(A) = \phi([a]G) = [a]\phi(G) = [a]([\lambda]G) = [\lambda]([a]G) = [\lambda]A.$$

□

The proof of the `final` lemma in Coq follows the proof detailed above. Remark that `phi` is a morphism defined on elliptic curve points (represented by the type `ec`). To prove the `final` lemma, we have to consider the restriction on n-torsion points. Indeed, at the first place we prove that $\forall P \in \mathcal{E}[n]$, $\phi(P)$ is also in $\mathcal{E}[n]$ and then we are able to define the restriction `tgphi` on n-torsion points.

```
Lemma phi_torsion_restriction n (P : ec E) :
  P *+ n = 0 -> phi (P *+ n) = 0.

Lemma tgphi_interne (P : ntorsion):
  (phi (tgpoint P)) *+ n == 0.

Definition tgphi (P : ntorsion) : ntorsion :=
  Torsion (tgphi_interne P).

Lemma ok_def_tgphi (P : ntorsion) :
  tgpoint (tgphi P) == phi (tgpoint P).

Lemma final: { m : nat | forall Q : ntorsion, phi Q = Q *+ m }.
```

The function `phi2l` allows us to extract the λ of the endomorphism:

```
Definition phi2l := tag final.

Lemma phi2lP (Q : ntorsion): phi Q = Q *+ phi2l.
```

Remark Remark that in SSREFLECT `<[G]>` is of type `{set ec_finGroupType}` while n-torsion points are represented by the type `ntorsion`. As a result, although we could prove that $\langle G \rangle$ and $\mathcal{E}[n]$ are isomorphic, we cannot write $\langle G \rangle = \mathcal{E}[n]$ as in the mathematical proof, because it implies using equality between different types.

4.4 The GLV algorithm

At this point, we have formalized proofs of correction for all the three independent parts of the algorithm. Some small additional details are needed in order to be able to compose them all together. For example, the double exponentiation algorithm `algoG` takes two *positive* integers as input, while the decomposition algorithm returns two integers that may be negative. To correct this we use the function `multexpoGZ` and we demonstrate that it is correct.

```
Variable G : zmodType.
```

```
Definition multiexpoGZ (w : nat) (n m : int) (P Q : G) : G :=
  let: (n, P) :=
    match n with Posz n => (n, P) | Negz n => (n.+1, -P) end in
  let: (m, Q) :=
    match m with Posz m => (m, Q) | Negz m => (m.+1, -Q) end in

  algoG G w (nat_to_bin n) (nat_to_bin m) P Q.

Lemma multiexpoGZ_correct w n m P Q : w != 0%N ->
  multiexpoGZ w n m P Q = P *~ n + Q *~ m.
```

Finally and under the cryptographic conditions explained above, we prove the lemma that the GLV algorithm is correct:

```
Variable K : finECUFieldType.
Variable E : ecuType K.
Variable n : nat.
Hypothesis prime_n : prime n.
Variable phi : {additive (ec E) -> (ec E)}.
Notation ntorsion := (torsion E n).
Variable G : ntorsion.
Hypothesis GNz : G != 0.
Hypothesis crypto_order: ~~ ( (n ^ 2) %| #|[set: ec E]| ).
Notation l := (@phi2l K E n prime_n G GNz phi foo).

Lemma final (w k : nat) : l != 0%N -> w != 0%N ->
  multiexpoGZ w (decomp n l k).1 (decomp n l k).2
    (tgpoint G) (phi G) = G *+ k.
```

4.5 Related work

An impressive verification work on elliptic curves is presented in [CHL+14], which targets low-level hand-optimized qhasm [Ber] code for Curve25519 [Ber06]. A realted approach is an application of our work (which will be detailed in the next chapter) and is described in the article [ZBB16]. It presents a verified elliptic curve library that covers three popular curves —Curve25519, Curve448, and NIST-P256— and a verified constant-time bignum library. The formal tool used is the dependently-typed programming language F* [SHK+16]. The elliptic curve interface is proved functionally correct against a mathematical specification derived from the Coq development presented in this thesis. To perform scalar multiplication, the Montgomery ladder is implemented while the curves are in short Weierstrass or Montgomery form and are represented by Jacobian or

projective coordinate systems. The two approaches make different trade-offs, in that the first aims towards a library extensible with minimal additional verification effort, whereas the second focus on verifying a highly-performant implementation of one curve.

In a separate line of work of verification of asymmetric cryptography, the `Ironclad` [HHL+14] crypto library also provides security guarantees for SHA, HMAC and RSA at the assembly level, but they have not verified elliptic curves so far.

Other verification efforts have targeted symmetric cryptography. [App15b] used the Coq proof assistant to prove that a legacy `SHA-256` implementation written in C was correct with regard to its specification and [BPYA15] showed that the OpenSSL implementation of `HMAC` using `SHA-256` implements its specifications correctly and provides the expected cryptographic guarantees. [DHL+05] presents a collection of functional correctness proofs for symmetric block encryption algorithms such as AES, MARS, Twofish, RC6, Serpent and IDEA. [Cry] provides a tool to verify that a cryptographic implementation matches a high-level specification, and this tool has been used to verify block ciphers and hash functions.

Most recently, [ABBD16] shows how to prove cryptographic security, functional correctness, and side-channel protection for a complex cryptographic construction all the way from high-level cryptographic definitions down to assembly code, using a combination of several different verification tools. Finally, a number of works address the problem of verifying the security of complex cryptographic constructions, protocols, and their implementations [BGZB09, BGHB11, BFK+14].

4.6 Comments and Future work

The formalization of GLV required few choices to be made from the part of the developers. Once the proofs were sufficiently detailed, the definition of the new datatypes and the formalization of all the statements emerged naturally, more or less like in the paper proof in [GLV01]. However, the part of the endomorphisms, which was based on the formalization of our elliptic curve library, came out without difficulty which assured us of the stability of our library.

An extension to the GLV development, which presents interest as future work is the computation of the value λ that characterizes the endomorphism. More precisely, λ is one of the roots of the characteristic polynomial of ϕ. In what follows, we will give a sketch of the proof of the above fact, that could be formalized in the future. Recall the theorem formalized in the previous section:

Lemma 4.5. *Let $\mathcal{E}$ be an elliptic curve defined on some finite field $\mathbb{K}$ and let $\phi : \mathcal{E} \to \mathcal{E}$ be an endomorphism of $\mathcal{E}$. Let G point of $\mathcal{E}$ of prime order n, such*

that n^2 does not divide the order of $\mathcal{E}$. Let $\langle G \rangle = \{[k]G \mid k = 0, 1, \ldots, n-1\}$ be the cyclic subgroup generated by G. Then, there exists $\lambda \in \mathbb{Z}_n$ such that $\forall A \in \langle G \rangle$, $\phi(A) = [\lambda]A$.

Let $\mathcal{E}[n]$ be the subgroup of n-torsion points, over the algebraic closure of $\mathbb{K}$ and $\mathcal{E}[n]_{(\mathbb{K})}$ the subgroup of n-torsion points over $\mathbb{K}$. We state the following proposition without giving a full proof:

$$\mathcal{E}[n] \text{ is isomorphic to } \mathbb{Z}_n \times \mathbb{Z}_n$$

This is a well known textbook result that can be found in [Gui10]. Since, $\mathcal{E}[n]$ is isomorphic to $\mathbb{Z}_n \times \mathbb{Z}_n$, there exists a two point basis P, Q such that $\forall S \in \mathcal{E}[n]$, $S = [x]P + [y]Q$.

Definition of the characteristic polynomial of ϕ

As shown in the previous section ϕ can be restricted to $\mathcal{E}[n]$. Let the image of the base be $\phi(P) = [a]P + [b]Q$ and $\phi(Q) = [c]P + [d]Q$ for some $a, b, c, d \in \mathbb{Z}_n$. Then, the image of a random point $S = [x]P + [y]Q$ can be computed by the linear transformation,

$$\phi(S) = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

The characteristic polynomial $\chi(t)$ of ϕ (over $\mathcal{E}[n]$) is defined as the characteristic polynomial of the matrix $\begin{bmatrix} a & c \\ b & d \end{bmatrix}$, in other words $\chi(t) = t^2 - (a+d)t + (ad - bc)$. If $\chi(t)$ has two distinct roots λ_1, λ_2 (which will always be the case in cryptographic applications), then $\mathcal{E}[n]$ can be split into $\mathcal{E}[n] = G_1 \oplus G_2$ where G_1 and G_2 are cyclic subgroups of $\mathcal{E}[n]$. From a linear algebra point of view, G_1 and G_2 are the eigenspaces corresponding to the eigenvalues λ_1 and λ_2.

Since χ is the characteristic polynomial of ϕ, by definition $\chi(\phi) = 0$. As a result, for all points S in $\mathcal{E}[n]_{(\mathbb{K})}$, $\chi(\phi)(S) = \chi([\lambda]S) = \mathcal{O}$ and so λ is one of the roots of χ.

5 Applications

The work presented in this thesis aimed to provide formal theory in order to verify elliptic curve software used in cryptography. In this chapter we present two ways of applying our work in order to certify elliptic curve algorithms for cryptography. An independent application of our work is the use of our library to formalize related mathematical structures.

5.1 Verifying GLV with CoqEAL (future work)

A first approach to the verification of elliptic curves algorithms is to use the CoqEAL methodology [DMS12] to obtain a certified implementation of the GLV algorithm and then extract from it an ML program. This is a natural way of proceeding since we have used Ssreflect as the base of our development. and it is left for future work.

CoqEAL is a methodology for proving correct efficient algebraic algorithms via refinements. First the algorithm is proven correct in an abstract mathematical setting, using all the high-level theory and properties from the Mathematical Components library. Then it is refined to a low-level implementation, using simpler data-structures, which can be actually run on a machine. The link between the two implementations is done through morphism lemmas that connect the two data-structures (the abstract one and the efficient one). The usability of the CoqEAL methodology is demonstrated on several different algorithms: matrix

rank computation, Winograd's fast matrix product, Karatsuba's polynomial multiplication, and the gcd of multivariate polynomials. The main advantage of this approach is that on one hand, the abstract high-level algorithm make use of the dependent types of SSREFLECT and all the corresponding properties to formalize a proof of correctness and on the other hand the efficient low-level implementation use simple types closer to real-life implementations. The separation of proofs from the computational content, guarantees the safety of the approach. More precisely, the following 3 steps methodology is used to construct efficient algorithms from high-level definitions:

1. Write an abstract version of the algorithm and prove it correct using the SSREFLECT structures and theory.
2. Implement an efficient version of the algorithm using the SSREFLECT structures and prove that it corresponds to the abstract version.
3. Translate the abstract data-structures and the efficient algorithm to the low-level data types.

The CoqEAL methodology comes with libraries that have implemented efficiently computable counterparts to several algebraic structures: $\mathbb{Z}$–modules, rings and fields.

Using CoqEAL to refine GLV in an efficient implementation will require writing an efficient version of the three algorithms presented in Chapter 4 (multi-exponentiation, decomposition and computation of the endomorphisms) and translate the elliptic curve group into an efficient data-structure. It will also be interesting to extend the abstract proof of correctness of the algorithm by the proof of an algorithm computing the value λ as explained in the end of Chapter 4.

5.2 A Verified Library of Elliptic Curves in F*

In real cryptographic settings, elliptic curve software is implemented in low-level programming languages such as C or assembly. This includes using efficient libraries for number representations and curve specific code which is often further modified to be side–channel resistant. To link our high-level development in SSREFLECT to a more realistic setting, I participated in the work of Jean Karim Zinzindohoué and Karthikeyan Bhargavan described in the article [ZBB16]. It presents a verified elliptic curve library that covers three popular curves - Curve25519, Curve448, and NIST-P256-and that can be easily extended with new curves. It also presents a verified constant-time generic bignum library, which is of independent interest. The elliptic curve implementation is written in the dependently-typed programming language F* [SHK+16] and proved functionally correct against a readable mathematical specification derived from the Coq

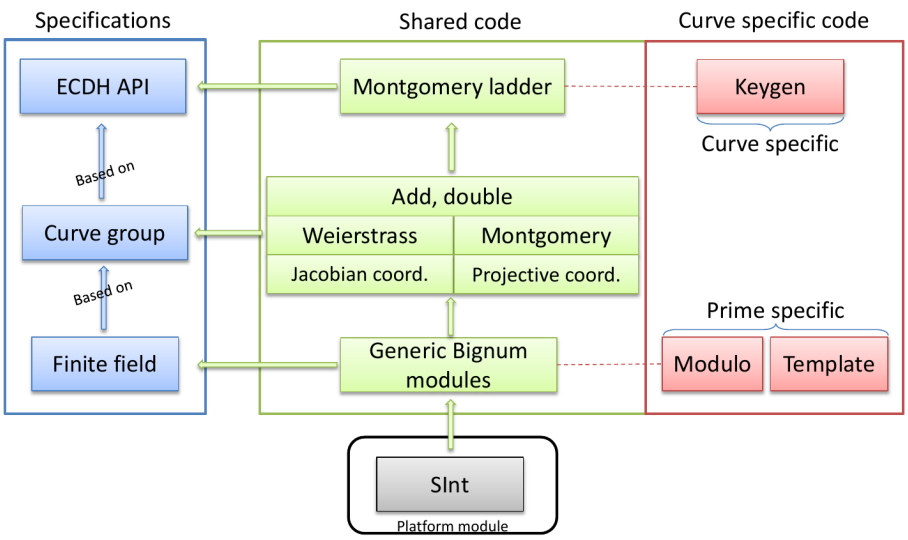


Figure 5.1 – Architecture of our verified elliptic curve library

development presented in this thesis. The F* library can be readily incorporated into larger verified cryptographic applications written in F*, such as miTLS. The goal of this development is to build a verified library that consists of multiple elliptic curves that maximally share code, so that the verification effort of adding a new curve can be minimized. Functional correctness of the code is certified, as well as enforcement of a source-level coding discipline that mitigates side-channels. Furthermore, special effort is made to be able to embed the library within a verified protocol implementation.

The elliptic curve library in F* provides a typed API that encodes the mathematical specification of elliptic curves. Each curve in the library is proved to satisfy this specification by typing. The Montgomery ladder is implemented to perform scalar multiplication while the curves in short Weierstrass or Montgomery form use Jacobian or projective coordinate systems. The curve code implements the same algorithmic optimizations as state-of-the-art elliptic curve implementations. In particular, they implemented a bignum library that allows each curve to choose its own unpacked bignum representation (called a template) and obtain verified field arithmetic for free, except for a few curve-specific functions that need to be implemented and verified separately. Mitigations against side-channels are systematically enforced throughout the library by treating secrets as opaque bytestrings whose values cannot be inspected. The architecture of the library is illustrated in the picture 5.1.

The elliptic curve API (The module *Curve Group* in the picture 5.1) is what directly links the implementation in F* with our development and goes all the way up to the ECDH Algorithm 1. The Coq definitions presented in Chapter 2

of this thesis are carefully transcribed as an F* interface, the Coq theorems are reflected as F* assumptions. While there is no formal link between F* and Coq, an informal discipline is imposed whereby all unverified elliptic curve assumptions in F* must be justified by a corresponding theorem in Coq.

In the F* API the curve definition assumes a finite field $\mathbb{K}$ represented by elements of type `felem`. Points on the plane are of type `affine_point`: they can either be `Inf` , the point at infinity, or a pair of `felem` coordinates. A point that verifies the curve equation satisfies `on_curve`, and is represented by the refined type `celem` , denoting curve elements. They define two operations over curve points: a negation function `neg` and an internal group operation `add` . The `ec_group` lemma says that `Inf` , `neg` , and `add` together form an abelian group structure. Hence, they can define scalar multiplication as repeated addition over the curve. The `AbelianGroup` predicate gives a textbook definition of an abelian group equipped with a neutral element `zero`, an opposite function `opp` and addition operator `add`.

We display below the F* API:

```
type AbelianGroup (#a:Type) (zero:a) (opp:a → Tot a)
                    (add:a → a → Tot a) =
 (∀ x y z. add (add x y) z = add x (add y z)) // Associative
 ∧ (∀ x y. add x y = add y x) // Commutative
 ∧ (∀ x. add x zero = x) // Neutral element
 ∧ (∀ x. add x (opp x) = zero) // Inverse

(∗ Field elements, parameters of the equation ∗)
val a: felem
val b: felem
val is_weierstrass_curve: unit −>
Lemma (4 +∗ a^3 ^+ 27 +∗ b^2 ≠ zero ∧
  characteristic ≠ 2 ∧ characteristic ≠ 3)

type affine_point =
  | Inf | Finite: x:felem → y:felem → affine_point

let on_curve p = is_Inf p || (is_Finite p &&
 (let x, y = get_x p, get_y p in y^2 = (x^3 ^+ a ^∗ x ^+ b)))
type CurvePoint (p:affine_point) = b2t(on_curve p)

let neg' p = if is_Inf p then Inf
 else Finite (Finite.x p) (−(Finite.y p))

let add' p1 p2 =
  if not(on_curve p1) then Inf
  else if not(on_curve p2) then Inf
```

```
  else if is_Inf p1 then p2
  else if is_Inf p2 then p1
  else (
    let x1 = get_x p1 in let x2 = get_x p2 in
    let y1 = get_y p1 in let y2 = get_y p2 in
    if x1 = x2 then (
      if y1 = y2 && y1 ≠ zero then (
   let lam = ((3 +∗ (x1^2) ^+ a) ^/ (2 +∗ y1)) in
   let x = ((lam^2) ^− (2 +∗ x1)) in
   let y = ((lam ^∗ (x1 ^− x)) ^− y1) in
   Finite x y
      ) else (...)))

(∗ Type of points on the curve ∗)
type celem = p:affine_point{CurvePoint p}
val neg: celem → Tot celem
val neg_lemma: p:celem −> Lemma (neg p = neg' p)
val add: p:celem → q:celem → Tot celem
val add_lemma: p:celem → q:celem → Lemma (add p q = add' p q)

val ec_group_lemma:
  unit → Lemma (AbelianGroup #celem Inf neg add)

(∗ EC multiplication of a point by a scalar ∗)
val smul : ℕ → celem → Tot celem
let smul n p = match n with
  | 0 → Inf | _ → add p (smul (n−1))
```

In our corresponding Coq interface, the finite field `K` is of type `fieldType` while points on the plane are of type `point` : they can either be `EC_Inf` , the point at infinity, or a pair of `K` coordinates. In exactly the same way as in the F* API, a point that verifies the curve equation satisfies the predicate `oncurve` and is represented by the type `ec`, denoting curve elements. The same two operations are defined over projective plane points and then lifted to curve points: a negation function `neg` and an internal group operation `add` . As described in Chapter 2, the function `addec` is proven to be the operation of an abelian group and therefore one can define scalar multiplication using the standard SSReflect notation.

We display here the Coq interface:

```
Record ecuFieldMixins (K:fieldType): Type :=
  Mixin { _: 2 != 0; _: 3 != 0 }.
Record ecuType :=
  {A:K; B:K; _:4 ∗ A^3 + 27 ∗ B^2 != 0}.
```

```
Inductive point := EC_Inf | EC_In of K & K.
Notation "(x, y)" := (EC_In x y).

Definition oncurve (p : point) :=
  if p is (x, y) then y^2 == x^3 + A * x + B else true.
Inductive ec : Type := EC p of oncurve p.

Definition neg (p : point) :=
  if p is (x, −y) then (x, −y) else EC_Inf.

Definition add (p1 p2 : point) :=
  let p1 := if oncurve p1 then p1 else EC_Inf in
  let p2 := if oncurve p2 then p2 else EC_Inf in
    match p1, p2 with
    | EC_Inf, _ => p2
    | _, EC_Inf => p1
    | (x1, y1), (x2, y2) =>
        if x1 == x2 then ... else
        let s := (y2 − y1) / (x2 − x1) in
        let xs := s^2 − x1 − x2 in
        (xs, − s * (xs − x1) − y1) end.

Lemma addO (p q : point): oncurve (add p q).
Definition addec (p1 p2 : ec) : ec := EC p1 p2 (addO p1 p2).

scalar_multiplication (n:nat) (p:point K) = p *+ n.
```

The full library of verified elliptic curves in F*, implementing the three popular curves Curve25519, Curve448, and NIST-P256, currently consists of about 5800 lines of code. It continues to evolve as new curves are added and existing code is refactored for efficiency and to simplify and speed up the proofs. The current version of the library is available at https://github.com/mitls/hacl-star/tree/master/ecc_star.

5.3 An SSREFLECT library for monoidal algebras

To formalize our elliptic curve library we have developed a number of data-structures, some of which remain of independent interest and could be used to formalize further mathematical theories. In this section we give an example of such an application, which makes use of the `freeg` structure to formalize multivariate polynomials.

In the article [BBRS16], a formal proof that e and π are transcendental numbers is presented. A transcendental number is a real or complex number that is not a root of a non-zero integer polynomial. In other words, a transcendental

number is any number that is not algebraic. All real transcendental numbers are irrational, because any rational number is a root of a degree one polynomial. Since the set of algebraic numbers is countable, and the set of real or complex numbers is uncountable, most real or complex numbers are transcendental. The most famous transcendental numbers are e and π.

The formalization presented in [BBRS16] relies on a proof by Niven [Niv39]. The methodology used to show that π and e are transcendental is identical [Niv39]: Supposing that the number is algebraic, results in an equality $E_p = E'_p$ where E_p and E'_p are two expressions depending on $p \in \mathbb{Z}$. Then for p sufficiently large, they prove that $|E_p| < (p-1)!$, while $|E'_p| > (p-1)!$ and by contradiction they conclude that the number is not algebraic. The proof that $|E_p| < (p-1)!$ uses properties of real and complex analysis, while the proof that $|E'_p| > (p-1)!$ is based on arithmetic and algebraic results. As a result, there are two parts of the proof, one based on calculus using the Coquelicot library and one based on algebra using the Mathematical Components library [Gon07, GAA+13, BLM15].

The proof of transcendence of π relies heavily on properties of multivariate and symmetric polynomials, therefore a new library was developed on top of the Mathematical Components library to formalize multivariate polynomials.

Multivariate polynomials are polynomial expressions with several variables. In Ssreflect, univariate polynomials are represented as the list of its coefficients. To be more precise, the polynomial $\Sigma_{k\in\mathbb{N}} a_k x^k$ is represented by the list $[a_0, a_1, ..., a_n]$ where a_n is the last non-zero coefficient. The above representation could lead to an intuitive representation of multivariate polynomials with a finite set of variables, using an enumeration of the countable set $\mathbb{N}^n$. In this case, a multivariate polynomial can be represented as the list of its coefficients, taken in the ordering given by the enumeration of $\mathbb{N}^n$. However, this representation is ineffective as explained in the article mainly because it depends on the chosen enumeration which may cause complications and also cannot be lifted to the case of an infinite number of variables. Another option would be representing the multivariate polynomial ring $R[x_1, x_2, ..., x_n]$ as $R[x_1][x_2]...[x_n]$, i.e. iterating the univariate construction for all indeterminates. For example, in this case a polynomial $p(x, y)$ is seen as a univariate polynomial of indeterminate y, whose coefficients are polynomials of indeterminate x. This representation is still not efficient in Coq, because again it cannot be lifted to the infinite case, and moreover it arises problems when trying to use the Canonical Structures mechanism of Ssreflect.

The solution described in [BBRS16] is to represent a commutative multivariate polynomial as a formal sum of the form $\Sigma a_i(X_1^{k_i^1} X_2^{k_i^2} ... X_n^{k_i^n})$. This part of the development extends the `freeg` structure for free abelian groups, described in Chapter 2 of this thesis, and initially designed to formalize divisors on elliptic curves. More precisely, to construct the type of commutative multivariate

polynomials in n variables, they extended `freeg` as follows: the set of coefficients is a monoid, and the set of generators is the type of finite functions from $I = \{1, 2, ..., n\}$ to $\mathbb{N}$. Another extension of `freeg` results in the non-commutative multinomials: a non-commutative multivariate polynomial in n variables, is an instance of `freeg` where the set of coefficients is a monoid, and the set of generators is the set of sequences `seq I`. The library formalized contains proofs for several properties of multivariate polynomials, including evaluation, derivation and the proof that this representation is isomorphic to the iterated construction.

6
Conclusion

Elliptic curves two different point of views From a mathematics point of view, an elliptic curve is a geometric object, i.e. a functor of points given by the curve equation. From a cryptographic point of view, an elliptic curve is a set of points satisfying the curve equation, meaning that the main action in cryptographic protocols, is the exchange of points between different parties. Nevertheless, in mathematics we make a distinction between the elliptic curve $E : y^2 = x^3 + Ax + B$ and its set of points, $E(\mathbb{K})$; and it is this distinction that allows us define more complicated notions on top of elliptic curves such as the function field or differentials. Regarding our formalization, we have adopted a cryptographic point of view, in the sense that the curve is the set of points satisfying the curve equation. However, the curve can also be seen as a functor, since the definition is parametrized by the base field and the curve parameters. The geometrical perspective is more obvious in the addition law, where we use lines reflected as polynomial formulas for coordinates.

To sum up We have presented a formalization of an elliptic curve library using the Ssreflect extension of Coq, which we hope will enable formal analysis of elliptic-curve schemes and algorithms. Our central result is a formal proof of Picard's theorem for elliptic curves whose immediate consequence is the associativity of the elliptic curve group operation. Based on this library, we have presented a formal proof of correctness of the GLV algorithm [GLV01] for scalar multiplication. This development includes the formal proofs of two independent

algorithms (multiexponentiation on a generic group and decomposition of the scalar) and also theory about computing endomorphisms on algebraic curves. Furthermore, we have presented an application of our work in Chapter 5 certifying real-life implementations of elliptic curve algorithms combining our development in Coq and F*.

Concerning Binary Curves In our formalization, we are concerned with elliptic curves, on fields with characteristic different from 2 and 3. We chose to restrict to this case because (i) the large prime field case is the most important for contemporary cryptography and (ii) the only other important case which is actually used, is the binary case, where the base field is $\mathbb{F}_{2^n}$ and would require a whole different field implementation. Nevertheless, the theory of characteristic 2 and 3 is essentially the same, with the exception that one would have to use different formulas for the addition law. So, in principle, all the results in this thesis could be adapted to the binary case. Binary curves are interesting mostly to hardware development, but since our interest lies in program verification (from a cryptographic point of view), and most of the curves in use are prime curves, our results remain important.

Using Ssreflect Working with Ssreflect as a user to formalize mathematics was sometimes pleasant and sometimes painful. The main things that I have learned is to be as rigorous as possible in mathematics, and a lot about programming (taking into account that I had not a strong background). The representation of the same mathematical structure into different data-structures was the subject that I found more interesting while working on this thesis. Refactoring the proofs and definitions in order to make them available for further use took most of the time and while important was not particularly pleasant.

Automation As already mention, during the work in this thesis, there were several times when we needed to manipulate large polynomial formulas, like the parts concerning the group law equations equations, the decomposition of rational functions, or the large technical part concerning the uniqueness of the Picard group representation. Those parts turned out to be quite technical in Ssreflect, mainly because of the lack of automation. For our formalization, we decided not to use any mathematics software designed to compute polynomial formulas or resolve equations in some ring, such as Sage or Maple because they come with no formal guarantee of their correctness. Our development is completely based on Coq and there are no holes, in the sense that there are no results admitted nor any non-verified tools used to aid computation. In Coq there exist a form of automation (the `ring` tactic) which was not compatible with the Ssreflect methodology. As a result, there were many computational proofs in Ssreflect which had to be developed step-by-step by the users. In

that context, Pierre-Yves Strub developed an interface between the **`ring`** tactic of Coq and the ring structures of SSREFLECT which allowed us to simplify the proof. Nevertheless, we would like to stress out the necessity for tactics that would provide more automation in SSREFLECT because it can be frustrating and discouraging to the developer to confront this kind of difficulties.

Bibliography

[ABBD16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 163–184. Springer, 2016.

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 5–17. ACM, 2015.

[ADGR07] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic (TOCL)*, 9(1):2, 2007.

[Adl79] Leonard M. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography (abstract). In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 55–60. IEEE Computer Society, 1979.

[AH14] Jeremy Avigad and John Harrison. Formally verified mathematics. *Commun. ACM*, 57(4):66–75, 2014.

[AM16] Enrico Tassi Assia Mahboubi. Mathematical components. Tutorial, 2016.

[App15a] Andrew W Appel. Verification of a cryptographic primitive: Sha-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7, 2015.

[App15b] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7, 2015.

[APS12] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, pages 1–25, 2012.

[BBC+14] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1267–1279. ACM, 2014.

[BBGO09] Santiago Zanella Béguelin, Gilles Barthe, Benjamin Grégoire, and Federico Olmedo. Formally certifying the security of digital signature schemes. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 237–250. IEEE Computer Society, 2009.

[BBPV11a] B.B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. Cryptology ePrint Archive, Report 2011/633, 2011. http://eprint.iacr.org/.

[BBPV11b] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. *IACR Cryptology ePrint Archive*, 2011:633, 2011.

[BBPV12] Billy B Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. In *Topics in Cryptology–CT-RSA 2012*, pages 171–186. Springer, 2012.

[BBRS16] Sophie Bernard, Yves Bertot, Laurence Rideau, and Pierre-Yves Strub. Formal proofs of transcendence for e and pi as an application of multivariate and symmetric polynomials. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 76–87. ACM, 2016.

[BC04] Yves Bertot and Pierre Castéran. Coq'Art: examples and exercises, 2004. http://www.labri.fr/Perso/~casteran/CoqArt.

[BCHL13] Joppe W. Bos, Craig Costello, Hüseyin Hisil, and Kristin E. Lauter. High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 331–348. Springer, 2013.

[BDK13] Michael Backes, Goran Doychev, and Boris Köpf. Preventing side-channel leaks in web traffic: A formal approach. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[Ber] D. J. Bernstein. qhasm: tools to help write high-speed software. https://cr.yp.to/qhasm.html.

[Ber06] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[Ber08] Yves Bertot. Introduction to dependent types in coq. University Lecture, 2008.

[BF01] Dan Boneh and Matthew K. Franklin. Identity based encryption from the weil pairing. *IACR Cryptology ePrint Archive*, 2001:90, 2001.

[BFK+14] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the tls handshake secure (as it is). In *Advances in Cryptology (CRYPTO)*, pages 235–255. 2014.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology (CRYPTO)*, pages 71–90, 2011.

[BGJB07] Gilles Barthe, Benjamin Grégoire, Romain Janvier, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *IACR Cryptology ePrint Archive*, 2007:314, 2007.

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

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. pages 90–101, 2009.

[Bih09] Sidi Ould Biha. Finite groups representation theory with coq. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, Held as Part of CICM 2009, Grand Bend, Canada, July 6-12, 2009. Proceedings*, volume 5625 of *Lecture Notes in Computer Science*, pages 438–452. Springer, 2009.

[Bih10] Sidi Ould Biha. *Composants mathématiques pour la théorie des groupes. (Mathematical components for groups theory)*. PhD thesis, University of Nice Sophia Antipolis, France, 2010.

[BJ03] Eric Brier and Marc Joye. Fast point multiplication on elliptic curves through isogenies. In *AAECC*, pages 43–50, 2003.

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[BPYA15] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl hmac. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., August 2015. USENIX Association.

[Bra39] Alfred Brauer. On addition chains. *Bull. Amer. Math. Soc.*, 45(10):736–739, 10 1939.

[But99] Levente Buttyan. Formal methods in the design of cryptographic protocols (state of the art). Technical report, 1999.

[Cas91] John William Scott Cassels. *LMSST: 24 Lectures on Elliptic Curves*, volume 24. Cambridge University Press, 1991.

[CC86] David V. Chudnovsky and Gregory V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7:385–434, 1986.

[CH85] Thierry Coquand and Gérard P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *European Conference on Computer Algebra (1)*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer, 1985.

[CHL+14] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying curve25519 software. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 299–309. ACM, 2014.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 1998.

[CNE+14] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 319–335. USENIX Association, 2014.

[Coh12] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory.* PhD thesis, École polytechnique, Nov 2012.

[Coh13] Cyril Cohen. Pragmatic quotient types in coq. In *ITP*, pages 213–228, 2013.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[Cry] Cryptol Development Team. Cryptol, the language of cryptography. http://www.cryptol.net/.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

[DHL+05] Jianjun Duan, Joe Hurd, Guodong Li, Scott Owens, Konrad Slind, and Junxing Zhang. Functional correctness proofs of encryption algorithms. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference (LPAR 2005)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 519–533. Springer, December 2005.

[DIK05] Christophe Doche, Thomas Icart, and David R. Kohel. Efficient scalar multiplication by isogeny decompositions. *IACR Cryptology ePrint Archive*, 2005:420, 2005.

[DIK06] Christophe Doche, Thomas Icart, and David R. Kohel. Efficient scalar multiplication by isogeny decompositions. In *Public Key Cryptography*, pages 191–206, 2006.

[DIM05] Vassil S. Dimitrov, Laurent Imbert, and Pradeep Kumar Mishra. Efficient and secure elliptic curve point multiplication using double-base chains. In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*, pages 59–78. Springer, 2005.

[DJM98] Vassil S. Dimitrov, Graham A. Jullien, and William C. Miller. An algorithm for modular exponentiation. *Inf. Process. Lett.*, 66(3):155–159, 1998.

[DMS12] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in coq. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.

[FLS15] Armando Faz-Hernández, Patrick Longa, and Ana H. Sánchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on GLV-GLS curves (extended version). *J. Cryptographic Engineering*, 5(1):31–52, 2015.

[Fre93] Friedrich Ludwig Gottlob Frege. *Grundgesetze der Arithmetik: Begriffsschriftlich abgeleitet. Erster Band.* Pohle, 1893.

[Fri98] Stefan Friedl. An elementary proof of the group law for elliptic curves. *The Group Law on Elliptic Curves*, 1998.

[Ful89] William Fulton. *Algebraic curves - an introduction to algebraic geometry (reprint vrom 1969).* Advanced book classics. Addison-Wesley, 1989.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[Gal12] Steven D. Galbraith. *Mathematics of Public Key Cryptography.* Cambridge University Press, 2012.

[Gam84] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.

[Gar11] François Garillot. *Generic Proof Tools and Finite Group Theory. (Outils génériques de preuve et théorie des groupes finis).* PhD thesis, École Polytechnique, Palaiseau, France, 2011.

[Gil04] Dowek Gilles. Théories des types. 2004.

[GLS09] Steven D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 518–535. Springer, 2009.

[GLV01] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms.

In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.

[GMR+07] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2007.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[Gon11] Georges Gonthier. Point-free, set-free concrete linear algebra. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2011.

[Gor98] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.

[GPP+16] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. 2016.

[Gui10] Philippe Guillot. *Courbes Elliptiques, une présentation élémentaire pour la cryptographie*. Lavoisier, 2010.

[HGF06] Joe Hurd, Mike Gordon, and Anthony Fox. Formalized elliptic curve cryptography. High Confidence Software and Systems, 2006.

[HHL+14] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[HIS14] JORDAN HISEL. Addition law on elliptic curves. 2014.

[Ica09] Thomas Icart. How to hash into elliptic curves. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2009.

[Kas15] Emilia Kasper. We ♡ SSL. Real World Cryptography, 2015.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[LD99] Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over gf(2^{m}) without precomputation. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 1999.

[LH16] Adam Langley and Mike Hamburg. Elliptic curves for security. IETF RFC 7748, 2016.

[Lon11] Patrick Longa. *High-Speed Elliptic Curve and Pairing-Based Cryptography*. PhD thesis, University of Waterloo, 2011.

[LS14] Patrick Longa and Francesco Sica. Four-dimensional gallant-lambert-vanstone scalar multiplication. *J. Cryptology*, 27(2):248–283, 2014.

[Mea03] Catherine A. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, 2003.

[Mil85] Victor S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, pages 417–426, 1985.

[Mil86] Victor S. Miller. Short programs for functions on curves. In *IBM THOMAS J. WATSON RESEARCH CENTER*, 1986.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[MOV93] Alfred Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Information Theory*, 39(5):1639–1646, 1993.

[MR03] Silvio Micali and Leonid Reyzin. Physically observable cryptography. *IACR Cryptology ePrint Archive*, 2003:120, 2003.

[Nat99] National Institute of Standards and Technology (NIST). RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE. Technical report, 1999.

[Nat13] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Technical report, 2013.

[Niv39] Ivan Niven. The transcendence of pi. *American Mathematical Monthly*, 46(8):469–471, October 1939.

[Ope15] OpenSSL. Bignum squaring may produce incorrect results (cve-2014-3570), January 2015. https://www.openssl.org/news/secadv/20150108.txt.

[OS01] Katsuyuki Okeya and Kouichi Sakurai. Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a montgomery-form elliptic curve. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2001.

[Pol78] John M. Pollard. Monte Carlo methods for index computation mod p. *Mathematics of Computation*, 32:918–924, 1978.

[PP89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013.

[Rei60] George W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.

[Sch87] René Schoof. Nonsingular plane cubic curves over finite fields. *J. Comb. Theory, Ser. A*, 46(2):183–211, 1987.

[SCQ02] Francesco Sica, Mathieu Ciet, and Jean-Jacques Quisquater. Analysis of the gallant-lambert-vanstone method based on efficient endomorphisms: Elliptic and hyperelliptic curves. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers*, volume 2595 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2002.

[SD08] Eric Whitman Smith and David L. Dill. Automatic formal verification of block cipher implementations. In *FMCAD*, pages 1–7, 2008.

[Sha84] Adi Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO*, pages 47–53, 1984.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[Sil09] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, Dordrecht, second edition, 2009.

[Smi16] Benjamin Smith. The $\mathbb{P}$-curve construction for endomorphism-accelerated elliptic curves. *J. Cryptology*, 29(4):806–832, 2016.

[SMY06] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A formal practice-oriented model for the analysis of side-channel attacks. *IACR e-print archive*, 134(2006):2, 2006.

[Ste03] Jacques Stern. Why provable security matters? In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 449–461. Springer, 2003.

[Str64] Ernst G. Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70:806–808, 1964. URL: `http://cr.yp.to/bib/entries.html#1964/straus`.

[Sut15] Andrew V. Sutherland. Elliptic curves as abelian groups. University Lecture, 2015.

[SvdW11] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *Arxiv preprint arXiv:1102.1323*, 2011.

[SVM04] Nigel Smart, Fre Vercauteren, and A. Muzereau. The equivalence between the dhp and dlp for elliptic curves used in practical applications. *LMS Journal of Computation and Mathematics*, 7:50–72, March 2004.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TH07] Laurent Théry and Guillaume Hanrot. Primality proving with elliptic curves. In *TPHOLs*, pages 319–333, 2007.

[Thé07] Laurent Théry. Proving the group law for elliptic curves formally. Technical Report RT-0330, INRIA, 2007.

[The10] The Coq Development Team. Chapter 4: Calculus of Inductive Constructions. Technical report, 2010.

[Wie06] Freek Wiedijk. Introduction. In Freek Wiedijk, editor, *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2006.

[ZBB16] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 296–309. IEEE Computer Society, 2016.

université PARIS-SACLAY
ÉCOLE DOCTORALE
INTERFACES
Approches Interdisciplinaires : Fondements, Applications et Innovation

A Formalization of Elliptic Curves for Cryptography. This thesis is in the domain of formalization of mathematics and of verification of cryptographic algorithms. The implementation of cryptographic algorithms is often a complicated task because cryptographic programs are optimized in order to satisfy both efficiency and security criteria. As a result it is not always obvious that a cryptographique program actually corresponds to the mathematical algorithm, i.e. that the program is correct. Errors in cryprtographic programs may be disastrous for the security of an entire cryptosystem, hence certification of their correctness is required. Formal systems and proof assistants such as Coq and Isabelle-HOL are often used to provide guarantees and proofs that cryptographic programs are correct. Elliptic curves are widely used in cryptography, mainly as efficient groups for asymmetric cryptography. To develop formal proofs of correctness for elliptic-curve schemes, formal theory of elliptic curves is needed.

Our motivation in this thesis is to formalize elliptic curve theory using the Coq proof assistant, which enables formal analysis of elliptic-curve schemes and algorithms. For this purpose, we used the Ssreflect extension and the mathematical libraries developed by the Mathematical Components team during the formalization of the Four Color Theorem. Our central result is a formal proof of Picard's theorem for elliptic curves: there exists an isomorphism between the Picard group of divisor classes and the group of points of an elliptic curve. An important immediate consequence of this proposition is the associativity of the elliptic curve group operation. Furthermore, we present a formal proof of correctness for the GLV algorithm for scalar multiplication on elliptic curve groups. The GLV algorithm exploits properties of the elliptic curve group in order to accelerate computation. It is composed of three independent algorithms: multiexponentiation on a generic group, decomposition of the scalar and computing endomorphisms on algebraic curves. This development includes theory about endomorphisms on elliptic curves and is more than 5000 lines of code. An application of our formalization is also presented.

Keywords: Cryptography, Formal methods (computer science), Elliptic curves, Coq (software)

Une formalisation des courbes elliptiques pour la cryptographie. Le sujet de ma thèse s'inscrit dans le domaine des preuves formelles et de la vérification des algorithmes cryptographiques. L'implémentation des algorithmes cryptographiques est souvent une tâche assez compliquée, parce qu'ils sont optimisés pour être efficaces et sûrs en même temps. Par conséquent, il n'est pas toujours évident qu'un programme cryptographique en tant que fonction, corresponde exactement à l'algorithme mathématique, c'est-à-dire que le programme soit correct. Les erreurs dans les programmes cryptographiques peuvent mettre en danger la sécurité de systèmes cryptographiques entiers et donc, des preuves de correction sont souvent nécessaires. Les systèmes formels et les assistants de preuves comme Coq et Isabelle-HOL sont utilisés pour développer des preuves de correction des programmes. Les courbes elliptiques sont largement utilisées en cryptographie surtout en tant que groupe cryptographique très efficace. Pour le développement des preuves formelles des algorithmes utilisant les courbes elliptiques, une théorie formelle de celles-ci est nécessaire. Dans ce contexte, nous avons développé une théorie formelle des courbes elliptiques en utilisant l'assistant de preuves Coq. Cette théorie est par la suite utilisée pour prouver la correction des algorithmes de multiplication scalaire sur le groupe des points d'une courbe elliptique.

Plus précisément, mes travaux de thèse peuvent être divisées en deux parties principales. La première concerne le développement de la théorie des courbes elliptiques en utilisant l'assistant des preuves Coq. Notre développement de plus de 15000 lignes de code Coq comprend la formalisation des courbes elliptiques données par une équation de Weierstrass, la théorie des corps des fonctions rationnelles sur une courbe, la théorie des groupes libres et des diviseurs des fonctions rationnelles sur une courbe. Notre résultat principal est la formalisation du théorème de Picard ; une conséquence directe de ce théorème est l'associativité de l'opération du groupe des points d'une courbe elliptique qui est un résultat non trivial à prouver. La seconde partie de ma thèse concerne la vérification de l'algorithme GLV pour effectuer la multiplication scalaire sur des courbes elliptiques. Pour ce développement, nous avons vérifier trois algorithmes indépendants : la multiexponentiation dans un groupe, la décomposition du scalaire et le calcul des endomorphismes sur une courbe elliptique. Nous avons également développé une formalisation du plan projectif et des courbes en coordonnées projectives et nous avons prouvé que les deux représentations (affine et projective) sont isomorphes.

Mon travail est à la fois une première approche à la formalisation de la géométrie algébrique élémentaire qui est intégré dans les bibliothèques de Ssreflect mais qui sert aussi à la certification de véritables programmes cryptographiques.

Mots-clés : Cryptographie, Méthodes formelles (informatique), Courbes elliptiques, Coq (logiciel)

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France