4-bit Window Pedersen Hash On The Baby Jubjub Elliptic Curve

Jordi Baylina[1] and Marta Bellés[1,2]

[1] *iden3,* [2] *Universitat Pompeu Fabra*

Contents

1 Scope

The 4-bit window Pedersen hash function is a secure hash function which maps a sequence of bits to a compressed point on an elliptic curve [4].

This proposal aims to standardize this hash function for use primarily within the arithmetic circuits of zero knowledge proofs, together with other generic uses such as for Merkle tree or any use cases requiring a secure hash function.

As part of the standard, the paper details the elliptic curve used for the hash function, the process to compute the Pedersen hash from a given sequence of bits, and the computation of the hash from a sequences of bits using an arithmetic circuit—which can be used within zero knowledge proofs.

Moreover the paper includes references to open-source implementations of the Pedersen hash function which follows the computation process details in this proposal.

2 Motivation

The primary advantage of this Pedersen hash function is its efficiency. The ability to compute the hash efficiently makes it an attractive proposal for use within the circuits associated with zk-SNARK proofs [1].

Having a standard, secure, and efficient hash function is one of the paramount aspect for implementing usable, comprehensible, and easily verifiable zero knowledge proofs.

3 Background

The Pedersen hash has already been defined and used by the ZCash team in Sapling, their latest network upgrade [3]. They construct it on the Jubjub elliptic curve and using 3-bit lookup tables. In this document, we propose a different implementation of the Pedersen hash function using Baby-Jubjub elliptic curve and 4-bit windows, which requires less constraints per bit than using 3-bit windows.

4 Terminology

4.1 Elliptic Curve: Baby-Jubjub

Consider the prime number

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

and let $\mathbb{F}_p$ be the finite field with p elements.

We define E_M as the *Baby-Jubjub* Montgomery elliptic curve defined over $\mathbb{F}_p$ given by equation

$$E : v^2 = u^3 + 168698u^2 + u.$$

The order of E_M is $n = 8 \times r$, where

$$r = 2736030358979909402780800718157159386076813972158567259200215660948447373041$$

is a prime number. Denote by $\mathbb{G}$ the subgroup of points of order r, that is,

$$\mathbb{G} = \{ \, P \in E(\mathbb{F}_p) \mid rP = O \, \} .$$

E_M is birationally equivalent to the Edwards elliptic curve

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

where $d = 9706598848417545097372247223557719406784115219466060233080913168975159366771.$

The birational equivalence [2, Thm. 3.2] 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 from E_M to E

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

4.2 Pedersen Hash

Let M be a sequence of bits. The *Pedersen hash* function of M is defined as follows:

- Let $P_0, P_1, \ldots, P_k$ be uniformly sampled generators of $\mathbb{G}$ (for some specified integer k).
- Split M into sequences of at most 200 bits and each of those into chunks of 4 bits[1]. More precisely, write

  $$M = M_0 M_1 \ldots M_l \quad \text{where} \quad M_i = m_0 m_1 \ldots m_{k_i} \quad \text{with} \quad \begin{cases} k_i = 49 & \text{for } i = 0, \ldots, l-1, \\ k_i \leq 49 & \text{for } i = l, \end{cases}$$

  where the m_j terms are chunks of 4 bits $[b_0 \, b_1 \, b_2 \, b_3]$. Define

  $$enc(m_j) = (2b_3 - 1) \cdot (1 + b_0 + 2b_1 + 4b_2)$$

  and let

  $$\langle M_i \rangle = \sum_{j=0}^{k_i - 1} enc(m_j) \cdot 2^{5j}.$$

[1] If M is not a multiple of 4, pad M to a multiple of 4 bits by appending zero bits.

We define the Pedersen hash of M as

$$H(M) = \langle M_0 \rangle \cdot P_0 + \langle M_1 \rangle \cdot P_1 + \langle M_2 \rangle \cdot P_2 + \cdots + \langle M_l \rangle \cdot P_l. \tag{1}$$

Note that the expression above is a linear combination of elements of $\mathbb{G}$, so itself is also an element of $\mathbb{G}$. That is, the resulting Pedersen hash $H(M)$ is a point of the elliptic curve E of order r.

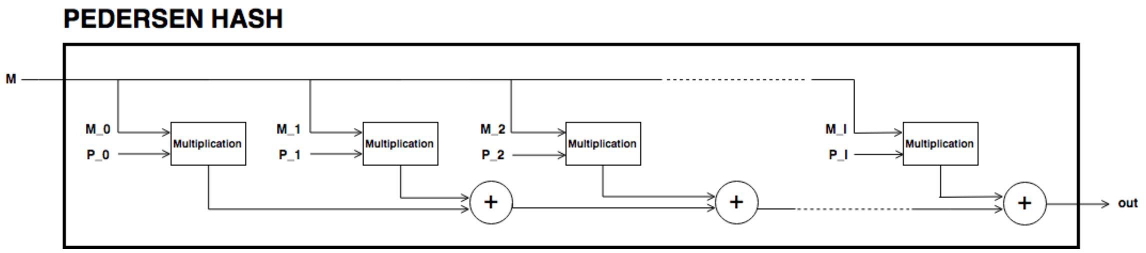
5 Description

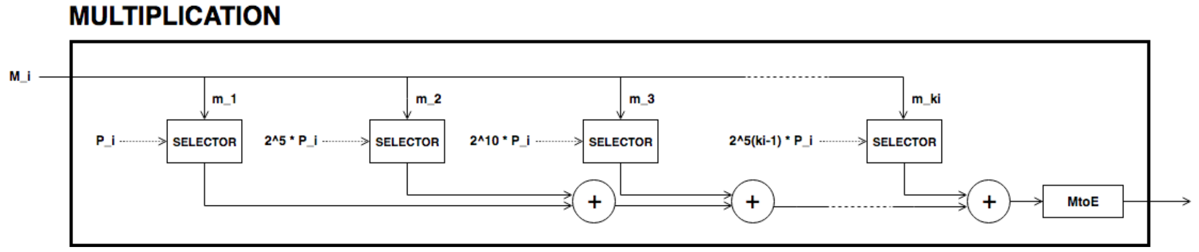
5.1 Set Of Generators

We generate the points $P_0, \ldots, P_k$ in such a manner that it is difficult to find a connection between any of these two points. More precisely, we take `D = "string_seed"` followed by a byte `S` holding that smallest number that `H = Keccak256(D || S)` results in a point in the elliptic curve E.

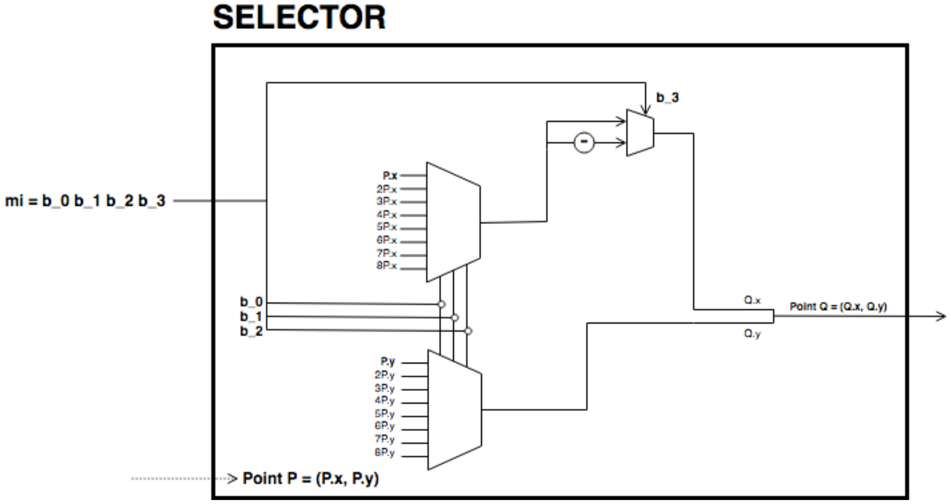
5.2 Computation Of The Pedersen Hash

In the following circuit PEDERSEN HASH, we have depicted the circuit used to compute the Pedersen hash of a message M described in equation 1. Each MULTIPLICATION box returns a term of the sum.





As the set of generators are fixed, we can precompute its multiples and use 4-bit lookup windows to select the right points. This is done as shown in the circuit called SELECTOR. This circuit receives 4-bit chunk input and returns a point. The first three bits are used to select the right multiple of the point and last bit decides the sign of the point. The sign determines if the x-coordinate should be taken positive or negative, as with Edwards curves, negating a point corresponds to the negation of its first coordinate.



5.3 Examples And Test Vectors

Work In Progress

6 Challenges

One of the main challenges to create this standard and to see it adopted by the community is to provide correct, usable, and well-maintained implementations in as many languages as possible.

Some effort is also required to audit and verify code coming from the community and claiming to implement the 4-bit window Pedersen hash function to prevent the propagation of potentially insecure implementations.

Finally, the proposal as it stands today includes the padding of the message M to a multiple of four bits. There are potentials issues with this approach where collisions can happen.

7 Security

7.1 Overflow Prevention

As we described in section 5.2, we use a windowed scalar multiplication algorithm with signed digits. Each 4-bit message chunk corresponds to a window called SELECTOR and each chunk is encoded as an integer from the set $\{-8..8\}\setminus\{0\}$. This allows a more efficient lookup of the window entry for each chunk than if the set $\{1..16\}$ had been used, because a point can be conditionally negated using only a single constraint [3].

As there are up to 50 segments per each generator P_i, the largest multiple of the generator P_i is $n \cdot P_i$ with

$$n = 2^0 \times 8 + 2^5 \times 8 + \left(2^5\right)^2 \times 8 \cdots + 2^{245} \times 8.$$

To avoid overflow, this number should be smaller than $(r-1)/2$. Indeed,

$$\begin{aligned} n &= 8 \times \sum_{k=0}^{49} 2^{5k} = 8 \times \frac{2^{250}-1}{2^5-1} \\ &= 466903585634339497675689455680193176827701551071131306610716064548036813064 \end{aligned}$$

and

$$\begin{aligned} \frac{r-1}{2} &= 1368015179489954701390400359078579693038406986079283629600107830474223686520 \\ &> n. \end{aligned}$$

8 Implementation

8.1 A Note On Efficency: Number Of Constraints Per Bit

When using 3-bit and 4-bit windows, we have **1 constraint for the sign** and **3 for the sum** (as we are using the Montgomery form of the curve, that requires only 3). Now let's look at the constraints required for the multiplexers.

With 3-bit windows we need only one constraint per multiplexer, so **2 constraints** in total.

Standard 4-bit windows require two constraints: one for the output and another to compute $s_0 * s_1$. So, a priori we would need 4 constraints, two per multiplexer. But we can reduce it to 3 as the computation of $s_0 * s_1$ is the same in both multiplexers, so this constraint can be reused. This way only **3 constraints** are required.

So, the amount of constraints per bit are:

- 3-lookup window : $(1+3+2)/3 = 2$ constraints per bit.
- 4-lookup window : $(1+3+3)/4 = 1.75$ constraints per bit.

The specific constraints can be determined as follows: let the multiplexers of coordinates x and y be represented by the following look up tables:

s_2	s_1	s_0	out
0	0	0	a_0
0	0	1	a_1
0	1	0	a_2
0	1	1	a_3
1	0	0	a_4
1	0	1	a_5
1	1	0	a_6
1	1	1	a_7

s_2	s_1	s_0	out
0	0	0	b_0
0	0	1	b_1
0	1	0	b_2
0	1	1	b_3
1	0	0	b_4
1	0	1	b_5
1	1	0	b_6
1	1	1	b_7

We can express them with the following 3 constraints:

- $aux = s_0 s_1$
- $out = [(a_7 - a_6 - a_5 + a_4 - a_3 + a_2 + a_1 - a_0) * aux + (a_6 - a_4 - a_2 + a_0) * s_1 + (a_5 - a_4 - a_1 + a_0) * s_0 + (a_4 - a_0)]z + (a_3 - a_2 - a_1 + a_0) * aux + (a_2 - a_0) * s_1 + (a_1 - a_0) * s_0 + a_0$
- $out = [(b_7 - b_6 - b_5 + b_4 - b_3 + b_2 + b_1 - b_0) * aux + (b_6 - b_4 - b_2 + b_0) * s_1 + (b_5 - b_4 - b_1 + b_0) * s_0 + (b_4 - b_0)]z + (b_3 - b_2 - b_1 + b_0) * aux + (b_2 - b_0) * s_1 + (b_1 - b_0) * s_0 + b_0$

8.2 Existing Implementations

Implementation of the specifications and arithmetic of the Baby-Jubjub curve:

- Barry WhiteHat (SAGE): `https://github.com/barryWhiteHat/baby_jubjub`.
- Jordi Baylina (circom language): `https://github.com/iden3/circomlib/blob/master/circuits/babyjub.circom`.

Implementation of the Pedersen Hash function:

- Jordi Baylina (circom language): `https://github.com/iden3/circomlib/blob/master/circuits/`.

9 Intellectual Property

The source code of the implementations listed in this proposal are publicly available. Circom is licensed under GPL3.

References

[1] Zcash open discussion: choose improved hash function for merkle tree (or replace merkle tree), Accessed February 25, 2018. `https://github.com/zcash/zcash/issues/2258`.

[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] Hopwood, D., Bowe, S., Hornby, T., and Wilcox, N. Zcash protocol specification version 2018.0-beta-31. `https://github.com/zcash/zips/blob/master/protocol/sapling.pdf`, November 14, 2018.

[4] Libert, B., Mouhartem, F., and Stehlé, D. Tutorial 8, 1016-17. Notes from the Master Course Cryptology and Security at the École normale supérieure de Lyon, `https://fmouhart.epheme.re/Crypto-1617/TD08.pdf`.