Community Standard Proposal

Generation of twisted Edwards elliptic curves for circuit use

Barry WhiteHat[1], Jordi Baylina[2], and Marta Bellés[2,3]

[1] *Ethereum foundation,* [2] *iden3,* [3] *Universitat Pompeu Fabra*

May 16, 2019

Contents

1 Scope 2

2 Motivation 2

3 Background and Terminology 2

3.1 Montgomery form . . . 2

3.2 Twisted Edwards form . . . 3

4 Generation of twisted Edwards and Montgomery elliptic curves 3

4.1 Choice of Montgomery equation . . . 4

4.2 Choice of generator and base points . . . 4

4.3 Transformation to twisted Edwards . . . 4

4.4 Optimisation of parameters . . . 5

5 Safety criteria 5

6 Implementations 6

7 Example: Baby Jubjub 6

7.1 Arithmetic In Baby Jubjub . . . 6

7.1.1 Addition Of Points . . . 7

7.1.2 Multiplication Of A Point Of E By A Scalar . . . 7

8 Intellectual Property 8

References 8

A Generation of curves 10

B Safety criteria 11

C Generation of Baby Jubjub 19

1 Scope

This proposal defines a deterministic algorithm for generating twisted Edwards elliptic curves defined over a given prime field. It also provides an algorithm for checking the safety of the curve against best known security attacks.

Additionally, we present an example that puts theory into practice: we detail the generation of the twisted Edwards curve Baby Jubjub and explain how the arithmetic of the curve can be efficiently implemented inside a circuit.

2 Motivation

The study of pairing-friendly elliptic curves and the efficient implementation of cryptographic functions inside circuits has reached a lot of popularity since the appearance of zero-knowledge protocols such as zkSNARKs [15, 7]. In blockchain community these protocols have become a crucial ingredient, not only in privacy aspects but also as a solution to scalability problems.

Inside a SNARK circuit one deals with elements of a certain finite field $\mathbb{F}_p$ where p is determined by the order of certain pairing-friendly elliptic curve used to generate the zero-knowledge proof. If one wants to implement functions involving elliptic curves inside a SNARK, such as the Pedersen hash function [10, Sec. 5.4.1.7] or EdDSA [11], a new curve defined over $\mathbb{F}_p$ is needed.

Choosing this new curve in twisted Edwards [2] or Montgomery form [14] seems the optimal choice for circuit use, as addition in the first case has a single complete formula and in the second, operating is very efficient. As we will see later, twisted Edwards and Montgomery curves are birationally equivalent, which allows as to easily combine both forms inside a circuit.

It is important that the generation of this new curve is done in a transparent and deterministic way, so that it is clear no external considerations are taken for defining it. This is paramount as it significantly reduces the possibility of a backdoor being present, thus leading to better security. Moreover, it is crucial that the new curve is also safe against best known attacks [6].

In this proposal we cover both aspects. We provide and algorithm that given a prime number p, deterministically generates a Montgomery and a birationally equivalent twisted Edwards curve defined over $\mathbb{F}_p$ and also an algorithm for checking its security. The algorithms have already been implemented and tested: in ZCash, the new curve derived from the order of BLS12-381, is called Jubjub [10], and in Ethereum, the adopted curve is called Baby Jubjub [19].

Although the generation of other types of curves, such as pairing-friendly and prime order curves, is out of scope of this proposal, we hope the procedures of generating them get standardised and available to the community soon.

3 Background and Terminology

In this section we present the main definitions needed to understand the document. We restrict the concepts and results to elliptic curves defined over a prime finite field. For general results about elliptic curves see [17] and [16] (this second book requires high mathematical background). The following table contains the notation used across the document.

Notation	**Description**
p	Prime number greater than 2.
$\mathbb{F}_p$	Finite field with p elements.
E_M	Montgomery elliptic curve defined over $\mathbb{F}_p$.
A, B	Parameters of equation $By^2 = x^3 + Ax^2 + x$.
E	Twisted Edwards elliptic curve defined over $\mathbb{F}_p$.
a, d	Parameters of equation $ax^2 + y^2 = 1 + dx^2y^2$.
n	Order of the curve. Typically, $n = h \times l$.
h	Cofactor.
l	Large prime number dividing n.
$\mathbb{G}^M$	Large prime subgroup of $E^M(\mathbb{F}_p)$. $\mathbb{G}^M = \left\{ P \in E^M(\mathbb{F}_p) \mid lP = O \right\}$.
$G_0^M = (x_0^M, y_0^M)$	Generator point of $E^M(\mathbb{F}_p)$.
$G_1^M = (x_1^M, y_1^M)$	Generator of $\mathbb{G}^M$. Also called base point.
$\mathbb{G}$	Large prime subgroup of $E(\mathbb{F}_p)$. $\mathbb{G} = \left\{ P \in E(\mathbb{F}_p) \mid lP = O \right\}$.
$G_0 = (x_0, y_0)$	Generator point of $E(\mathbb{F}_p)$.
$G_1 = (x_1, y_1)$	Generator of $\mathbb{G}$. Also called base point.

3.1 Montgomery form

In this part we define and describe elliptic curves in Montgomery form. We follow [14] and [2].

Definition 3.1 (Montgomery curve)**.** Let $p \geq 3$ be a prime and $\mathbb{F}_p$ the finite field of order p. For $A \in \mathbb{F}_p \backslash \{-2, 2\}$ 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 in Montgomery curves.

Theorem 3.2. *Let $P_1 = (x_1, y_1) \neq O$ and $P_2 = (x_2, y_2) \neq O$ be two points of the Baby-JubJub elliptic curve E_M in Montgomery form.*

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

$$\begin{aligned} \Lambda &= (y_2 - y_1)/(x_2 - x_1), \\ x_3 &= B\Lambda^2 - A - x_1 - x_2, \\ y_3 &= \Lambda(x_1 - x_3) - y_1. \end{aligned} \tag{1}$$

If $P_1 = P_2$, then $2 \cdot P_1$ is a point $P_3 = (x_3, y_3)$ with coordinates

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

Proof. See [12]. □

Theorem 3.3. *The order of a Montgomery curve is a multiple of 4.*

Proof. See [14, Sec. 10.3.2]. □

3.2 Twisted Edwards form

In this section we describe twisted Edwards curves following the definition and results from [2].

Definition 3.4 (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 next theorem shows, twisted Edwards curves have complete addition formulas. This makes these curves very efficient inside circuits.

Theorem 3.5. *Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points 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 neutral element is the point $O = (0, 1)$ and the inverse of a point (x, y) is $(-x, y)$.

Proof. [4, Sec. 3]. □

As the following theorem states, 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 3.6. *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 form 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

$$(u, v) \to (x, y) = \left(\frac{u}{v}, \frac{u - 1}{u + 1}\right). \tag{3}$$

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 [2, Thm. 3.2]. □

4 Generation of twisted Edwards and Montgomery elliptic curves

We present a method that, given a prime number p, we get a twisted Edwards curve defined over $\mathbb{F}_p$. 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 curve.
- Order of the curve and its decomposition into the product of a cofactor and a large prime.
- Generator and base points.

As the finite field is defined by the input p, no specification of this parameter is required. In the same way, the order of the curve and its decomposition is 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 a generator and a base point.

We have divided the procedure in four steps. We start by deterministically generating a Montgomery elliptic curve E^M over $\mathbb{F}_p$ and then setting the generator and base points. Afterwards, we convert the curve and the points to twisted Edwards using theorem 3.6. Last step consists on rescaling all parameters so that less operations are required when doing arithmetic in the curve [8]. All algorithms are implemented in appendix A.

4.1 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 [13] and the work of Zcash team.

The following algorithm takes a prime p, fixes $B = 1$ and returns the elliptic curve defined over $\mathbb{F}_p$ with smallest coefficient A such that $A - 2$ is a multiple of 4. The assumption $A - 2$ divisible by 4 comes from the fact that as this value is used in many operations, so trying to keep it smaller and divisible by four is a reasonable assumption [13]. As with $A = 1$ and $A = 2$ the equation does not describe a smooth curve, the algorithm starts with $A = 3$.

For primes congruent to 1 mod 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 don't have to worry about checking for points on the twist, because the twist cofactor will be the smaller of the two [13]. For a prime congruent to 3 mod 4, both the curve and twist cofactors can be 4, and this is minimal.

Algorithm 1: Generation of E^M

Input: p
Output: A, B, n, h, l

1. **Fix** $B = 1$.
2. **Start** with $A = 3$.
3. **Check** that $(A - 2) \neq 0 \mod 4$. **Otherwise**, increment A by 1 and check this condition again.
4. **Check** that the equation $y^2 = x^3 + Ax^2 + x$ defines an elliptic curve over $\mathbb{F}_p$. **Otherwise**, increment A by 1 and go back to step 3.
5. **Compute** the group order n.
6. **If** $p = 1 \mod 4$: **Check** that the cofactor is 8 and the cofactor of the twist is 4. **Otherwise**, increment A by 1 and go back to step 3.
7. **If** $p = 3 \mod 4$: **Check** that the cofactor of the curve and its twist is 4. **Otherwise**, increment A by 1 and go back to step 3.
8. **Compute** l.
9. **Return** A, B, n, h and l.

4.2 Choice of generator and base points

To pick a generator G_0 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 $G_1 = 8 \cdot G_0$, which has order l.

Algorithm 2: Generator and Base points of E^M

Input: E^M, n, h
Output: G_0^M, G_1^M

1. **Start** with $u = 1$.
2. **Find** v such that (u, v) is a point of E^M. **Otherwise**, increment u by 1 and repeat the step.
3. **Check** that (u, v) has order n. **Otherwise**, 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.

4.3 Transformation to twisted Edwards

Use birational map from equation (3) to get the coefficients, generator and base points in twisted Edwards form.

Algorithm 3: Convert E^M to E

Input: A, B, $G_0^M = (x_0^M, y_0^M)$, $G_1^M = (x_1^M, y_1^M)$
Output: a, d, G_0, G_1
1 **Compute** $a = (A+2)/B$ and $d = (A-2)/B$
2 **Compute** $x_0 = x_0^M / y_0^M$
3 **Compute** $y_0 = (x_0^M - 1)/(x_0^M + 1)$
4 **Set** $G_0 = (x_0, y_0)$
5 **Compute** $x_1 = x_1^M / y_1^M$
6 **Compute** $y_1 = (x_1^M - 1)/(x_1^M + 1)$
7 **Set** $G_1 = (x_1, y_1)$
8 **Return** a, d, G_0 and G_1.

4.4 Optimisation of parameters

As pointed out in [8, Sec. 3.1], if $-a$ is a square in $\mathbb{F}_p$, it is possible to optimise the number of operations in a twisted Edwards curve by scaling it.

Theorem 4.1. *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 just by applying the map. □

The following algorithm 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.

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

Input: a, d, $G_0 = (x_0, y_0)$, $G_1 = (x_1, y_1)$
Output: f, $a' = a/f^2$, $d' = -d/a$,
$G_0' = (x_0/f, y_0)$, $G_1' = (x_1/f, y_1)$
1 **If** $-a$ is a square in $\mathbb{F}_p$:
2   **Take** $f = \sqrt{-a}$
3   **Set** $a' = -1$ and $d' = -d/a$
4   **Compute** $x_0' = x_0/f$ and $x_1' = x_1/f$
5   **Set** $G_0' = (x_0', y_0)$ and $G_1' = (x_1', y_1)$
6   **Return** f, a', d', G_0' and G_1'
7 **Else** :
8   **Set** $f = 1$
9   **Return** f, a, d, G_0 and G_1

After applying the above algorithm, the map that transforms E^M to E becomes the composition of maps from theorems 3.6 and 4.1.

5 Safety criteria

This section specifies the safety criteria that the elliptic curve should satisfy. The choices of of security parameters are based on the joint work of Bernstein and Lange summarised in [5].

In appendix B we provide an implementation of the algorithm that should be run after finding the elliptic curve as proposed in previous section. The algorithm is based on the the code of Daira Hopewood [9] which is an extension of the original SAGE code from [5] to general twisted Edwards curves.

Curve parameters

Check 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 generator of $\mathbb{G}$.

Elliptic Curve Discrete Logarithm Problem

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* [6, Sec. V.1]: 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* [6, Sec. V.2]: we require the embedding degree to be at least $(l-1)/100$.
- *High discriminant* [6, Sec. IX.3]: we require the complex-multiplication field discriminant D to be larger than 2^{100}.

Elliptic Curve Cryptography

- *Ladders* [12]: check the curve supports the Montgomery ladder.
- *Twists* [5, twist]: check it is secure against the small-subgroup attack, invalid-curve attacks and twisted-attacks.
- *Completeness* [5, complete]: 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* [3]: check availability of maps that turn elliptic-curve points indistinguishable from uniform random strings.

6 Implementations

Implementations for generating curves.

- Barry WhiteHat: `https://github.com/barryWhiteHat/baby_jubjub`
- zCash: `https://github.com/zkcrypto/jubjub/blob/master/doc/derive/derive.sage`

Derivation of curves.

- Jubjub elliptic curve: `https://github.com/barryWhiteHat/baby_jubjub`
- Baby Jubjub elliptic curve: `https://github.com/barryWhiteHat/baby_jubjub`

Algorithm of safety criteria.

- Bernstein and Lange: `https://safecurves.cr.yp.to/verify.html`
- Daira Hopewood: `https://github.com/daira/jubjub/blob/master/verify.sage`
- Barry WhiteHat: `https://github.com/barryWhiteHat/baby_jubjub/blob/master/findCurve.sage`

7 Example: Baby Jubjub

In Ethereum, the pairing-friendly elliptic curve used for generating zkSNARK proofs is the prime order curve BN128. More specifically, the order of the curve is

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617.$$

We were interested in deterministically finding a twisted Edwards elliptic curve defined over $\mathbb{F}_p$. To do so, we implemented the procedure described in section 4 and added the code in appendix C. We named the resulting curve *Baby Jubjub*. We summarise the outputs and description of the parameters in the following table.

Vars	Baby Jubjub Parameters
p	21888242871839275222246405745257275088548364400416034343698204186575808495617
a	-1
d	12181644023421730124874158521699555681764249180949974110617291017600649128846
n	21888242871839275222246405745257275088614511777268538073601725287587578984328
h	8
l	2736030358979909402780800718157159386076813972158567259200215660948447373041
x_0	995203441582195749578291179787384436505546430278305826713579947235728471134
y_0	5472060717959818805561601436314318772137091100104008585924551046643952123905
x_1	5299619240641551281634865583518297030282874472190772894086521144482721001553
y_1	16950150798460657717958625567821834550301663161624707787222815936182638968203

Curve *Baby Jubjub* passed all safety checks described in section 4. The security evidence is shown in [19].

7.1 Arithmetic In Baby Jubjub

We decided to include a section showing how arithmetic on Baby Jubjub curve can be implemented inside a circuit. The ideas can easily be adapted to other curves.

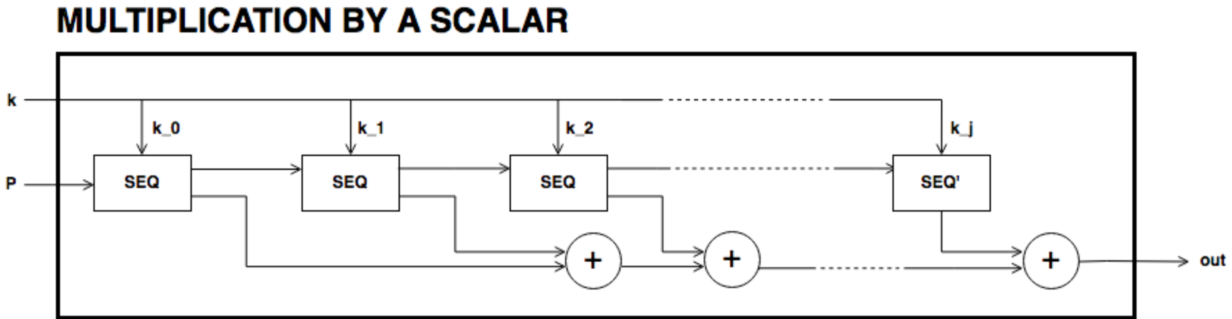
7.1.1 Addition Of Points

When adding points of elliptic curves in Montgomery form, one has to be careful if the points being added are equal (doubling) or not (adding) and if one of the points is the point at infinity [14]. Edwards curves have the advantage that there is no such case distinction and doubling can be performed with exactly the same formula as addition [2]. In comparison, operating in Montgomery curves is cheaper. For the exact number of operations required in different forms of elliptic curves, see [2].

7.1.2 Multiplication Of A Point Of E By A Scalar

Let $P \neq O$ be a point of the Edwards curve E of order strictly greater than 8 (i.e. $P \in \mathbb{G}$) and let k a binary number representing an element of $\mathbb{F}_p$. We describe the circuit used to compute the point $k \cdot P$.

1. First, we divide k into chunks of 248 bits. If k is not a multiple of 248, we take j segments of 248 bits and leave a last chunk with the remaining bits. More precisely, write
$$k = k_0 k_1 \dots k_j$$
with
$$\begin{cases} k_i = b_0^i b_1^i \dots b_{247}^i & \text{for } i = 0, \dots, j-1, \\ k_j = b_0^j b_1^j \dots b_s^j & \text{with } s \leq 247. \end{cases}$$
Then,
$$k \cdot P = k_0 \cdot P + k_1 \cdot 2^{248} P + \dots + k_j \cdot 2^{248j} P. \quad (4)$$
This sum is done using the following circuit. The terms of the sum are calculated separately inside the SEQ boxes and then added together.



2. Each SEQ box takes a point of E of the from $P_i = 2^{248i} P$ for $i = 0, \dots, j-1$ and outputs two points
$$2^{248} \cdot P_i \quad \text{and} \quad \sum_{n=0}^{247} b_n \cdot 2^n \cdot P_i.$$
The first point is the input of the next $(i+1)$-th SEQ box (note that $2^{248} \cdot P_i = P_{i+1}$) whereas the second output is the computation of the i-th term in expression (4). The precise circuit is depicted in next two figures SEQ and WINDOW.

3. The idea of the circuit is to first compute
$$Q = P_i + b_1 \cdot (2P_i) + b_2 \cdot (4P_i) + b_3 \cdot (8P_i) + \dots + b_{247} \cdot (2^{247} P_i),$$
and output the point
$$Q - b_0 \cdot P_i.$$
This permits the computation of Q using the Montgomery form of Baby-Jubjub and only use twisted Edwards for the second calculation. The reason to change forms is that, in the calculation of the output, we may get a sum with input the point at infinity if $b_0 = 0$.

Still, we have to ensure that none of the points being doubled or added when working in E_M is the point at infinity and that we never add the same two points.

   - By assumption, $P \neq O$ and $\mathrm{ord}(P) > 8$. Hence, by Lagrange theorem [1, Corollary 4.12], P must have order r, $2r$, $4r$ or $8r$. For this rea-

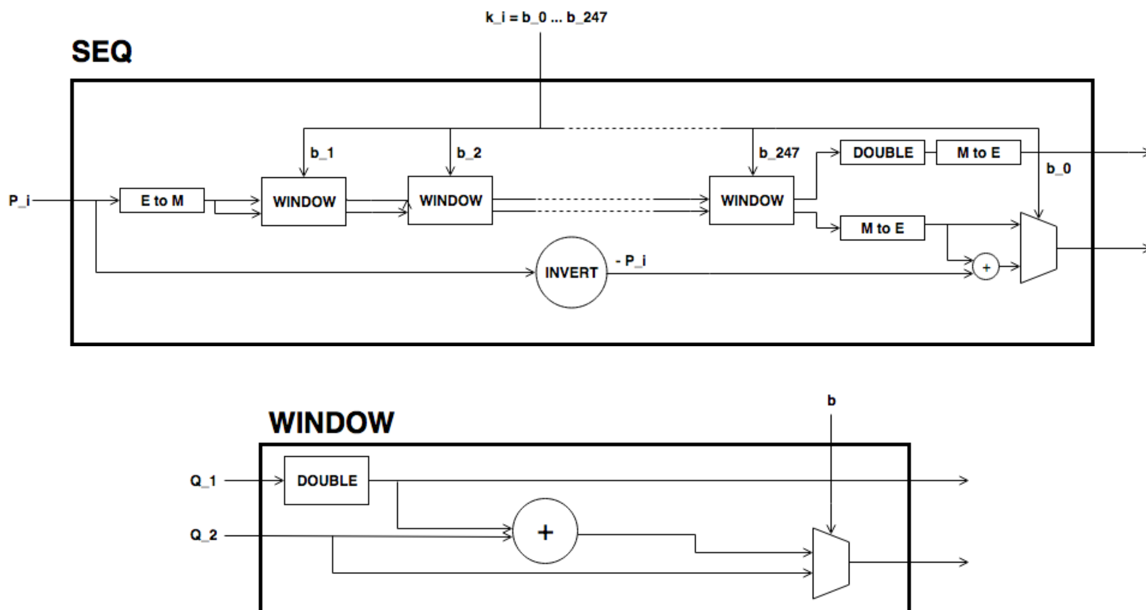
son, none of the points in E_M being doubled or added in the circuit is the point at infinity, because for any integer m, 2^m is never a multiple of r, even when 2^m is larger than r, as r is a prime number. Hence, $2^m \cdot P \neq O$ for any $m \in \mathbb{Z}$.

- Looking closely at the two inputs of the sum, it is easy to realize that they have different parity, one is an even multiple of P_i and the other an odd multiple of P_i, so they must be different points. Hence, the sum in E_M is done correctly.

4. The last term of expression (4) is computed in a very similar manner. The difference is that the number of bits composing k_j may be shorter and that there is no need to compute P_{j+1}, as there is no other SEQ box after this one. So, there is only output, the point $k_j \cdot P_j = k_j \cdot 2^{248j} P$. This circuit is named SEQ'.

The artihmetic on Baby Jubjub has already been implemented. Here are two available codes:

- Barry WhiteHat: https://github.com/barryWhiteHat/baby_jubjub_ecc
- iden3: https://github.com/iden3/circomlib



8 Intellectual Property

To ensure it is freely available to everyone, we release this proposal under Creative Commons.

References

[1] BAUMSLAG, B., AND CHANDLER, B. *Schaum's outline of Theory and Problems of Group Theory*. Schaum's outline series. McGraw-Hill Book Company, New York, 1968. http://poincare.matf.bg.ac.rs/~zarkom/Book_Shaums_Group_theory.pdf.

[2] BERNSTEIN, D. J., BIRKNER, P., JOYE, M., LANGE, T., AND PETERS, C. Twisted edwards curves. Cryptology ePrint Archive, Report 2008/013, March 13, 2008. https://eprint.iacr.org/2008/013.

[3] BERNSTEIN, D. J., HAMBURG, M., KRASNOVA, A., AND LANGE, T. Elligator: Elliptic-curve points indistinguishable from uniform random strings. Cryptology ePrint Archive, Report 2013/325, 2013. https://eprint.iacr.org/2013/325.

[4] BERNSTEIN, D. J., AND LANGE, T. Faster addition and doubling on elliptic curves. *Ad-*

vances in cryptology—ASIACRYPT 2007, 13th international conference on the theory and application of cryptology and information security, Kuching, Malaysia, December 2–6, 2007, proceedings, 4833 (2007), 29–50. http://cr.yp.to/newelliptic/newelliptic-20070906.pdf.

[5] Bernstein, D. J., and Lange, T. Safecurves: choosing safe curves for elliptic-curve cryptography. https://safecurves.cr.yp.to, Accessed February 25, 2019.

[6] Blake, I., Seroussi, G., and Smart, N. *Elliptic Curves in Cryptography*, vol. 256 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 1999.

[7] Groth, J. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Report 2016/260, 2016. https://eprint.iacr.org/2016/260.

[8] Hisil, H., Wong, K. K.-H., Carter, G., and Dawson, E. Twisted edwards curves revisited. Cryptology ePrint Archive, Report 2008/522, 2008. https://eprint.iacr.org/2008/522.

[9] Hopwood, D. Supporting evidence for security of the jubjub curve to be used in zcash. https://github.com/daira/jubjub/blob/master/verify.sage, November 2, 2017.

[10] Hopwood, D., Bowe, S., Hornby, T., and Wilcox, N. Zcash protocol specification version 2019.0.0 [overwinter+sapling]. https://github.com/zcash/zips/blob/master/protocol/protocol.pdf, May 1, 2019.

[11] Josefsson, S., and Liusvaara, I. Edwards-curve digital signature algorithm (eddsa), January, 2017. https://tools.ietf.org/html/8032.

[12] L. Montgomery, P. Montgomery, p.l.: Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation - Math. Comput. 48* (01 1987), 243–243.

[13] Langley, A., Hamburg, M., and Turner, S. Elliptic Curves for Security. RFC 7748, January, 2016. https://rfc-editor.org/rfc/rfc7748.txt.

[14] Okeya, K., Kurumatani, H., and 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* (London, UK, UK, 2000), PKC '00, Springer-Verlag, pp. 238–257.

[15] Parno, B., Howell, J., Gentry, C., and Raykova, M. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2013), IEEE. Best Paper Award.

[16] Silverman, J. H. *The Arithmetic of Elliptic Curves*, vol. 106 of *Graduate Texts in Mathematics*. Springer, New York, 1994.

[17] Washington, L. C. *Elliptic Curves. Number theory and cryptography*, 2 ed. Chapman & Hall/CRC Press, United States of America, 2008.

[18] Weerwag, T. Verifying an elliptic curve cryptographic algorithm using coq and the ssreflect extension, 2016. https://www.ru.nl/publish/pages/813276/weerwag_timmy_-1a.pdf.

[19] WhiteHat, B. Baby-jubjub supporting evidence. GitHub, October 19, 2018. https://github.com/barryWhiteHat/baby_jubjub.

A Generation of curves

```
import sys
import pdb

def findCurve(prime, curveCofactor, twistCofactor, _A):
        F = GF(prime)
        A = _A
        while A < _A + 100000:
                print A
                if (A-2.) % 4 != 0:
                        A+=1.
                        continue
                try:
                        E = EllipticCurve(F, [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 A, E

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

def find3Mod4(prime):
        assert((prime % 4) == 3)
        return findCurve(prime, 4, 4)

def findGenPoint(prime, A, EC, N):
        F = GF(prime)
        for uInt in range(1, 1e3):
                u = F(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 mont_to_ted(u, v , r):
```

```
        x = Mod(u / v, r)
        y = Mod((u-1)/(u+1), r)
        return(x, y)

def ted_to_mont(x, y , r):
        u = Mod((1 + y )/ ( 1 - y)  , r)
        v = Mod((1 + y ) / ( (1 - y) * x) , r )
        return(u,v)

def isOnEd(x,y,r,a,d):
        return Mod(Mod(a,r)*(x**2),r) + Mod(y**2 , r) - 1
        - Mod(d,r)*(Mod(x**2,r))*(Mod(y**2,r)) == 0
```

B Safety criteria

```
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

def writefile(fn,s):
  fd = open(fn,'w')
  fd.write(s)
  fd.close()

def expand2(n):
  s = ""

  while n != 0:
    j = 16
    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 - 10):
      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 requirement(fn,istrue):
  writefile(fn,str(istrue) + '\n')
  return istrue

def verify():
  try:
    os.mkdir('proof')
  except OSError as e:
    if e.errno != EEXIST: raise

  try:
    s = set(map(Integer, readfile('primes').split()))
  except IOError, e:
    if e.errno != ENOENT: raise
    s = set()

  needtofactor = SortedSet()
  V = SortedSet() # distinct verified primes
  verify_primes(V, s, needtofactor)
  verify_pass(V, needtofactor)

  old = V
  needtofactor.update(V)
  while len(needtofactor) > len(old):
    k = len(needtofactor) - len(old)
    sys.stdout.write('Factoring %d integer%s' % (k, '' if k == 1 else 's'))
    sys.stdout.flush()
    for x in needtofactor:
      if x not in old:
        for (y, z) in factor(x):
          s.add(y)
        sys.stdout.write('.')
        sys.stdout.flush()

    print('')

    old = needtofactor.copy()
    verify_primes(V, s, needtofactor)

  writefile('primes', '\n'.join(map(str, s)) + '\n')
  writefile('verify-primes', '<html><body>\n' +
                             ''.join(('2\n' if v == 2 else
                                      '<a href=proof/%s.html>%s</a>\n' % (v,v)) for v in V)
                             '</body></html>\n')

  verify_pass(V, needtofactor)


def verify_primes(V, s, needtofactor):
  for n in sorted(s):
```

```
    if not n.is_prime() or n in V: continue
    needtofactor.add(n-1)
    if n == 2:
      V.add(n)
      continue
    for trybase in primes(2,10000):
      base = Integers(n)(trybase)
      if base^(n-1) != 1: continue
      proof = 'Primality proof for n = %s:\n' % n
      proof += '<p>Take b = %s.\n' % base
      proof += '<p>b^(n-1) mod n = 1.\n'
      f = factor(1)
      for v in reversed(V):
        if f.prod()^2 <= n:
          if n % v == 1:
            u = base^((n-1)/v)-1
            if u.is_unit():
              if v == 2:
                proof += '<p>2 is prime.\n'
              else:
                proof += '<p><a href=%s.html>%s is prime.</a>\n' % (v,v)
              proof += '<br>b^((n-1)/%s)-1 mod n = %s, which is a unit, inverse %s.\n' % (v
              f *= factor(v)^(n-1).valuation(v)
      if f.prod()^2 <= n: continue
      if n % f.prod() != 1: continue
      proof += '<p>(%s) divides n-1.\n' % f
      proof += '<p>(%s)^2 > n.\n' % f
      proof += "<p>n is prime by Pocklington's theorem.\n"
      proof += '\n'
      writefile('proof/%s.html' % n,proof)
      V.add(n)
      break


def verify_pass(V, needtofactor):
  p = Integer(readfile('p'))
  k = GF(p)
  kz.<z> = k[]
  l = Integer(readfile('l'))
  x0 = Integer(readfile('x0'))
  y0 = Integer(readfile('y0'))
  x1 = Integer(readfile('x1'))
  y1 = Integer(readfile('y1'))
  shape = readfile('shape').strip()
  rigid = readfile('rigid').strip()

  safefield = True
  safeeq = True
  safebase = True
  saferho = True
  safetransfer = True
  safedisc = True
  saferigid = True
  safeladder = True
```

```
safetwist = True
safecomplete = True
safeind = True

pstatus = 'Unverified'
if not p.is_prime(): pstatus = 'False'
needtofactor.add(p)
if p in V: pstatus = 'True'
if pstatus != 'True': safefield = False
writefile('verify-pisprime',pstatus + '\n')

pstatus = 'Unverified'
if not l.is_prime(): pstatus = 'False'
needtofactor.add(l)
if l in V: pstatus = 'True'
if pstatus != 'True': safebase = False
writefile('verify-lisprime',pstatus + '\n')

writefile('expand2-p','= %s\n' % expand2(p))
writefile('expand2-l','<br>= %s\n' % expand2(l))

writefile('hex-p',hex(p) + '\n')
writefile('hex-l',hex(l) + '\n')
writefile('hex-x0',hex(x0) + '\n')
writefile('hex-x1',hex(x1) + '\n')
writefile('hex-y0',hex(y0) + '\n')
writefile('hex-y1',hex(y1) + '\n')

gcdlpis1 = gcd(l,p) == 1
safetransfer &= requirement('verify-gcdlp1',gcdlpis1)

writefile('verify-movsafe','Unverified\n')
writefile('verify-embeddingdegree','Unverified\n')
if gcdlpis1 and l.is_prime():
  u = Integers(l)(p)
  d = l-1
  needtofactor.add(d)
  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
    safetransfer &= requirement('verify-movsafe',(l-1)/d <= 100)
    writefile('verify-embeddingdegree','<font size=1>%s</font><br>= (l-1)/%s\n' % (d,(l-1

t = p+1-l*round((p+1)/l)
if l^2 > 16*p:
  writefile('verify-trace','%s\n' % t)
  f = factor(l)
  d = (p+1-t)/l
  needtofactor.add(d)
```

```
  for v in V:
    while d % v == 0:
      d //= v
      f *= factor(v)
  writefile('verify-cofactor','%s\n' % f)
else:
  writefile('verify-trace','Unverified\n')
  writefile('verify-cofactor','Unverified\n')

D = t^2-4*p
needtofactor.add(D)
for v in V:
  while D % v^2 == 0: D /= v^2
if prod([v for v in V if D % v == 0]) != -D:
  writefile('verify-disc','Unverified\n')
  writefile('verify-discisbig','Unverified\n')
  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()
  writefile('verify-disc','<font size=1>%s</font><br>= <font size=1>%s</font><br>&#x2248;
  safedisc &= requirement('verify-discisbig',D < -2^100)

pi4 = 0.785398163397448309615660845819875721O5
rho = log(pi4*l)/log(4)
writefile('verify-rho','%.1f\n' % rho)
saferho &= requirement('verify-rhoabove100',rho.numerical_approx() >= 100)

twistl = 'Unverified'
d = p+1+t
needtofactor.add(d)
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

writefile('verify-twistl','%s\n' % twistl)
writefile('verify-twistembeddingdegree','Unverified\n')
writefile('verify-twistmovsafe','Unverified\n')
if twistl == 'Unverified':
  writefile('hex-twistl','Unverified\n')
  writefile('expand2-twistl','Unverified\n')
  writefile('verify-twistcofactor','Unverified\n')
  writefile('verify-gcdtwistlp1','Unverified\n')
  writefile('verify-twistrho','Unverified\n')
  safetwist = False
else:
  writefile('hex-twistl',hex(twistl) + '\n')
```

```
writefile('expand2-twistl','<br>= %s\n' % expand2(twistl))
f = factor(1)
d = (p+1+t)/twistl
needtofactor.add(d)
for v in V:
  while d % v == 0:
    d //= v
    f *= factor(v)
writefile('verify-twistcofactor','%s\n' % f)
gcdtwistlpis1 = gcd(twistl,p) == 1
safetwist &= requirement('verify-gcdtwistlp1',gcdtwistlpis1)

movsafe = 'Unverified'
embeddingdegree = 'Unverified'
if gcdtwistlpis1 and twistl.is_prime():
  u = Integers(twistl)(p)
  d = twistl-1
  needtofactor.add(d)
  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
    safetwist &= requirement('verify-twistmovsafe',(twistl-1)/d <= 100)
    writefile('verify-twistembeddingdegree',"<font size=1>%s</font><br>= (l'-1)/%s\n" %

rho = log(pi4*twistl)/log(4)
writefile('verify-twistrho','%.1f\n' % rho)
safetwist &= requirement('verify-twistrhoabove100',rho.numerical_approx() >= 100)

precomp = 0
joint = l
needtofactor.add(p+1-t)
needtofactor.add(p+1+t)
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)
writefile('verify-jointrho','%.1f\n' % rho)
safetwist &= requirement('verify-jointrhoabove100',rho.numerical_approx() >= 100)
```

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

if shape in ('edwards', 'tedwards'):
  d = Integer(readfile('d'))
  a = 1
  if shape == 'tedwards':
    a = Integer(readfile('a'))

  writefile('verify-shape','Twisted Edwards\n')
  writefile('verify-equation','%sx^2+y^2 = 1%+dx^2y^2\n' % (a, d))
  if a == 1:
    writefile('verify-shape','Edwards\n')
    writefile('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':
  writefile('verify-shape','Montgomery\n')
  A = Integer(readfile('A'))
  B = Integer(readfile('B'))
  equation = '%sy^2 = x^3<wbr>%+dx^2+x' % (B,A)
  if B == 1:
    equation = 'y^2 = x^3<wbr>%+dx^2+x' % A
  writefile('verify-equation',equation + '\n')

  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':
  writefile('verify-shape','short Weierstrass\n')
  a = Integer(readfile('a'))
  b = Integer(readfile('b'))
  writefile('verify-equation','y^2 = x^3<wbr>%+dx<wbr>%+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

writefile('verify-elliptic',str(elliptic) + '\n')
safeeq &= requirement('verify-iselliptic',elliptic != 0)
safebase &= requirement('verify-isoncurve0',level0 == 0)
safebase &= requirement('verify-isoncurve1',level1 == 0)
```

```
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'

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
  writefile('verify-numorder2',str(numorder2) + '\n')
  writefile('verify-numorder4',str(numorder4) + '\n')
  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
  safecomplete &= requirement('verify-completesingle',completesingle)
  safecomplete &= requirement('verify-completemulti',completemulti)
  safecomplete &= requirement('verify-ltimesbase1is0',l * E([x1,y1]) == 0)
  writefile('verify-ltimesbase1',str(l * E([x1,y1])) + '\n')
  writefile('verify-cofactorbase01',str(((p+1-t)//l) * E([x0,y0]) == E([x1,y1])) + '\n')
except:
  writefile('verify-numorder2','Unverified\n')
  writefile('verify-numorder4','Unverified\n')
  writefile('verify-ltimesbase1','Unverified\n')
  writefile('verify-cofactorbase01','Unverified\n')
  safecomplete = False

montladder = False
for r,e in (z^3+a*z+b).roots():
  if (3*r^2+a).is_square():
    montladder = True
safeladder &= requirement('verify-montladder',montladder)

indistinguishability = False
elligator2 = False
if (p+1-t) % 2 == 0:
```

```
    if b != 0:
      indistinguishability = True
      elligator2 = True
  safeind &= requirement('verify-indistinguishability',indistinguishability)
  writefile('verify-ind-notes','Elligator 2: %s.\n' % ['No','Yes'][elligator2])

  saferigid &= (rigid == 'fully rigid' or rigid == 'somewhat rigid')

  safecurve = True
  safecurve &= requirement('verify-safefield',safefield)
  safecurve &= requirement('verify-safeeq',safeeq)
  safecurve &= requirement('verify-safebase',safebase)
  safecurve &= requirement('verify-saferho',saferho)
  safecurve &= requirement('verify-safetransfer',safetransfer)
  safecurve &= requirement('verify-safedisc',safedisc)
  safecurve &= requirement('verify-saferigid',saferigid)
  safecurve &= requirement('verify-safeladder',safeladder)
  safecurve &= requirement('verify-safetwist',safetwist)
  safecurve &= requirement('verify-safecomplete',safecomplete)
  safecurve &= requirement('verify-safeind',safeind)
  requirement('verify-safecurve',safecurve)

originaldir = os.open('.',os.O_RDONLY)
for i in range(1,len(sys.argv)):
  os.fchdir(originaldir)
  os.chdir(sys.argv[i])
verify()
```

C Generation of Baby Jubjub

```
prime = 21888242871839275222246405745257275088548364400416034343698204186575808495617
Fr = GF(prime)
h = 8 # cofactor

A = int(sys.argv[1])
A, EC = find1Mod4(prime, h, 4, A)

B = 1
a = Fr((A + 2) / B)
d = Fr((A - 2) / B)

print "a " , a , "d " , d , sqrt(d)
# check we have a safe twist
assert(not d.is_square())
assert(a*d*(a-d)!=0)

s = factor(EC.order())
print ("l : " , s)
N = h * s # order of the curve
print (factor(EC.quadratic_twist().order()))

# get generator point
u_gen, v_gen, w_gen = findGenPoint(prime, A, EC, N)
```

```
# find that generator point on the edwards curve
gen_x, gen_y = mont_to_ted(u_gen, v_gen, prime)
# make sure the generator point is on the twisted edwards curve
assert(isOnEd(gen_x, gen_y, prime, a , d))
# go back to montgomery
u , v = ted_to_mont(gen_x, gen_y, prime)
# confirm we are back where we started from
assert (u == u_gen)
assert (v == v_gen)

# get base point on montgomery curve by multiplying the generator point by h
base_x , base_y, base_z = h*EC(u_gen, v_gen)
# find the same points on twisted edwards curve
base_x , base_y = mont_to_ted(base_x , base_y, prime)

# the generator is on the twisted edwards curve
assert(isOnEd(base_x,base_y, prime , a , d))

print ("generator :" , gen_x, gen_y)
print ("base :", base_x, base_y)
```