mathematics

MDPI

Article

Twisted Edwards Elliptic Curves for Zero-Knowledge Circuits

Marta Bellés-Muñoz [1,*], Barry Whitehat [2], Jordi Baylina [3], Vanesa Daza [1] and Jose Luis Muñoz-Tapia [4]

[1] Department of Information and Communications Technology, Pompeu Fabra University, Tànger Building, 08018 Barcelona, Spain; vanesa.daza@upf.edu
[2] Independent Researcher, 6300 Zug, Switzerland; barrywhitehat@protonmail.com
[3] 0KIMS, Eschenring 11, 6300 Zug, Switzerland; jordi@baylina.cat
[4] Department of Network Engineering, Campus Nord, Polytechnic University of Catalonia, 08034 Barcelona, Spain; jose.luis.munoz@upc.edu
* Correspondence: marta.belles@upf.edu

Abstract: Circuit-based zero-knowledge proofs have arose as a solution to the implementation of privacy in blockchain applications, and to current scalability problems that blockchains suffer from. The most efficient circuit-based zero-knowledge proofs use a pairing-friendly elliptic curve to generate and validate proofs. In particular, the circuits are built connecting wires that carry elements from a large prime field, whose order is determined by the number of elements of the pairing-friendly elliptic curve. In this context, it is important to generate an inner curve using this field, because it allows to create circuits that can verify public-key cryptography primitives, such as digital signatures and encryption schemes. To this purpose, in this article, we present a deterministic algorithm for generating twisted Edwards elliptic curves defined over a given prime field. We also provide an algorithm for checking the resilience of this type of curve against most common security attacks. Additionally, we use our algorithms to generate Baby Jubjub, a curve that can be used to implement elliptic-curve cryptography in circuits that can be validated in the Ethereum blockchain.

Keywords: zero-knowledge proof; elliptic curve; blockchain; privacy

Citation: Bellés-Muñoz, M.; Whitehat, B; Baylina, J.; Daza, V.; Muñoz-Tapia, J.L. Twisted Edwards Elliptic Curves for Zero-Knowledge Circuits. *Mathematics* **2021**, *9*, 3022. https://doi.org/10.3390/math9233022

Academic Editors: Ioana Boureanu and Liqun Chen

Received: 31 October 2021
Accepted: 19 November 2021
Published: 25 November 2021

1. Introduction

Since the first occurrence of an elliptic curve in one of Diophantus' Arithmetica books, elliptic curves have played an increasingly important role in mathematics. These mathematical objects showed their practical potential in the 1980s, when Koblitz [1] and Miller [2] independently proved that some techniques used in modern cryptography could also be applied to elliptic curve groups, and that the resulting schemes were more efficient. After four decades of research and development, elliptic-curve cryptography (ECC) now has widespread exposure and acceptance; and industry, banking, and government standards, have already migrated from classic public-key cryptography to ECC [3].

In 2008, blockchain, the technology behind most cryptocurrencies, was added to the list of practical uses for ECC. In Satoshi's seminal Bitcoin paper [4], ECC was used for securing the various transactions occurring on the network, for controlling the generation of new currency units, and verifying the transfer of digital assets and tokens. A few years later, privacy-oriented cryptocurrencies, such as Monero and Zcash, incorporated new cryptographic techniques to ensure user anonymity and obfuscate payment details. For instance, Monero started using ring signatures and Pedersen commitments [5], while Zcash exploited a type of protocols called zero-knowledge (ZK) proofs [6].

ZK proofs allow one party to convince another that a statement is true without revealing any information beyond the veracity of the statement [7]. In general, they are used to show that someone knows the result of a computation without revealing the solution. For example, with ZK proofs, one can prove that someone holds the private key associated with a certain public key without revealing the private key. Another example is

to prove that a transaction has been computed correctly without leaking any information about the transaction details.

Among different ZK systems, the most suitable for blockchain applications are the ones called ZK succinct non-interactive arguments of knowledge (ZK-SNARKs). This type of protocol does not require any interaction between the parties involved, and the size of the proofs is small [8–10]. Most ZK-SNARK constructions make use of bilinear pairings over elliptic curve groups to achieve efficient verification, too. In this way, blockchain users can send proofs as part of a transaction to a smart contract, who verifies if the proof is correct and performs a specific action depending on whether the proof verification is satisfied or not.

Before proving computational statements with ZK-SNARKs, statements have to be expressed as an $\mathbb{F}_p$-arithmetic circuit (also called ZK circuit or ZK-SNARK circuit), which is a circuit made up of additions and multiplications modulo a certain prime p. The specific construction of the protocol is what determines p, but typically p is a large prime number of approximately 254 bits that is determined by the order of a pairing-friendly elliptic curve [11]. For instance, in Ethereum, p is the order of BN256 elliptic curve, and in Zcash, p is the large prime order subgroup of BLS12-381.

Classical cryptographic schemes consisted mostly of boolean operations, which makes them inefficient when evaluated inside a ZK-SNARK circuit. As an example, the Zcash circuit relied on the SHA256 hash function to create a message-authentication code to prevent malleability, for generating pseudo-random strings and for commitments. However, each invocation of SHA256 added tens of thousands of multiplication gates to ZK-SNARK circuits, making this hash the primary cost when generating ZK-SNARK proofs [12]. These issues motivated the search for algebraic primitives to replace SHA256 and other inefficient functions. So, instead of hash functions such as SHA256 inside ZK-SNARK circuits, the idea was to use ECC that works in large prime fields, which is the natural representation of circuits. For this reason, new schemes that relied on elliptic curves were gradually adopted in ZK-SNARK circuits.

In this context, two prominent examples are Pedersen hashes [13] and the Edwards digital signature algorithm (EdDSA) [14]. These schemes can be built efficiently by using elliptic curves that can be represented in the twisted Edwards form.The nice feature of this form is that there is a single formula for doubling and adding points of the curve (Section 6, [15]).There is another form of representing elliptic curves called Montgomery, that makes computations faster but has different formulas for adding and doubling points [16]. The nice thing is that the twisted Edwards form is generally birationally equivalent to a Montgomery curve, so the curve can be easily converted from one form to another (Theorem 3.2, [15]). Inside our ZK-SNARK circuit, we can use the Montgomery form when we know for sure that either we are adding different points or we are adding the same point, and use twisted Edwards when, depending on the inputs of the circuit, this cannot be assured. Combining the two forms in this way makes the implementation of the group law inside ZK-SNARK circuits very efficient.

1.1. Motivation

In order to implement the Pedersen hash and the EdDSA inside a ZK-SNARK circuit, one needs curves that are defined over the finite field of prime order p, namely $\mathbb{F}_p$, where p is determined by the particular choice of pairing-friendly elliptic curve used to generate ZK-SNARK proofs. It is crucial to choose an appropriate twisted Edwards elliptic curve with optimal parameters for the cryptographic schemes, since the choice of the curve has great impact on their security and efficiency. Moreover, it is important to generate curves in a transparent and deterministic way, so that anyone can audit and recreate the procedure. Transparency is paramount, as it significantly reduces the possibility of a backdoor being present, thus leading to better security. Needless to say, it is crucial that the new curves are also tested for resilience against best known attacks, such as the rho method, or additive

and multiplicative transfers, which attack the discrete logarithm problem over elliptic curve groups [17].

1.2. Our Contributions

In this paper, we present a set of deterministic algorithms that, given a field $\mathbb{F}_p$, allows us to generate secure twisted Edwards elliptic curves that are suitable for $\mathbb{F}_p$-arithmetic circuits and allow the efficient computation of ZK-SNARK proofs that prove ECC statements. There have already been two curves that have been generated using these methods. On the one side, Jubjub curve for Zcash, defined over the scalar field of BLS12-381 and on the other side, Baby Jubjub for Ethereum, defined over the scalar field of BN256. The present work is a formalization and generalization of the common efforts to generate suitable twisted Edwards curves for ZK-SNARK circuits.

1.3. Organization of the Paper

The rest of the paper is structured in five main sections. In Section 2, we introduce related work on generation of elliptic curves in the ZK context. In the following Section 3, we provide a general overview of ZK proofs and classical theory of elliptic curves. In Section 4, we present a deterministic algorithm for generating twisted Edwards elliptic curves defined over a given prime field. We complement the work in Section 5, with the security checks that a twisted Edwards elliptic curve should pass. We base this section on the work from Bernstein and Lange gathered in [18]. In Section 6, we use our algorithms to generate Baby Jubjub, a curve that can be used to implement elliptic-curve schemes inside $\mathbb{F}_p$-arithmetic circuits for Ethereum, and which has already been used in practical applications such as Hermez, a Layer-2 payment system, and Tornado Cash, a payment mixer. Finally, we conclude in Section 7 with some discussion of the work and future research directions. Additionally, we provide Appendix A with a SAGE implementation of the security checks presented in Section 5, which was used to prove that Baby Jubjub was safe under best known security attacks.

2. Related Work

The interest of society and regulators in privacy, both individual's privacy and industry trade secrets, has grown in the recent years, and has ended up with new legislation such as the General Data Protection Regulation (GDPR) passed by the European Union [19] and the California Consumer Privacy Act (CCPA) [20]. Privacy is specially relevant in blockchain, where it is crucial to combine the inherent transparency of the system with the users' privacy rights [21–23]. To this purpose, in the blockchain space, there has been an intensive use of cryptographic schemes that make use of elliptic curves, which has motivated the appearance of new problems that have been tackled by both industry and academia.

For instance, the need for efficient pairing-friendly curves in ZK-SNARK schemes resurfaced the work from Barreto and Naehrig [24], and Barreto, Lynn, and Scott [25], who developed techniques to generate pairing-friendly elliptic curves that had an optimal Ate pairing. The vast application of curves derived from their work, which are usually called BN and BLS curves, resulted in undergoing processes from the IRTF Crypto Forum Research Group to standardize particular instantiations of these curves [26,27], such as the BN-256 and the BLS12-381, which are used in digital signature schemes and ZK-SNARK protocols all over the Internet.

The order of these curves is what determines the type of statements we can prove using ZK. More precisely, the largest prime p dividing the curve's order fixes the field in which we can do modular arithmetic. As a result, computational statements that involve elliptic-curve operations can only be proved efficiently with curves that are defined over the prime field $\mathbb{F}_p$. Hence, the implementation of ECC schemes that make use of twisted Edwards curves require new curves defined over $\mathbb{F}_p$. Table 1 summarizes the most closely related contributions to our work.

Table 1. Comparison with related work.

System	Outer Curve	Inner Curve
Zcash [12]	BLS12-381	Jubjub
Masson et al. [28]	BLS12-381	Bandersnatch
Our Proposal	BN-256	Baby Jubjub
Ben-Sasson et al.[29]	MNT4	MNT6
ZEXE [30]	CP6-782	BLS12-377
Housni et al. [31]	BW6-761	BLS12-377

The Zcash team was the first to generate a suitable curve for $\mathbb{F}_p$-arithmetic circuits. Since Zcash ZK-SNARK constructions are based on BLS12-381, their curve, named Jubjub, was expressly built over the BLS12-381 scalar field.

Recently, a new elliptic curve built over the BLS12-381 scalar field was introduced in [28], but although this curve allows a faster scalar multiplication algorithm than Jubjub, it does not provide any performance improvement in multi-scalar multiplications or in the ZK circuit representations.

We have used a similar approach to generate Baby Jubjub, which is an embedded elliptic curve designed to operate on the field produced by the BN256 elliptic curve. This work that we are presenting covers the case in which we want to use circuits that can verify public key cryptography primitives such as digital signatures and encryptions in Ethereum. Then, these proofs can be verified by an Ethereum smart contract because the Ethereum virtual machine (EVM) has an operation to compute pairings with the BN256 elliptic curve. We presented and discussed the techniques used to generate Jubjub and Baby Jubjub at the second annual ZKProof Standardization Workshop in Berkeley. The present work is a reviewed, formalized and generalized version of the efforts towards the standardization of the generation of suitable twisted Edwards curves for ZK-SNARK circuits.

The generation of other type of curves for ZK-SNARK circuits has also appeared in other lines of research. For instance, the authors of [29] presented the first practical setting of recursive proof composition with a cycle of two Miyaji–Nakabayashi–Takano (MNT) pairing-friendly elliptic curves [32]. The idea of their proposal is that proofs generated from one curve can feasibly reason about proofs generated from the other curve. To achieve this, one curve's order is the other curve's base field order and vice versa. Although current MNT cycles of curves are quite expensive at the 128-bit security, the work opens the door to the possibility of having succinct blockchains that are verifiable with one single proof. Bowe et al. [30] proposed ZEXE, a construction that follows a relatively relaxed approach to find a suitable pair of curves that form a chain rather than a cycle. A later work from El Housni and Guillevic [31] improved ZEXE with a new curve that makes the verification of composed ZK-SNARK proofs significantly faster.

3. Background

In this section, we review the main ideas behind ZK proving systems and elliptic curve theory, focusing on twisted Edwards and Montgomery forms.

3.1. Zero-Knowledge Proofs

Zero-knowledge (ZK) proofs allow one party, called prover, to convince another one, called verifier, that a statement is true without revealing any information beyond the veracity of the statement. In this context, we understand a statement as a relation between an instance, a public input known to both prover and verifier, and a witness, a private input only known by the prover, which belongs to a language $\mathcal{L}$ in the nondeterministic polynomial time (NP) complexity class [7]. More specifically, a ZK protocol satisfies that a verifier will accept the proof if it is generated by an honest prover (completeness) and the verifier will not accept proofs from a dishonest prover (soundness). These two properties protect the verifier against dishonest provers. On the other hand, ZK proofs

must ensure that the proofs do not leak any information about the statement being proved (zero-knowledge), guaranteeing this way the privacy of the prover's secret information against malicious verifiers [7].

Most ZK systems operate in the model of $\mathbb{F}_p$-arithmetic circuits, which are circuits composed by wires that carry values from a prime field $\mathbb{F}_p$ and connect them to addition and multiplication gates modulo p. We consider that an assignment to the wires is valid if for every gate, the value on the output wires matches that gate's operation and the values on its input wires. In this context, a ZK proof typically allows to prove the existence of a valid circuit wires assignment, but there is a special class of protocols known as arguments of knowledge that allow to prove that, with very high probability, the prover knows a valid assignment to the wires of the circuit.

An argument of knowledge is considered a ZK succinct non-interactive argument of knowledge (ZK-SNARK) if there is no interaction between the prover and the verifier during the generation of a proof and, regardless of the size of the statement being proved, has succinct proof size (e.g., [9]—proofs are ≈ 200 bytes). Most ZK-SNARKs also guarantee short verification time by making use of bilinear pairings over groups of points of an elliptic curve [8–10]. The largest prime p diving of the curve is what determines the modular arithmetic performed in the arithmetic circuits. For instance, a ZK-SNARK protocol that uses pairings over an elliptic curve of prime order p allows proving statements regarding $\mathbb{F}_p$-arithmetic circuits.

3.2. Elliptic Curves

In this section, we first give a general overview of elliptic curves defined over any field K, following the definitions and notation in [33] (Chapter VI), and then, we focus on twisted Edwards and Montgomery elliptic curves defined over prime finite fields. For general results about elliptic curves, we refer the reader to [34,35].

Definition 1 (Elliptic curve)**.** *Let K be a field of characteristic $\neq 2, 3$ and let*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

be a cubic polynomial equation with $a_1, \ldots, a_6 \in K$ and with no multiple roots. An elliptic curve E over K consists of the set of points (x, y) with $x, y \in K$ which satisfy Equation (1), together with a single element called point at infinity, denoted by $\mathcal{O}$.

An important result about the set of points of an elliptic curve is that they form an additive Abelian group. To define the addition of points, which might seem a little unnatural at first, it is easier to think of elliptic curves as objects in the two-dimensional projective space.

Group Law 1. *Let E be an elliptic curve and P, Q points on E. Let $L \subset \mathbb{P}^2$, where $\mathbb{P}^2$ denotes the 2-dimensional projective space, be the line that intersects both P and Q and denote by R the third point of intersection between E and L. Let L' be the line that intersects both R and $\mathcal{O}$. We define $P + Q$ as the third point of intersection between E and L'.*

The following example illustrates visually the composition of points in the plane of an ordinary curve defined over the field of real numbers $\mathbb{R}$.

Example 1. *Let E be the elliptic curve $E : y^2 = x^3 - 2x$ defined over $\mathbb{R}$. In Figure 1, we see the composition $P + Q$ for two points P and Q with distinct y-coordinates. The line through the points intersects the curve in a third point R and $P + Q$ is the reflection across the x-axis.*

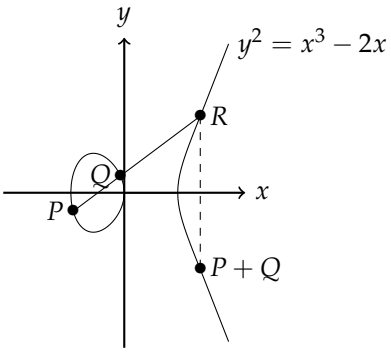

Figure 1. The Group Law for two distinct points P, Q on the elliptic curve $E : y^2 = x^3 - 2x$.

Proposition 1. *The composition of points on an elliptic curve E defined in Group Law 1 makes E into an additive Abelian group with identity* $\mathcal{O}$.

Proof. See (Proposition 2.2, Chapter III, [34]). □

In cryptography, we are interested in elliptic curves over finite fields. For the rest of the article, we shall let K be a prime-order finite field $\mathbb{F}_p$ for some prime p. In this context, it makes sense to define the order of the curve and the order of a point.

Definition 2 (Order)**.** *The order of an elliptic curve defined over* $\mathbb{F}_p$ *is its number of points. The order of a point P is the smallest positive integer n such that* $nP = \mathcal{O}$.

The following result is a particular case of Hasse's theorem, which provides an upper and a lower bound of the order of an elliptic curve.

Theorem 1 (Hasse's Bound Theorem)**.** *Let n be the number of points on an elliptic curve defined over* $\mathbb{F}_p$. *Then,*

$$|n - (p+1)| \leq 2\sqrt{p}.$$

Proof. See (Theorem 1.1, Chapter V, [34]). □

Definition 3. *Let n be the order of an elliptic curve E. We say that a point* G_0 *is a generator of the curve, if it has order n. When n is a composite number of the form* $n = h \times l$, *where h is a small number (typically called cofactor) and l is a large prime number, we say that* G_1 *is a base point, if it has order l.*

Montgomery and Twisted Edwards Curves

Below, we define and describe elliptic curves in Montgomery and twisted Edwards form. In this part, we follow [15,36].

Definition 4 (Montgomery curve)**.** *Let* $p \geq 3$ *be a prime and* $\mathbb{F}_p$ *the finite field of order p. For* $A, B \in \mathbb{F}_p$, $A \in \mathbb{F}_p \backslash \{-2, 2\}$ *and* $B \in \mathbb{F}_p \backslash \{0\}$, *an elliptic curve defined by*

$$E^M : By^2 = x^3 + Ax^2 + x$$

is called a Montgomery (elliptic) curve.

The following theorem presents the addition formulas for Montgomery curves.

Theorem 2. *Let* $P_1 = (x_1, y_1) \neq \mathcal{O}$ *and* $P_2 = (x_2, y_2) \neq \mathcal{O}$ *be two points of a Montgomery curve* E^M. *Then:*

- *If $P_1 \neq P_2$, then the sum $P_1 + P_2$ is a third point $P_3 = (x_3, y_3)$ with coordinates*

$$\begin{aligned} \Lambda &= (y_2 - y_1)/(x_2 - x_1), \\ x_3 &= B\Lambda^2 - A - x_1 - x_2, \\ y_3 &= \Lambda(x_1 - x_3) - y_1. \end{aligned} \tag{2}$$

- *If $P_1 = P_2$, then $P_1 + P_1$ is a point $P_3 = (x_3, y_3)$ with coordinates*

$$\begin{aligned} \Lambda &= (3x_1^2 + 2Ax_1 + 1)/(2By_1), \\ x_3 &= B\Lambda^2 - A - 2x_1, \\ y_3 &= \Lambda(x_1 - x_3) - y_1. \end{aligned} \tag{3}$$

Proof. See [16]. □

Theorem 3. *The order of a Montgomery curve is divisible by 4.*

Proof. See (Section 10.3.2, [36]). □

Definition 5 (Twisted Edwards curve)**.** *Let $p \geq 3$ be a prime and $\mathbb{F}_p$ the finite field of order p. For distinct $a, b \in \mathbb{F}_p \backslash \{0\}$, an elliptic curve defined by*

$$E : ax^2 + y^2 = 1 + dx^2y^2$$

is called a twisted Edwards (elliptic) curve.

As the next theorem shows, twisted Edwards curves have complete addition formulas, which makes these curves very efficient to implement inside ZK-SNARK circuits.

Theorem 4. *Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points of a twisted Edwards elliptic curve E. The sum $P_1 + P_2$ is a third point $P_3 = (x_3, y_3)$ with*

$$\begin{aligned} \lambda &= dx_1x_2y_1y_2, \\ x_3 &= (x_1y_2 + y_1x_2)/(1 + \lambda), \\ y_3 &= (y_1y_2 - x_1x_2)/(1 - \lambda). \end{aligned}$$

Note that the inverse of a point (x, y) in a twisted Edwards curve is $(-x, y)$.

Proof. See ([37], Section 3). □

The following theorem states that Montgomery and twisted Edwards curves are birationally equivalent. The theorem also gives the birational map that allows the transformation from one form to the other.

Theorem 5. *Every twisted Edwards curve E over $\mathbb{F}_p$ is birationally equivalent over $\mathbb{F}_p$ to a Montgomery curve E^M with parameters*

$$A = 2\,\frac{a + d}{a - d} \quad \text{and} \quad B = \frac{4}{a - d}.$$

The birational equivalence from E to E^M is the map

$$(x, y) \to (u, v) = \left(\frac{1 + y}{1 - y}, \frac{1 + y}{(1 - y)x}\right)$$

with inverse

$$(u, v) \to (x, y) = \left(\frac{u}{v}, \frac{u-1}{u+1}\right). \tag{4}$$

Conversely, every Montgomery curve over $\mathbb{F}_p$ *is birationally equivalent over* $\mathbb{F}_p$ *to a twisted Edwards curve with parameters*

$$a = \frac{A+2}{B} \quad \text{and} \quad d = \frac{A-2}{B}.$$

Proof. See (Theorem 3.2, [15]). □

4. Our Proposal

In this section, we present a method that, given a prime number p, we get a twisted Edwards curve defined over $\mathbb{F}_p$.

4.1. General Overview

Our algorithm takes a prime number p and returns a twisted Edwards curve defined over $\mathbb{F}_p$. More precisely, the specific outputs of the algorithm are:

- The prime order of the finite field the curve is defined over (which is the input p);
- Parameters a and d of the equation that defines the twisted Edwards curve;
- The order of the curve and its decomposition into the product of a cofactor and a large prime;
- A generator and a base point for the curve.

Since the finite field is defined by the input p, no specification of this parameter is required. The order of the curve and its decomposition are also determined once the parameters of the equation describing the curve are fixed. Hence, the only remaining specifications are parameters a and d and the choice of generator and base point.

We have divided the procedure in four steps:

1. *Choice of Montgomery Equation*: we start by deterministically generating a Montgomery elliptic curve E^M over $\mathbb{F}_p$;
2. *Choice of Generator and Base Points*: we set the generator and base points of E^M;
3. *Transformation to Twisted Edwards*: we convert the curve E^M to its birationally equivalent twisted Edwards form and the generator and base points using the maps from Theorem 5;
4. *Optimization of Parameters*: if possible, we rescale all parameters so that the arithmetic in the curve can be sped up [38].

All algorithms presented in this section have been implemented in SAGE programming language, and are presented in Section 6.

4.2. Choice of Montgomery Equation

We start by finding a Montgomery curve defined over $\mathbb{F}_p$ where p is a given prime number. The assumptions and algorithm presented are based on the work of [39] and Zcash team [40].

Algorithm 1 takes a prime p, fixes $B = 1$ and returns the Montgomery elliptic curve defined over $\mathbb{F}_p$ with the smallest coefficient A such that $A - 2$ is a multiple of 4. This approach comes from the fact that, when defining a Montgomery curve, the smaller A is, the faster the group operation becomes. More precisely, as pointed out in [41], for the best performance, we need $(A-2)/4$ to be small. As with $A = 1$ and $A = 2$, a twisted Edwards equation does not describe a smooth curve, so the algorithm starts with $A = 3$.

For primes congruent to 1 modulo 4, the minimal cofactors of the curve and its twist are either $\{4, 8\}$ or $\{8, 4\}$. We choose a curve with the latter cofactors, so that any algorithms that take the cofactor into account do not have to worry about checking for points on the

twist, because the twist cofactor will be the smaller of the two [39]. For a prime congruent to 3 modulo 4, both the curve and twist cofactors can be 4, and this is minimal.

Algorithm 1: Generation of E^M

Input: prime number p
Output: coefficients A, B, order n, cofactor h, prime l
1 **fix** $B = 1$.
2 **start** with $A = 3$.
3 **if** $(A-2) = 0 \mod 4$:
4 **continue**.
5 **else** :
6 **increment** A by 1 and go back to line 3.
7 **if** equation $y^2 = x^3 + Ax^2 + x$ defines an elliptic curve over $\mathbb{F}_p$:
8 **continue**.
9 **else** :
10 **increment** A by 1 and go back to line 3.
11 **compute** the group order n and cofactor h.
12 **if** $p = 1 \mod 4$:
13 **if** (cofactor is 8 **and** cofactor of twist is 4) :
14 **set** $h = 8$.
15 **else** :
16 **increment** A by 1 and go back to line 3.
17 **if** $p = 3 \mod 4$:
18 **if** (cofactor **and** cofactor of twist is 4) :
19 **set** $h = 4$.
20 **else** :
21 **increment** A by 1 and go back to line 3.
22 **compute** $l = n/h$.
23 **return** A, B, n, h and l.

4.3. Choice of Generator and Base Points

To pick a generator G_0^M of the curve, we choose the smallest element of $\mathbb{F}_p$ that corresponds to an x-coordinate of a point in the curve of order n. Then, as a base point, we define $G_1^M = 8 \cdot G_0^M$, which has order l. The steps are written down in Algorithm 2.

Algorithm 2: Generator and Base Points of E^M

Input: Montgomery curve E^M, order n, cofactor h
Output: generator G_0^M, base point G_1^M
1 **start** with $u = 1$.
2 **find** v such that (u, v) is a point of E^M. **else**, increment u by 1 and repeat the step.
3 **check** that (u, v) has order n. **else**, increment u by 1 and go back to step 2.
4 **set** $G_0^M = (u, v)$ and $G_1^M = h \cdot G_0$.
5 **return** G_0^M and G_1^M.

4.4. Transformation to Twisted Edwards

In Algorithm 3, we use the birational map from Equation (4) to get the coefficients, generator and base points in the twisted Edwards form.

4.5. Optimization of Parameters

As pointed out in [38] (Section 3.1), if $-a$ is a square in $\mathbb{F}_p$, it is possible to optimize the number of operations in a twisted Edwards curve by scaling it.

Theorem 6. *Consider a twisted Edwards curve defined over $\mathbb{F}_p$ given by equation $ax^2 + y^2 = 1 + dx^2y^2$. If $-a$ is a square in $\mathbb{F}_p$, then the map $(x, y) \to (x/\sqrt{-a}, y)$ defines the curve $-x^2 + y^2 = 1 + (-d/a)x^2y^2$. We denote by $f = \sqrt{-a}$ the scaling factor.*

Proof. The result follows directly from the map's definition. □

Algorithm 3: convert E^M to E

Input: Montgomery coefficients A, B, generator $G_0^M = (x_0^M, y_0^M)$, base $G_1^M = (x_1^M, y_1^M)$
Output: twisted Edwards coefficients a, d, generator G_0, base point G_1

compute $a = (A+2)/B$ and $d = (A-2)/B$.
compute $x_0 = x_0^M / y_0^M$.
compute $y_0 = (x_0^M - 1)/(x_0^M + 1)$.
set $G_0 = (x_0, y_0)$.
compute $x_1 = x_1^M / y_1^M$.
compute $y_1 = (x_1^M - 1)/(x_1^M + 1)$.
set $G_1 = (x_1, y_1)$.
return a, d, G_0 and G_1.

The following Algorithm 4 rescales, if possible, the twisted Edwards curve found in the previous step as described in the previous theorem. It also converts the generator and base points to the new coordinates. After applying the algorithm, the map that transforms E^M to E becomes the composition of maps from Theorems 5 and 6.

Algorithm 4: if possible, rescale E with $a = -1$

Input: coefficients a, d, generator $G_0 = (x_0, y_0)$, base point $G_1 = (x_1, y_1)$
Output: scaling factor f, coefficients $a' = a/f^2$, $d' = -d/a$, generator $G_0' = (x_0/f, y_0)$, base point $G_1' = (x_1/f, y_1)$

if $-a$ is a square in $\mathbb{F}_p$:
    take $f = \sqrt{-a}$.
    set $a' = -1$ and $d' = -d/a$.
    compute $x_0' = x_0/f$ and $x_1' = x_1/f$.
    set $G_0' = (x_0', y_0)$ and $G_1' = (x_1', y_1)$.
    return f, a', d', G_0' and G_1'.
else :
    set $f = 1$.
    return f, a, d, G_0 and G_1.

5. Security Analysis

This section specifies the safety criteria that the elliptic curve should satisfy. The choices of security parameters are based on the joint work of Bernstein and Lange summarized in [18]. In Appendix A, we provide an implementation of the algorithm that should be run after finding the elliptic curve as proposed in the previous section. The algorithm is based on the code from [42], which is an extension of the original SAGE code from [18], to general twisted Edwards curves.

Curve Parameters: We check that all given parameters describe a well-defined elliptic curve over a prime finite field:

- The given number p is prime;
- The given parameters define an equation that corresponds to an elliptic curve;
- The product of h and l results into the order of the curve and the point G_0 is a generator;
- The given number l is prime and the point G_1 is a base point.

Elliptic Curve Discrete Logarithm Problem: We check that the discrete logarithm problem remains difficult in the given curve. For that, we check it is resistant to the following known attacks:

- Rho method (Section V.1, [17]): we require the cost for the rho method, which takes on average around $0.886\sqrt{l}$ additions, to be above 2^{100};
- Additive and multiplicative transfers (Section V.2, [17]): we require the embedding degree to be at least $(l-1)/100$;

- High discriminant (Section IX.3, [17]): we require the complex-multiplication field discriminant D to be larger than 2^{100}.

  Elliptic Curve Cryptography: We check if the curve is suitable for ECC:

- Ladders [16]: check the curve supports the Montgomery ladder;
- Twists (twist, [18]): check if it is secure against the small-subgroup attack, invalid-curve attacks and twisted-attacks;
- Completeness (complete, [18]): check if the curve has complete single-scalar and multiple-scalar formulas. It is enough to check that there is only one point of order 2 and 2 of order 4;
- Indistinguishability [43]: check availability of maps that turn elliptic-curve points indistinguishable from uniform random strings.

6. Implementation

Ethereum, the second-largest blockchain, uses BN256 to generate and verify ZK-SNARK proofs. BN256 is a pairing-friendly elliptic curve of prime order

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617.$$

In order to prove ECC statements with ZK-SNARKs, Ethereum needed a new curve defined over $\mathbb{F}_p$. In this section, we present a SAGE implementation of the algorithms presented in the previous sections, and we use them to generate *Baby Jubjub*, a twisted Edwards elliptic curve suitable for ZK-SNARK circuits in Ethereum.

First, we implemented Algorithm 1 in Listing 1, which generates a Montgomery curve E^M with the smallest A satisfying the conditions we described in Section 4.2. In the last lines of code, we instantiate the functions using the prime number p, which is the order of BN256 curve, and enforcing the resulting curve to have cofactor $h = 8$.

```
def findCurve(prime, curveCofactor, twistCofactor, _A):
    Fp = GF(prime)
    A = _A
    while A < _A + 200000:
        if (A-2.) % 4 != 0:
            A+=1.
            continue
        try:
            E = EllipticCurve(Fp, [0, A, 0, 1, 0])
        except:
            A+=1.
            continue

        groupOrder = E.order()
        if (groupOrder % curveCofactor != 0 or not is_prime(groupOrder // curveCofactor)):
            A+=1
            continue

        twistOrder = 2*(prime+1)-groupOrder
        if (twistOrder % twistCofactor != 0 or not is_prime(twistOrder // twistCofactor)):
            A+=1
            continue

        return E, A, 1, groupOrder, curveCofactor, groupOrder // curveCofactor

def find1Mod4(prime, curveCofactor, twistCofactor, A):
    assert((prime % 4) == 1)
    return findCurve(prime, curveCofactor, twistCofactor, A)

# Baby Jubjub in Montgomery form
prime = 21888242871839275222246405745257275088548364400416034343698204186575808495617
Fp = GF(prime)
h = 8
A = 1.
EC, A, B, n, h, l = find1Mod4(prime, h, 4, A)
```

Listing 1. Generation of E^M.

The result from Listing 1 is that the smallest A satisfying our conditions is A = 168,698. As a result, the Montgomery form of Baby Jubjub is defined over $\mathbb{F}_p$ by equation:

$$y^2 = x^3 + 168{,}698x^2 + x.$$

The function `findCurve` also returns the order of the curve, which in this case is

$$n = 21888242871839275222246405745257275088614511777268538073601725287587578984328,$$

where $n = h \times l$, with $h = 8$, and l is the large prime number

$$l = 2736030358979909402780800718157159386076813972158567259200215660948447373041.$$

The next step is to generate a generator and a base point for Baby Jubjub in Montgomery form. For that, we implemented Algorithm 2 in Listing 2. Recall that the algorithm is deterministic and takes as a generator G_0^M, the point of the curve of order n with smallest x-coefficient, and as a base point, $G_1^M = h \cdot G_0^M$. The last two lines of code are used to find a valid generator and base point for Baby Jubjub in Montgomery form.

```
def findGenPoint(prime, A, EC, N):
    Fp = GF(prime)
    for uInt in range(1, 1e3):
        u = Fp(uInt)
        v2 = u^3 + A*u^2 + u
        if not v2.is_square():
            continue
        v = v2.sqrt()

        point = EC(u, v)
        pointOrder = point.order()
        if pointOrder == N:
            return point

def findBasePoint(EC, h, u, v):
    return h*EC(u, v)

# Generator and base points of Baby Jubjub in Montgomery form
gen_u, gen_v, gen_w = findGenPoint(prime, A, EC, n)
base_u , base_v, base_w = findBasePoint(EC, h, gen_u, gen_v)
```

Listing 2. Generator G_0 and base point G_1 of E^M.

The resulting points from Listing 2 are the generator $G_0^M = (x_0^M, y_0^M)$ with coordinates

$$x_0^M = 7,$$
$$y_0^M = 4258727773875940690362607550498304598101071202821725296872974770776423442226,$$

and the base point $G_1^M = (x_1^M, y_1^M)$ with coordinates

$$x_1^M = 7117928050407583618111176421555214756675765419608405867398403713213306743542,$$
$$y_1^M = 1457726821888189942096677968769020542522743157772865981997519849112717931562 6.$$

Algorithm 3 maps a Montgomery curve to its twisted Edwards form. We divided the algorithm in three different functions. The first function `mont_to_ted` converts a Montgomery point to a twisted Edwards point, the function `ted_to_mont` does the opposite, and `is_on_ted` checks if a point is a solution to a given twisted Edwards equation. Although the last two functions are not needed in the original algorithm, we implemented them in order to have sanity checks after the conversion from Montgomery to twisted Edwards form.

After the conversion maps, we get that the twisted Edwards form of Baby Jubjub is described by equation

$$168700x^2 + y^2 = 1 + 168696x^2y^2.$$

The code from Listing 3 also outputs the generator $G_0 = (x_0, y_0)$ and base point $G_1 = (x_1, y_1)$ in twisted Edwards form. The specific outputs are that G_0 has coordinates

$$x_0 = 995203441582195749578291179787384436505546430278305826713579947235728471134,$$
$$y_0 = 547206071795981880556160143631431877213709110010400858592455104664395212390 5,$$

and G_1 has coordinates

$$x_1 = 529961924064155128163486558351829703028287447219077289408652114448272100 1553,$$
$$y_1 = 169501507984606577179586255678218345503016631616247077872228159361826389682 03.$$

```
def mont_to_ted(u, v , prime):
    Fp = GF(prime)
    x = Fp(u / v)
    y = Fp((u-1)/(u+1))
    return(x, y)

def ted_to_mont(x, y , prime):
    Fp = GF(prime)
    u = Fp((1 + y )/ ( 1 - y))
    v = Fp((1 + y ) / ( (1 - y) * x))
    return(u, v)

def is_on_ted(x, y, prime, a, d):
    Fp = GF(prime)
    return Fp(a*(x**2) + y**2 - 1 - d*(x**2)*(y**2)) == 0

# Conversion of Baby Jubjub to twisted Edwards
a = Fp((A + 2) / B)
d = Fp((A - 2) / B)

# Check we have a safe twist and discriminant != 0
assert(not d.is_square())
assert(a*d*(a-d)!=0)

# Conversion of generator to twisted Edwards
gen_x, gen_y = mont_to_ted(gen_u, gen_v, prime)
assert(is_on_ted(gen_x, gen_y, prime, a , d))

# Sanity check: the inverse map returns the original point in Montgomery
u , v = ted_to_mont(gen_x, gen_y, prime)
assert (u == gen_u)
assert (v == gen_v)

# Conversion of base point to twisted Edwards
base_x , base_y = mont_to_ted(base_u, base_v, prime)
assert(is_on_ted(base_x,base_y, prime , a , d))
```

Listing 3. Conversion of E^M to E.

The last Algorithm 4 tries to escalate the twisted Edwards form of the curve, so that the equation has parameter $a = -1$. This last step is implemented in Listing 4 and in the case of Baby Jubjub, the resulting scaling factor is

$$f = 191198285430522507438125134410332993163761020901489688989116827585546665709 0.$$

This way, the optimal version of Baby Jubjub in twisted Edwards form is given by equation

$$-x^2 + y^2 = 1 + d'x^2y^2,$$

where

$$d' = 121816440234217301248741585216995556817642491809499741106172910176006491288 46.$$

```
def scaling(a, d, prime):
    Fp = GF(prime)
    if Fp(-a).is_square():
        f = sqrt(Fp(-a));
        a_ = Fp(a / (f*f));
        d_ = Fp(d / (-a));
        if a_ == Fp(-1):
            a_ = -1
        else:
            a_ = a;
            d_ = a;
    return a_, d_, f

def ted_to_tedprime(x, y, prime, scaling_factor):
    Fp = GF(prime)
    x_ = Fp(x * (-scaling_factor))
    y_ = y;
    return(x_, y_)

def tedprime_to_ted(x_, y_, prime, scaling_factor):
    Fp = GF(prime)
    x = Fp(x_ / (-scaling_factor))
    y = y_
    return(x, y)

def is_on_ted_prime(x, y, prime, a_, d_):
    Fp = GF(prime)
    return Fp(a_*(x**2) + y**2 - 1 - d_*(x**2)*(y**2)) == 0

# Conversion of E to E'
a_, d_, f = scaling (a, d, prime)

# Conversion of generator to E'
gen_x_, gen_y_ = ted_to_tedprime(gen_x, gen_y, prime, f);
assert(is_on_ted_prime(gen_x_, gen_y_, prime, a_ , d_))

# Sanity check: the inverse map returns the original point in E
u , v = tedprime_to_ted(gen_x_prime, gen_y_prime, prime, f)
assert (u == gen_x)
assert (v == gen_y)

# Conversion of base point to E'
base_x_, base_y_ = ted_to_tedprime(base_x, base_y, prime, f);
assert(is_on_ted_prime(base_x_,base_y_, prime , a_ , d_))

# Sanity check: the inverse map returns the original point in E
u , v = tedprime_to_ted(base_x_prime, base_y_prime, prime, f)
assert (u == base_x)
assert (v == base_y)
```

Listing 4. Scaling of E to E'.

After generating Baby Jubjub, we checked that the curve passed all safety checks described in Section 5. The security evidence is shown in [44]. The determinism and transparency of the procedure allows any party to reproduce the generation of the curve and ensure its resilience against best-known security attacks [45].

Baby Jubjub curve was accepted as an Ethereum Improvement Proposal [11], and it is currently being used in several projects running over Ethereum, such as Hermez, a Layer-2 payment system, and Tornado Cash, a payment mixer.

7. Discussion

In the recent years, ZK proofs arose as a potential solution to blockchain privacy and scalability issues and, today, we can see many zero-knowledge protocols integrated and deployed in various blockchain projects. The use of cryptography in the blockchain space arises new efficiency needs that motivate novel lines of research that combine theoretical and practical aspects. In particular, the recent implementation of ZK protocols has had a huge impact in the interest for generating types of curves with special properties. The correct and transparent generation of new elliptic curves is paramount to the

success of cryptographic primitives that can help blockchains improve their privacy and scalability guarantees.

In this paper, we presented a deterministic algorithm for generating twisted Edwards elliptic curves defined over a given prime field. We also provided an algorithm for checking the safety of a curve against best known security attacks. Additionally, we gave an example that puts theory into practice: we detailed the generation of the twisted Edwards curve Baby Jubjub, which is currently being deployed and used in several projects based on Ethereum blockchain. The generation of other types of curves, such as pairing-friendly and prime order curves and efficient pairs of cyclic curves, which can help build more efficient ZK-SNARK schemes and opens the door to the creation of succinct blockchains, remains as future work.

Author Contributions: Conceptualization, M.B.-M. and B.W.; methodology, M.B.-M. and B.W.; software, B.W.; validation, M.B.-M.; formal analysis, M.B.-M.; investigation, B.W.; resources, B.W. and J.B.; writing—original draft preparation, M.B.-M.; writing—review and editing, M.B.-M., V.D. and J.L.M.-T.; supervision, V.D. and J.L.M.-T.; project administration, J.B., V.D. and J.L.M.-T.; funding acquisition, J.B. and V.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research has been partially funded by the projects Project RTI2018-102112-B-100 (AEI/FEDER, UE), i3Market (H2020-ICT-2019-2 grant number 871754) and TCO-RISEBLOCK (PID2019-110224RB-I00).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BN	Barreto–Naehrig
BLS	Barreto–Lynn–Scott
CCPA	California Consumer Privacy Act
ECC	Elliptic-Curve Cryptography
EdDSA	Edwards Digital Signature Algorithm
EVM	Ethereum Virtual Machine
GDPR	General Data Protection Regulation
IRTF	Internet Research Task Force
MNT	Miyaji–Nakabayashi–Takano
NP	Nondeterministic Polynomial Time
ZK	Zero-Knowledge
ZK-SNARK	Zero-Knowledge Succinct Non-interactive Argument of Knowledge

List of Mathematical Symbols

The following mathematical symbols are used in this manuscript:

p	Prime number
$\mathbb{F}_p$	Finite field of order p
E	Elliptic curve in twisted Edwards form
E^M	Elliptic curve in Montgomery form
n	Order of the elliptic curve
l	Largest prime divisor of n
h	Cofactor
G_0	Generator of E
G_0^M	Generator of E^M
G_1	Base point of E
G_1^M	Base point of E^M

Appendix A. Safety Checks against Known Attacks

We provide an implementation of an algorithm based on the code from [42], that checks if a given twisted Edwards curve is safe against the attacks described in Section 5.

```
# It outputs all results in the console.

import os
import sys
from errno import ENOENT, EEXIST
from sortedcontainers import SortedSet

def readfile(fn):
    fd = open(fn,'r')
    r = fd.read()
    fd.close()
    return r

# It expresses n as sums or differences of sparse powers of 2 (if possible).
def expand2(n):
    s = ""

    while n != 0:
        j = 1
        while 2**j < abs(n): j += 1
        if 2**j - abs(n) > abs(n) - 2**(j-1): j -= 1

        if abs(abs(n) - 2**j) > 2**(j - 1):
            if n > 0:
                if s != "": s += " + "
                s += str(n)
            else:
                s += " - " + str(-n)
            n = 0
        elif n > 0:
            if s != "": s += " + "
            s += "2^" + str(j)
            n -= 2**j
        else:
            s += " - 2^" + str(j)
            n += 2**j

    return s

def verify(curve):

    p = Integer(readfile(curve+'/p')) # Prime p.
    k = GF(p) # Finite field F_p.
    kz.<z> = k[] # Polynomial ring k[z].
    l = Integer(readfile(curve+'/l')) # Large prime l dividing |E(F_p)|.
    x0 = Integer(readfile(curve+'/x0')) # (x0,y0): generating point of E.
    y0 = Integer(readfile(curve+'/y0'))
    x1 = Integer(readfile(curve+'/x1')) # (x1,y1): base point of E[l].
    y1 = Integer(readfile(curve+'/y1'))
    shape = readfile(curve+'/shape').strip()
    s = readfile(curve+'/primes')
    rigid = readfile(curve+'/rigid').strip()

    safefield = True
    safeeq = True
    safebase = True
    saferho = True
    safetransfer = True
    safedisc = True
    saferigid = True
    safeladder = True
    safetwist = True
    safecomplete = True
    safeind = True

    V = [] # Distinct verified primes.
    for line in s.split():
        n = Integer(line)
        if n.is_prime(): # Instead of generating the original Pocklington primality proofs.
            if not n in V: V += [n]

    # Verify p is prime.
    pstatus = 'Unverified'
    if not p.is_prime(): pstatus = 'False'
    if p in V: pstatus = 'True'
    if pstatus != 'True': safefield = False
    print('verify-pisprime: %s\n' %pstatus)

    # Verify l is prime.
    pstatus = 'Unverified'
    if not l.is_prime(): pstatus = 'False'
    if l in V: pstatus = 'True'
    if pstatus != 'True': safebase = False
    print('verify-lisprime: %s\n' %pstatus)

    # Write l and p as sums or differences of sparse powers of 2 (if possible).
    print('expand2-p: p = %s\n' % expand2(p))
    print('expand2-l: l = %s\n' % expand2(l))

    # Write the variables in base 16.
    print('hex-p:  %s' %hex(p))
    print('hex-l:  %s' %hex(l))
    print('hex-x0: %s' %hex(x0))
    print('hex-x1: %s' %hex(x1))
    print('hex-y0: %s' %hex(y0))
    print('hex-y1: %s\n' %hex(y1))

    # Verify gcd(l,p) = 1. (Else, if l=p -> DL easy to solve via additive transfers.)
    gcdlpis1 = gcd(l,p) == 1
    print('verify-gcdlp1: %s\n' %gcdlpis1)
```

```
# Verify if embedding degree is large. (Else, multiplicative transfers (or MOV attacks) are easy.)
    # The embedding degree is the smallest integer k such that l divides (p^k-1).
        # It could also be computed (it takes longer): k = (Integers(l)(p)).multiplicative_order()
    # Brainpool and SafeCurves require embedding degree > (l-1)/100.
        # Actually, [Balasubramin, Koblitz] showed MOV is subexponential if k < (log(p))^2.
print('verify-movsafe: Unverified')
print('verify-embeddingdegree: Unverified')
if gcdlpis1 and l.is_prime():
    u = Integers(l)(p)
    d = l-1
    for v in V:
        while d % v == 0: d /= v
    if d == 1:
        d = l-1
        for v in V:
            while d % v == 0:
                if u^(d/v) != 1: break
                d /= v
        print('verify-movsafe: %s' %((l-1)/d <= 100) )
        print('verify-embeddingdegree: %s = (l-1)/%s\n' % (d,(l-1)/d))

# Compute the Frobenius trace t. It should satisfy |E(F_p)|=p+1-t.
    # Hasse's theorem: |t|<2*sqrt(p).
# Also compute the cofactor such that |E(F_p)| = cofactor * l.
    # If E is Montgomery curve, the cofactor has to be a multiple of 4.
t = p+1-l*round((p+1)/l)
if l^2 > 16*p:
    print('verify-trace: %s' % t)
    f = factor(1)
    d = (p+1-t)/l
    for v in V:
        while d % v == 0:
            d //= v
       f *= factor(v)
    print('verify-cofactor: %s\n' % f)
else:
    print('verify-trace: Unverified')
    print('verify-cofactor: Unverified\n')

# Compute the complex-multiplication field discriminant D:
    # Let s^2 be the largest square dividing t^2-4p. Then (t^2-4p)/s^2 is a squarefree negative integer
.
    # If (t^2-4p)/s^2 mod 4 = 1, then D = (t^2-4p)/s^2.
    # Otherwise, D = 4(t^2-4p)/s^2.
# Verify D is big: SafeCurves requires |D|>2^100.
D = t^2-4*p
for v in V:
    while D % v^2 == 0: D /= v^2
if prod([v for v in V if D % v == 0]) != -D:
    print('verify-disc: Unverified')
    print('verify-discisbig: Unverified')
    safedisc = False
else:
    f = -prod([factor(v) for v in V if D % v == 0])
    if D % 4 != 1:
        D *= 4
        f = factor(4) * f
    Dbits = (log(-D)/log(2)).numerical_approx()
    print('verify-disc: %s = %s; -2^%.1f' % (D,f,Dbits))
    print('verify-discisbig: %s\n' %(D < -2^100))

# Verify that the cost of the Pollard's rho attack is above 2^100.
pi4 = 0.78539816339744830961566084581987572105
rho = log(pi4*l)/log(4)
print('verify-rho: %.1f' % rho)
print('verify-rhoabove100: %s\n' %(rho.numerical_approx() >= 100))

# Verify security against twist attacks.
twistl = 'Unverified'
d = p+1+t
for v in V:
    while d % v == 0: d /= v
if d ==1:
    d = p+1+t
    for v in V:
        if d % v == 0:
            if twistl == 'Unverified' or v > twistl: twistl = v

print('verify-twistl: %s\n' % twistl)
print('verify-twistmovsafe: Unverified')
print('verify-twistembeddingdegree: Unverified\n')
if twistl == 'Unverified':
    print('hex-twistl: Unverified\n')
    print('expand2-twistl: Unverified\n')
    print('verify-twistcofactor: Unverified\n')
    print('verify-gcdtwistlp1: Unverified\n')
    print('verify-twistrho: Unverified\n')
    safetwist = False
else:
    print('hex-twistl: %s\n' %hex(twistl))
    print('expand2-twistl: %s\n' % expand2(twistl))
    f = factor(1)
    d = (p+1+t)/twistl
    for v in V:
        while d % v == 0:
            d //= v
            f *= factor(v)
    print('verify-twistcofactor: %s\n' % f)
    gcdtwistlpis1 = gcd(twistl,p) == 1
    print('verify-gcdtwistlp1: %s\n' %gcdtwistlpis1)

    movsafe = 'Unverified'
    embeddingdegree = 'Unverified'
    if gcdtwistlpis1 and twistl.is_prime():
        u = Integers(twistl)(p)
        d = twistl-1
        for v in V:
            while d % v == 0: d /= v
```

```
        if d == 1:
            d = twistl-1
            for v in V:
                while d % v == 0:
                    if u^(d/v) != 1: break
                    d /= v
            print('verify-twistmovsafe: %s' %((twistl-1)/d <= 100))
            print('verify-twistembeddingdegree: %s = (l^-1)/%s\n' % (d,(twistl-1)/d))

    rho = log(pi4*twistl)/log(4)
    print('verify-twistrho %.1f' % rho)
    print('verify-twistrhoabove100: %s\n' %(rho.numerical_approx() >= 100))

    precomp = 0
    joint = 1
    for v in V:
        d1 = p+1-t
        d2 = p+1+t
        while d1 % v == 0 or d2 % v == 0:
            if d1 % v == 0: d1 //= v
            if d2 % v == 0: d2 //= v
            # best case for attack: cyclic; each power is usable
                # also assume that kangaroo is as efficient as rho
            if v + sqrt(pi4*joint/v) < sqrt(pi4*joint):
                precomp += v
                joint /= v

    rho = log(precomp + sqrt(pi4 * joint))/log(2)
    print('verify-jointrho: %.1f' % rho)
    print('verify-jointrhoabove100: %s\n' %(rho.numerical_approx() >= 100))

x0 = k(x0)
y0 = k(y0)
x1 = k(x1)
y1 = k(y1)

# Verify if the equation defines and elliptic curve.
# Verify the shape of the elliptic curve.
# Verify both points [x0,y0] and [x1,y1] are on the curve.
if shape in ('edwards', 'tedwards'):
        d = Integer(readfile(curve+'/d'))
        a = 1
        if shape == 'tedwards':
            a = Integer(readfile(curve+'/a'))

        print('verify-shape: Twisted Edwards')
        print('verify-equation: %sx^2+y^2 = 1%+dx^2y^2\n' % (a, d))
        if a == 1:
            print('verify-shape: Edwards')
            print('verify-equation: x^2+y^2 = 1%+dx^2y^2\n' % d)

        a = k(a)
        d = k(d)
        elliptic = a*d*(a-d)
        level0 = a*x0^2+y0^2-1-d*x0^2*y0^2
        level1 = a*x1^2+y1^2-1-d*x1^2*y1^2

if shape == 'montgomery':
    print('verify-shape: Montgomery')
    A = Integer(readfile(curve+'/A'))
    B = Integer(readfile(curve+'/B'))
    if B == 1: print('verify-equation: y^2 = x^3%s+dx^2+x\n' %A)
    else: print('verify-equation: %sy^2 = x^3%s+dx^2+x\n' %(B,A))

    A = k(A)
    B = k(B)
    elliptic = B*(A^2-4)
    level0 = B*y0^2-x0^3-A*x0^2-x0
    level1 = B*y1^2-x1^3-A*x1^2-x1

if shape == 'shortw':
    print('verify-shape: short Weierstrass')
    a = Integer(readfile(curve+'/a'))
    b = Integer(readfile(curve+'/b'))
    print('verify-equation: y^2 = x^3%s+dx%s+d\n' % (a,b))

    a = k(a)
    b = k(b)
    elliptic = 4*a^3+27*b^2
    level0 = y0^2-x0^3-a*x0-b
    level1 = y1^2-x1^3-a*x1-b

print('verify-elliptic: %s' %str(elliptic)) # discriminant of the eq.
print('verify-iselliptic: %s' %(elliptic != 0)) # if the eq. defines an elliptic curve.
print('verify-isoncurve0: %s' %(level0 == 0)) # if generating point is on the curve.
print('verify-isoncurve1: %s\n' %(level1 == 0)) # if base point is on the curve.

# Transform an Edwards or a twised Edwards curve to a Montgomery curve.
if shape in ('edwards', 'tedwards'):
    A = 2*(a+d)/(a-d)
    B = 4/(a-d)
    x0,y0 = (1+y0)/(1-y0),((1+y0)/(1-y0))/x0
    x1,y1 = (1+y1)/(1-y1),((1+y1)/(1-y1))/x1
    shape = 'montgomery'

# Transform a Montgomery curve to a short Weierstrass.
if shape == 'montgomery':
    a = (3-A^2)/(3*B^2)
    b = (2*A^3-9*A)/(27*B^3)
    x0,y0 = (x0+A/3)/B,y0/B
    x1,y1 = (x1+A/3)/B,y1/B
    shape = 'shortw'

try:
    E = EllipticCurve([a,b])
    numorder2 = 0
    numorder4 = 0
    for P in E(0).division_points(4):
```

```
            if P != 0 and 2*P == 0:
                numorder2 += 1
            if 2*P != 0 and 4*P == 0:
                numorder4 += 1
        print('verify-numorder2: %s' %str(numorder2))
        print('verify-numorder4: %s\n' %str(numorder4))

        # Verify completeness.
        completesingle = False
        completemulti = False
        if numorder4 == 2 and numorder2 == 1:
            # Complete Edwards form, and Montgomery with unique point of order 2.
            completesingle = True
            completemulti = True
        # Should extend this to allow complete twisted hessian.
        print('verify-completesingle: %s' %completesingle)
        print('verify-completemulti: %s\n' %completemulti)

        print('verify-ltimesbase1is0: %s' %(l * E([x1,y1]) == 0))
        print('verify-ltimesbase1: %s\n' %(str(l * E([x1,y1]))) )

        print("verify-cofactorbase01: it can not be done as I do not have z0.")
        print('verify-cofactorbase01: %s\n' %(str(((p+1-t)//l) * E([x0,y0]) == E([x1,y1]))))
    except:
        print('verify-numorder2: Unverified')
        print('verify-numorder4: Unverified\n')

        print('verify-ltimesbase1: Unverified')
        print('verify-cofactorbase01: Unverified\n')
        safecomplete = False

    # Verify monladder.
    montladder = False
    for r,e in (z^3+a*z+b).roots():
        if (3*r^2+a).is_square():
            montladder = True
        print('verify-montladder: %s' %montladder)

    # Verify indistinguishability.
    indistinguishability = False
    elligator2 = False
    if (p+1-t) % 2 == 0:
        if b != 0:
            indistinguishability = True
            elligator2 = True
    print('verify-indistinguishability: %s' %indistinguishability)
    print('verify-ind-notes: Elligator 2: %s\n' % (['No','Yes'][elligator2]))

    # Verify rigidity (by reading the file "rigid").
    saferigid &= (rigid == 'fully rigid' or rigid == 'somewhat rigid')

    safecurve = True
    print('verify-safefield: %s' %safefield)
    print('verify-safeeq: %s' %safeeq)
    print('verify-safebase: %s' %safebase)
    print('verify-saferho: %s' %saferho)
    print('verify-safetransfer: %s' %safetransfer)
    print('verify-safedisc: %s' %safedisc)
    print('verify-saferigid: %s' %saferigid)
    print('verify-safeladder: %s' %safeladder)
    print('verify-safetwist: %s' %safetwist)
    print('verify-safecomplete: %s' %safecomplete)
    print('verify-safeind: %s' %safeind)
```

Listing A1. Algorithm written in SAGE that verifies if a given twisted Edwards curve passes the checks described in Section 5.

References

1. Koblitz, N. Elliptic Curve Cryptosystems. *Math. Comput.* **1987**, *48*, 203–209. [CrossRef]
2. Miller, V.S. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology—CRYPTO '85 Proceedings, Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 18–22 August 1985*; Williams, H.C., Ed.; Springer: Berlin/Heidelberg, Germany, 1986; pp. 417–426. [CrossRef]
3. Hankerson, D.; Menezes, A.J.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer: Berlin, Heidelberg, Germany, 2003.
4. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2009. Available online: http://www.bitcoin.org/bitcoin.pdf (accessed on 30 October 2021).
5. Noether, S.; Mackenzie, A.; Lab, T. Ring Confidential Transactions. *Ledger* **2016**, *1*, 1–18. [CrossRef]
6. Ben-Sasson, E.; Chiesa, A.; Garman, C.; Green, M.; Miers, I.; Tromer, E.; Virza, M. Zerocash: Decentralized Anonymous Payments from Bitcoin. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; IEEE Computer Society: Los Alamitos, CA, USA, 2014, pp. 459–474. [CrossRef]
7. Goldwasser, S.; Micali, S.; Rackoff, C. The knowledge complexity of interactive proof systems. *SIAM J. Comput.* **1989**, *18*, 186–208. [CrossRef]
8. Parno, B.; Howell, J.; Gentry, C.; Raykova, M. Pinocchio: Nearly Practical Verifiable Computation. *Commun. ACM* **2016**, *59*, 103–112. [CrossRef]
9. Groth, J. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology—EUROCRYPT 2016*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 305–326. [CrossRef]

10. Gabizon, A.; Williamson, Z.J.; Ciobotaru, O. PLONK: Permutations over Lagrange-Bases for Oecumenical Noninteractive Arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. 2019. Available online: https://ia.cr/2019/953 (accessed on 30 October 2021).
11. WhiteHat, B.; Bellés, M.; Baylina, J. Baby Jubjub Elliptic Curve. Ethereum Improvement Proposal, EIP-2494. 29 January 2020. Available online: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2494.md (accessed on 30 October 2021).
12. Electric Coin Company. What Is Jubjub? Available online: https://z.cash/technology/jubjub/ (accessed on 30 October 2021).
13. Libert, B.; Mouhartem, F.; Stehlé, D. Notes from the Master Course Cryptology and Security at the École Normale Supérieure de Lyon. Tutorial 8, 2016–2017. Available online: https://fmouhart.epheme.re/Crypto-1617/TD08.pdf (accessed on 30 October 2021).
14. Josefsson, S.; Liusvaara, I. Edwards-Curve Digital Signature Algorithm (EdDSA). Internet Research Task Force (IRTF). Request for Comments: 8032, January 2017. Available online: https://tools.ietf.org/html/8032 (accessed on 30 October 2021),
15. Bernstein, D.J.; Birkner, P.; Joye, M.; Lange, T.; Peters, C. Twisted Edwards Curves. In *Progress in Cryptology—AFRICACRYPT 2008*; Vaudenay, S., Ed.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 389–405. [CrossRef]
16. Montgomery, P.L. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Math. Comput.* **1987**, *48*, 243–243. [CrossRef]
17. Blake, I.; Seroussi, G.; Smart, N. *Elliptic Curves in Cryptography*; London Mathematical Society Lecture Note Series; Cambridge University Press: Cambridge, UK, 1999; Volume 256.
18. Bernstein, D.J.; Lange, T. SafeCurves: Choosing Safe Curves for Elliptic-Curve Cryptography. Available online: https://safecurves.cr.yp.to (accessed on 30 October 2021).
19. European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the EUROPEAN PARLIAMENT and of the Council of 27 April 2016 on the Protection of Natural Persons with Regard to the Processing of Personal Data and on the Free Movement of Such Data, and Repealing Directive 95/46/EC (General Data Protection Regulation). 2016. Available online: https://eur-lex.europa.eu/eli/reg/2016/679 (accessed on 30 October 2021).
20. The California Consumer Privacy Act of 2018. An Act to Add Title 1.81.5 (Commencing with Section 1798.100) to Part 4 of Division 3 of the Civil Code, Relating to privacy, 2018. Available online: https://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.5 (accessed on 30 October 2021).
21. Bellés-Muñoz, M.; Baylina, J.; Daza, V.; Muñoz, J.L. New Privacy Practices for Blockchain Software. *IEEE Softw.* **2021**, [CrossRef]
22. Salleras, X.; Daza, V. ZPiE: Zero-Knowledge Proofs in Embedded Systems. *Mathematics* **2021**, *9*, 2569. [CrossRef]
23. Sestrem Ochôa, I.; Reis Quietinho Leithardt, V.; Calbusch, L.; De Paz Santana, J.F.; Delcio Parreira, W.; Oriel Seman, L.; Albenes Zeferino, C. Performance and Security Evaluation on a Blockchain Architecture for License Plate Recognition Systems. *Appl. Sci.* **2021**, *11*, 1255. [CrossRef]
24. Barreto, P.S.L.M.; Naehrig, M. Pairing-Friendly Elliptic Curves of Prime Order. In Proceedings of the 12th International Conference on Selected Areas in Cryptography (SAC'05), Kingston, ON, Canada, 11–12 August 2005; Springer: Berlin/Heidelberg, Germany, 2005; p. 319–331. [CrossRef]
25. Barreto, P.S.L.M.; Lynn, B.; Scott, M. Constructing Elliptic Curves with Prescribed Embedding Degrees. In *Security in Communication Networks*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 257–267. [CrossRef]
26. Kasamatsu, K.; Kanno, S.; Kobayashi, T.; Kawahara, Y. Barreto-Naehrig Curves. Network Working Group. Internet-Draft. February 2014. Available online: https://tools.ietf.org/html/draft-kasamatsu-bncurves-01 (accessed on 30 October 2021).
27. Yonezawa, S.; Chikara, S.; Kobayashi, T.; Saito, T. Pairing-Friendly Curves. Network Working Group. Internet-Draft. January 2019. Available online: https://tools.ietf.org/html/draft-yonezawa-pairing-friendly-curves-00 (accessed on 30 October 2021).
28. Masson, S.; Sanso, A.; Zhang, Z. Bandersnatch: A Fast Elliptic Curve Built over the BLS12-381 Scalar Field. Cryptology ePrint Archive, Report 2021/1152. 2021. Available online: https://ia.cr/2021/1152 (accessed on 30 October 2021).
29. Ben-Sasson, E.; Chiesa, A.; Tromer, E.; Virza, M. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *Advances in Cryptology—CRYPTO 2014*; Garay, J.A.; Gennaro, R., Eds.; Springer: Berlin/Heidelberg, Germay, 2014; pp. 276–294. [CrossRef]
30. Bowe, S.; Chiesa, A.; Green, M.; Miers, I.; Mishra, P.; Wu, H. ZEXE: Enabling Decentralized Private Computation. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; IEEE Computer Society: Los Alamitos, CA, USA, 2020; pp. 947–964. [CrossRef]
31. Housni, Y.E.; Guillevic, A. Optimized and Secure Pairing-Friendly Elliptic Curves Suitable for One Layer Proof Composition. Cryptology ePrint Archive, Report 2020/351. 2020. Available online: https://ia.cr/2020/351 (accessed on 30 October 2021).
32. Miyaji, A.; Nakabayashi, M.; Nonmembers, S. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* **2001**, *84*, 1234–1243.
33. Koblitz, N. *A Course in Number Theory and Cryptography*, 2nd ed.; Graduate Texts in Mathematics; Springer: Berlin/Heidelberg, Germany, 1994.
34. Silverman, J.H. *The Arithmetic of Elliptic Curves. Graduate Texts in Mathematics*; Springer: New York, NY, USA, 1994; Volume 106.
35. Washington, L.C. *Elliptic Curves. Number theory and Cryptography*, 2nd ed.; Chapman & Hall/CRC Press: Boca Raton, FL, USA, 2008.
36. Okeya, K.; Kurumatani, H.; Sakurai, K. Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications. In Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography (PKC '00), Melbourne, VIC, Australia, 18–20 January 2000; Springer: London, UK, 2000; pp. 238–257. [CrossRef]

37. Bernstein, D.J.; Lange, T. Faster Addition and Doubling on Elliptic Curves. In *Advances in Cryptology—ASIACRYPT 2007*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 29–50. [CrossRef]
38. Hisil, H.; Wong, K.K.H.; Carter, G.; Dawson, E. Twisted Edwards Curves Revisited. In *Advances in Cryptology—ASIACRYPT 2008*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 326–343. [CrossRef]
39. Langley, A.; Hamburg, M.; Turner, S. Elliptic Curves for Security. Internet Research Task Force (IRTF). Request for Comments: 7748. January 2016. Available online: https://tools.ietf.org/html/7748 (accessed on 30 October 2021),
40. Bowe, S. Derivation of Jubjub Elliptic Curve. GitHub. 2019. Available online: https://github.com/zkcrypto/jubjub/blob/master/doc/derive/derive.sage (accessed on 30 October 2021).
41. Bernstein, D.J.; Lange, T. Montgomery curves and the Montgomery Ladder. Cryptology ePrint Archive, Report 2017/293. 2017. Available online: https://ia.cr/2017/293 (accessed on 30 October 2021).
42. Hopwood, D. Supporting Evidence for Security of the Jubjub Curve to Be Used in Zcash. Available online: https://github.com/daira/jubjub/blob/master/verify.sage (accessed on 30 October 2021).
43. Bernstein, D.J.; Hamburg, M.; Krasnova, A.; Lange, T. Elligator: Elliptic-Curve Points Indistinguishable from Uniform Random Strings. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13), Berlin Germany, 4–8 November 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 967–980. [CrossRef]
44. WhiteHat, B. Baby Jubjub Supporting Evidence. GitHub. 2018. Available online: https://github.com/barryWhiteHat/baby_jubjub (accessed on 30 October 2021).
45. Singh, S.K.; Tanwar, S. Analysis of Software Testing Techniques: Theory to Practical Approach. *Indian J. Sci. Technol.* **2016**, *9*, 1–6. [CrossRef]