EFFICIENT CRYPTOGRAPHIC OPERATIONS IN PROOF CIRCUITS (ZK-SNARK) USING ELLIPTIC CURVE EMBEDDING

Dewi Davies-Batista, d.j.davies-batista@student.rug.nl
Supervisors: M. Alghazwi & Assistant Prof. Dr F. Turkmen

Abstract: Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) [1] support the construction of any arithmetic circuit and produce constant-cost proof verification. Due to their low verification costs, zk-SNARKs are frequently used to scale blockchains such as Ethereum. Computation can be done off-chain and a proof is submitted to the smart contracts. The use-case we are considering in this report is calculating the mean of homomorphically encrypted data (private genome data containing SNP counts) inside the proof circuit, thus generating a proof to be verified on the blockchain.

Verification of zk-SNARK proofs involves operations on an elliptic curve E over some base field $\mathbf{F}_p$ (prime p) in which proof circuits operate. Thus, operations in the circuit must be expressed as operations over $\mathbf{F}_p$. The problem is the high cost of doing cryptographic operations over $\mathbf{F}_q$ when q is magnitudes larger in size than $|E(\mathbf{F}_p)|$ as it can lead to high emulation overhead and high proof generation costs. For instance, generating a Groth16 [1] zk-SNARK for Paillier encryption [2] with 2048-bit keys requires over 256 GB.

To address this issue, we leverage the exponential ElGamal encryption scheme based on elliptic curves $E_1/\mathbf{F}_p$ and $E_2/\mathbf{F}_q$ where $q = |E_1(\mathbf{F}_p)|$. This allows us to rely on curve embedding (in a similar way to what is used in ZEXE [3] and Zcash [4]), which reduces the cost of elliptic curve operations inside proof circuits. Specifically, we are considering the exponential ElGamal encryption scheme over the Baby Jubjub curve [5] to perform homomorphic addition, scalar multiplication, encryption, re-encryption and re-randomisation. The goal of this project is to improve the efficiency of performing these operations over the Baby Jubjub curve.

1 Introduction

In this report we introduce improved implementations of homomorphic addition, homomorphic encryption, scalar multiplication of arbitrary on-curve points and re-randomisation, all in the context of the exponential elliptic curve ElGamal encryption scheme, as compared to the implementation given in the `previous_implementation` directory of [6]. These implementations are relevant to the protocol proposed in [7], in which genomic data that researchers would typically be unwilling to share, can be shared securely via the use of homomorphic encryption, zero-knowledge proofs and a blockchain system. Currently, the implementation in use to employ this protocol, which we refer to as 'the previous implementation', makes use of arithmetic circuits written in Circom [8] which can be found at [9]. These circuits were originally written by iden3, the startup company that developed Circom but, at the time of writing, efficiency was not a high priority. As such, it is naturally of interest to try to implement these circuits in a more efficient manner.

We first outline the procedures that the circuits in our implementation help to address. This includes homomorphic addition, encryption, re-encryption and re-randomisation using exponential elliptic curve ElGamal encryption. These circuits are written in Circom, a programming language used to write arithmetic circuits, each producing some number of 'non-linear constraints', a reliable measure for a circuit's efficiency. We write about this in more detail in sub-section 2.6 and give an informal motivation for its use as a metric for efficiency. The primary cost of these circuits is the cost of computing scalar multiples of arbitrary on-curve points. As such, the main improvement made in the new implementation is a more efficient implementation of elliptic curve scalar multiplication of arbitrary on-curve points, further discussed in section 3. We then benchmark the new implementation against the previous implementation comparing the number of non-linear constraints produced by each circuit, their run times and the sizes of the final keys produced. Finally, we consider further developments that could be made to improve the new implementation such as implementing elliptic curve scalar multiplication using lesser known but more efficient methods.

2 Preliminaries

In this section, we describe the procedures relevant to the protocol described in [7]: homomorphic addition, homomorphic encryption, re-encryption and re-randomisation in the context of exponential elliptic curve ElGamal encryption. We aim to write circuits that are as capable but more efficient than the circuits previously used, found in the `previous_implementation` directory of [6]. We conclude this section by discussing non-linear constraints produced by circuits written in Circom and their relevance in measuring the efficiency of a given circuit.

2.1 Homomorphic encryption

Given a plaintext space $\mathcal{M}$ and a ciphertext space $\mathcal{C}$, an encryption scheme $\text{Enc} : \mathcal{M} \to \mathcal{C}$ is homomorphic if

$$\text{Enc}(m_1 + m_2) = \text{Enc}(m_1) + \text{Enc}(m_2)$$

for all $m_1, m_2 \in \mathcal{M}$. This definition of course depends on what is meant by addition in $\mathcal{M}$ and $\mathcal{C}$ but are typically intuitive operations given the structure of each space. An example of this is the exponential elliptic curve ElGamal encryption scheme, further detailed in sub-section 2.2, in which $\mathcal{M} = \mathbf{Z}$ and $\mathcal{C} = E(\mathbf{F}_p) \times E(\mathbf{F}_p)$. In this scheme we take addition in $\mathcal{M}$ to be integer addition and addition in $\mathcal{C}$ to be point-wise elliptic curve addition.

Encryption schemes that offer homomorphic addition allow for the convenient and cost-effective addition of ciphertexts without the need for intermediate instances of decryption.

2.2 Exponential elliptic curve ElGamal encryption

Exponential elliptic curve ElGamal encryption is a homomorphic encryption scheme used in [7] for its homomorphic properties along with its lack of large verification key sizes, a problem experienced when using alternative homomorphic encryption schemes such as Paillier encryption. Given a prime p, an elliptic curve E over the finite field $\mathbf{F}_p$ of p elements, a generator $P \in E(\mathbf{F}_p)$ and uniformly chosen random integers $r, s \in \{1, \ldots, |E(\mathbf{F}_p)| - 1\}$, the ciphertext of a (typically small) plaintext value $m \in \mathbf{Z}$ is given by $\text{Enc}_{r,s}(m) = ([r]P, [m]P + [r]Q)$ where $Q = [s]P$. The randomly generated integer s is a shared secret key between both parties and can be used to decrypt a given ciphertext while r is used purely to obtain random-looking points on the curve for the sake of the intended security of the encryption scheme. The security of $\text{Enc}(m)$ is ensured by the difficulty of computing discrete logarithms in $E(\mathbf{F}_p)$ when $|E(\mathbf{F}_p)|$ is sufficiently large.

As such, homomorphic encryption in the context of this paper invokes one instance of elliptic curve addition, one instance of elliptic curve scalar multiplication of arbitrary on-curve points and two instances of elliptic curve scalar multiplication of a pre-chosen generator.

2.3 Homomorphic addition

As mentioned in section 2.1, in the context of exponential elliptic curve ElGamal encryption: $\mathcal{M} = \mathbf{Z}$, $\mathcal{C} = E(\mathbf{F}_p) \times E(\mathbf{F}_p)$, where p is a large prime, and the addition of ciphertexts $\text{Enc}_{r_1,s}(m_1) = ([r_1]P, [m_1]P + [r_1]Q)$ and $\text{Enc}_{r_2,s}(m_2) = ([r_2]P, [m_2]P + [r_2]Q)$ is point-wise elliptic curve addition, that is

$$\begin{aligned}
&\text{Enc}_{r_1,s}(m_1) + \text{Enc}_{r_2,s}(m_2) \\
&= \big([r_1]P + [r_2]P, [m_1]P + [r_1]Q + [m_2]P + [r_2]Q\big) \\
&= \big([r_1 + r_2]P, [m_1 + m_2]P + [r_1 + r_2]Q\big) \\
&= \text{Enc}_{r_1+r_2,s}(m_1 + m_2)
\end{aligned}$$

where $Q = [s]P$, and is hence homomorphic. Note that this invokes only two instances of elliptic curve addition.

2.4 Re-encryption

Given the ciphertext

$$\text{Enc}_{r,s}(m) = ([r]P, [m]P + [r]Q) =: (C_1, C_2)$$

and its corresponding secret key s, one can recover $[m]P$ by noting that $Q = [s]P$ and computing

$$\begin{aligned}
&C_2 - [s]C_1 \\
&= [m]P + [r]Q - [s]([r]P) \\
&= [m]P + [r]([s]P) - [r \cdot s]P \\
&= [m]P + [r \cdot s]P - [r \cdot s]P \\
&= [m]P.
\end{aligned}$$

After this, one can use baby-step giant-step or Pollard rho's algorithm to compute the discrete logarithm of $[m]P$ and hence retrieve the plaintext m. To re-encrypt a recovered plaintext point $[m]P$, one expectedly computes $([r']P, [m]P + [r']Q)$ where $r' \in \{1, \ldots, |E(\mathbf{F}_p)| - 1\}$ is a uniformly randomly chosen integer, as in homomorphic encryption. Note that this requires two instances of elliptic curve scalar multiplication of a pre-chosen generator, one for computing $[r']P$ and on for computing $[r']Q = [r' \cdot s]P$ as the secret key s is known to the user, and one instance of elliptic curve addition.

This circuit is not benchmarked in section 4 as there was no such circuit developed in the previous

implementation. It's worth noting though that the previous implementation has a more efficient circuit for elliptic curve scalar multiplication of a pre-chosen generator. The reason for this is that both the circuit for scalar multiplication of a pre-chosen generator and for arbitrary on-curve points use the same underlying algorithms as it was initially intended that both would be more efficient. This is discussed briefly in section 5 as an implementation that uses a hybrid of the currently new and previous implementations would be more efficient in the form of fewer non-linear constraints. As such, if one is to use re-encryption, they should consider using the circuit for scalar multiplication of a pre-chosen generator provided in the previous implementation as opposed to the circuit used in the new implementation.

2.5 Re-randomisation

Given the ciphertext

$$\text{Enc}_{r,s}(m) = ([r]P, [m]P + [r]Q) =: (C_1, C_2)$$

and a uniformly randomly chosen integer $r' \in \{1, \ldots, |E(\mathbf{F}_p)| - 1\}$ one can re-randomise the ciphertext by computing

$$\text{Re-Rand}_{r'}(\text{Enc}_{r,s}(m)) = (C_1 + [r']P, C_2 + [r']Q).$$

Note that this invokes one instance of elliptic curve scalar multiplication of the pre-chosen generator, one instance of elliptic curve scalar multiplication of an arbitrary on-curve point and two instances of elliptic curve addition. By re-randomizing the ciphertext, the original ciphertext is transformed, providing an additional layer of security.

2.6 Non-linear constraints in arithmetic circuits

In the context of this report, an important metric for the efficiency of a circuit written in Circom is the number of non-linear constraints it produces. In Circom, a non-linear constraint is produced when a computation involving the product of two input signals is made. For example, if k and l are input signals to a circuit then computing the product $k \cdot l$ in the circuit will add a non-linear constraint. If instead l is a variable with a fixed value defined in the circuit then computing the product $k \cdot l$ would not produce a non-linear constraint.

An informal motive for using the number of non-linear constraints as a metric when assessing the efficiency of a given arithmetic circuit is the following: assuming k and l are inputs that in practice are high-bit integers, the computation of the product $k \cdot l$, which produces a non-linear constraint, is far more computationally expensive than simply computing $a \cdot l$ where a is a relatively small integer defined in the circuit. As such, when assessing the efficiency of a circuit whose inputs are sufficiently large, it suffices to only consider the number of non-linear constraints as the cost of the computations that produce these non-linear constraints massively overshadows the cost of any arithmetic operation involving a preset constant.

3 New methods of scalar multiplication

ElGamal encryption, re-encryption and re-randomisation rely on circuit implementations of elliptic curve scalar multiplication of both arbitrary on-curve points on Baby Jubjub and of a pre-chosen generator. The cost of these two circuits differs heavily, with scalar multiplication of arbitrary on-curve points on Baby Jubjub expectedly costing more. Together, these circuits account for the majority of the cost of executing ElGamal encryption, re-encryption and re-randomisation. As such, a natural approach to reducing the cost of these procedures is to find a cheaper way to perform scalar multiplication, which was the approach taken in developing the new implementation.

In the previous implementation, iden3's circuit implementations of scalar multiplication are used - `escalarmulfix` for scalar multiplication of a pre-chosen generator and `escalarmulany` for scalar multiplication of an arbitrary on-curve point. `escalarmulfix` and `escalarmulany` use the sliding-windowed method, an efficient method of computing scalar multiples of points on an elliptic curve. The cost of these circuits is given in Table 3.3, `escalarmulfix` producing 523 non-linear constraints and `escalarmulany` producing 2301 non-linear constraints. It should be noted that the scalar input to these circuits must be given in its binary representation. As such, a circuit converting the integer by which we scale a point to its binary representation must be used. In iden3's case, this is the `bitify` circuit which produces an additional 253 non-linear constraints.

To improve upon the unusually high number of non-linear constraints produced by `escalarmulany`, the new implementation relies on the double-and-add method of scalar multiplication. Though double-and-add is in theory less efficient, the cost of scalar multiplication of arbitrary on-curve points on Baby Jubjub in the new implementation is heavily reduced, enough to reduce the cost of encryption and re-randomisation, as demonstrated in comparing Table 4.3 against Table 4.4 and Table 4.5 against Table 4.6.

3.1 double-and-add for arbitrary on-curve points

The following is an overview of how double-and-add for arbitrary on-curve points, `doubleAndAddAny` in the `new_implementation` directory of [6], was implemented. Note that this is not an in-depth line-by-line assessment of how the circuit is written but instead summarises what is written as well as the underlying challenges faced while writing in Circom along with how these challenges were resolved. An explicit assessment of the cost of these circuits is given in section 4.

Traditionally, the double-and-add method of elliptic curve scalar multiplication is done in a way similar to the following. Given the scalar k and base point P as input, one computes $[k]P$ as follows:

Algorithm 3.1 Traditional double-and-add for scalar multiplication on an elliptic curve $E/\mathbf{F}_p$.

Input: An integer $k > 0$ and a point $P \in E(\mathbf{F}_p)$.
Output: $R = [k]P \in E(\mathbf{F}_p)$

```
R ← O
while k > 0 do
    bit ← k%2
    if bit = 1 then
        R ← R + P
    P ← P + P
    k ← k >> 1
return R
```

In implementing the double-and-add method in Circom, one must take into account that input signals, i.e. the input scalar k, and its modified state can not be used explicitly as a conditional. As such, implementing double-and-add is not as simple as in the brief pseudocode written above. To get around this, we instead implement something similar to the following.

Algorithm 3.2 Condition-free double-and-add for scalar multiplication on an elliptic curve $E/\mathbf{F}_p$.

Input: An integer $k > 0$ and a point $P \in E(\mathbf{F}_p)$.
Output: $R = [k]P \in E(\mathbf{F}_p)$

```
R ← P
firstBit ← k%2
while k > 0 do
    bit ← k%2
    R ← [bit](R + P) + [1 − bit](R + O)
    P ← P + P
    k ← k >> 1
R ← R − [1 − firstBit]P
return R
```

The current bit of k is given by $k\%2$ and so if it is 0 then the additive identity $\mathcal{O}$ is added to the returned point R, i.e. R remains the same as before adding, and if the current bit is 1 then P is added to R, as in regular double-and-add. In doing this, the input scalar k is not used explicitly as a conditional.

The second consideration to make when implementing this in Circom is that in computing

$$R \leftarrow [bit](R + P) + [1 - bit](R + \mathcal{O}),$$

$+$ denotes Montgomery elliptic curve addition. As such, adding $\mathcal{O}$ to the returned point R requires adding by a point that does not change R. For a curve in Twisted Edwards form, this can be achieved simply by adding the additive identity $(0, 1)$ to R but in Montgomery form, Baby Jubjub does not have an additive identity, which poses the following challenge: given a point $P = (x_1, y_1)$ on an elliptic curve in Montgomery form, what point $Q = (x_2, y_2)$ should be added to P such that $P + Q = P$?

To address this, note the following. Given points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on an elliptic curve in its Montgomery form with coefficients A and B, one computes the sum $P + Q = (x_3, y_3)$ via

Algorithm 3.3 Point addition on a Montgomery elliptic curve $E/\mathbf{F}_p$.

Input: Points $P = (x_1, y_1)$, $Q = (x_2, y_2) \in E(\mathbf{F}_p)$.
Output: $P + Q = (x_3, y_3) \in E(\mathbf{F}_p)$

```
λ ← (y2 − y1)(x2 − x1)^−1
x3 ← Bλ^2 − A − x1 − x2
y3 ← λ(x1 − x3) − y1
return (x3, y3)
```

Taking $P = (-x - A, -y)$ and $Q = (0, -y)$, Algorithm 3.3 yields $(x_3, y_3) = (x, y)$ and so $P + Q = (x, y)$. As such, in implementing double-and-add in Circom, if the current bit of k is 0, we instead set the returned point R to be the sum of the points $(-x - A, -y)$ and $(0, -y)$ where x and y are the coordinates of R before addition. With this in mind, our approach to double-and-add can be summarised by Algorithm 3.4.

Algorithm 3.4 Circom-tailored double-and-add for scalar multiplication on an elliptic curve $E/\mathbf{F}_p$.

Input: An integer $k > 0$ and a point $P \in E(\mathbf{F}_p)$.
Output: $R = [k]P \in E(\mathbf{F}_p)$

```
R ← P
firstBit ← k%2
while k > 0 do
    bit ← k%2
    R ← [bit](R+P)+[1−bit]((−Rx − A, −Ry) +
    (0, −Ry))
    P ← P + P
    k ← k >> 1
R ← R − [1 − firstBit]P
return R
```

Figure 4.1: Times taken to execute homomorphic addition with varying input sizes

3.2 double-and-add for a pre-chosen generator

For scalar multiplication of a pre-chosen generator, `doubleAndAddGenerator`, an almost identical approach is taken. The only change is that the set of doubles of the pre-chosen generator are pre-computed and stored in a separate circuit, `generatorDoubles`. In doing this, one heavily reduces the number of non-linear constraints required by `doubleAndAddGenerator` as no doubling is needed. It's advised that the pre-computation of these doubles is done in a different language, for example Python, as computing these doubles in Circom itself and manually formatting the `generatorDoubles` circuit would be laborious.

4 Results

Here we give an assessment of the efficiency of the circuits used in each implementation in terms of their non-linear constraints and run times. We assess homomorphic addition, encryption and re-randomisation.

In the previous implementation, homomorphic addition, encryption, re-randomisation and re-encryption use a mix of addition and scalar multiplication in both Montgomery and Twisted Edwards form. There is good motive for the original circuits used to be written in such a way but for the purposes of the protocol described in [7], a lot of this is unnecessary. An example of this is the switching from Twisted Edwards form to Montgomery form, and vice versa, via a birational map in computing elliptic curve scalar multiples. In the new implementation, Baby Jubjub remains purely in Montgomery form and no interchanging to Twisted Edwards form is done. The main advantage of this, as we will see in sub-section 4.2, is that the number of non-linear constrains produced in performing homomorphic addition is halved.

4.1 Benchmark methodology

There are three phases to a full proof using the Groth16 scheme used in our benchmarking: proof set up, proof generation and proof verification. Proof verification is constant in time, and so is left out of the result illustrations in this section. The run times of proof set up and proof generation typically scale linearly with respect to the number of non-linear constraints produced by the circuit and act as a good metric of the efficiency of a circuit.

In order to measure the run time of each circuit, we used a typescript program, `benchmarkMultCircuits`, to return the average run time of each circuit with a varying number of inputs each running 10 times. For example, to measure the efficiency of the homomorphic addition circuits, we measured the average run time of the circuit given 10, 100 and 1000 inputs, each 10 times, and then took the average. For homomorphic encryption and re-randomisation, the number of inputs varied from 10 to 100 in intervals of 10.

All benchmarks were made on an M1 MacBook Air with 16GB of RAM.

4.2 Homomorphic addition

In the previous implementation, homomorphic addition is implemented using addition on Baby Jubjub in Twisted Edwards form. The cost of elliptic curve addition in Twisted Edwards form in Circom is 6 non-linear constraints. As stated in section 2.2, homomorphic addition invokes two instances of elliptic curve addition and so the overall cost of homomorphic addition in the previous implementation is 12 non-linear constraints, as displayed in Table 4.1.

In the new implementation however, all elliptic curve operations are performed on Baby Jubjub in its Montgomery form. This is advantageous as elliptic curve addition in Montgomery form induces 3 non-linear constraints and so the cost of homomorphic addition in the new implementation is 6

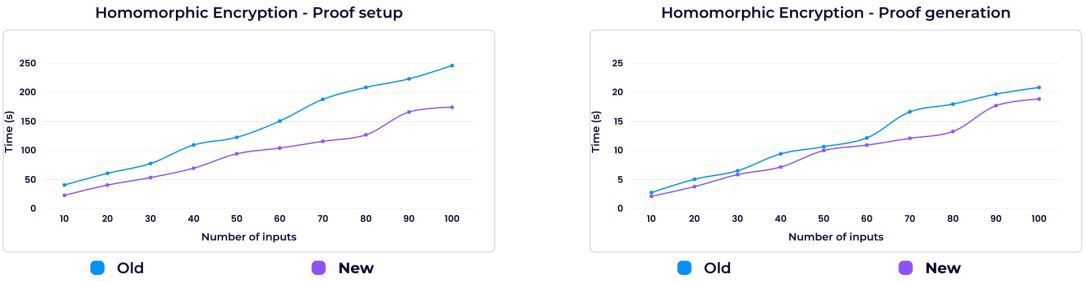

Figure 4.2: Times taken to execute homomorphic encryption with varying input sizes

non-linear constraints.

This reduction in the number of non-linear constraints produced by 50% is perhaps the most significant improvement of the new implementation but is better understood by the time taken to run given 10, 100 and 1000 inputs illustrated in Figure 4.1. Expectedly, we see marginal differences in both the setup and generation phases of the proof at an input size of 10 and begin to see the advantages of the new implementation at the larger input sizes of 100 and 1000. At 100 inputs, there was a reduction in setup time by 16.6% and at 1000 inputs there was a reduction by 12%. As for the proof generation times, there is a 22% reduction at 100 inputs and a 34.7% reduction at 1000 inputs.

On top of these reductions in run time, there is a reduction in the size of the final key files used in the proof verification phase. At 10 inputs there is a reduction by 32.7% in size, 34.9% at 100 inputs and 29% at 1000 inputs.

Hom. Add.	Sub-function(s)	Non-linear constraints
Previous	**Total**	**12**
	`Tw. Edw. Addition` **(x2)**	12
New	**Total**	**6**
	`Mont. Addition` **(x2)**	6

Table 4.1: Number of non-linear constraints produced by homomorphic addition in both implementations

4.3 Homomorphic encryption

For homomorphic encryption, we can attribute the reduced run times to the more efficiently written scalar multiplication circuits, whose outline was given in section 3. In the previous implementation, where iden3's scalar multiplication circuits are used, a conversion from Twisted Edwards form to Montgomery, and vice versa, is made, along with other processes that are unnecessary for the purposes of either implementation. These processes are listed in the Sub-function column of Table 4.2.

Hom. Enc.	Sub-function	Non-linear constraints
Previous	**Total**	**3859**
	`Bitify` **(x2)**	506
	`Scal. Mult. Any`	2301
	`Scal. Mult. Gen.` **(x2)**	1046
	`Tw. Edw. Addition`	6
New	**Total**	**3260**
	`Scal. Mult. Any`	1757
	`Scal. Mult. Gen.` **(x2)**	1500
	`Mont. Addition`	3

Table 4.2: Number of non-linear constraints produced by homomorphic encryption in both implementations

The main reduction in non-linear constraints is the result of the new circuit for the scalar multiplication arbitrary on-curve points, denoted by `Scal. Mult. Any` in Table 4.2. We see that the previous implementation's `Scal. Mult. Any` produced 2301 non-linear constraints while the new implementation's `Scal. Mult. Any` produces 1757 non-linear constrains, a reduction of 544. Additionally, for `Scal. Mult. Any` to be used in the previous implementation, iden3's circuit, `Bitify`, must be invoked, producing an additional 253 non-linear constraints.

We see that homomorphic encryption in the new implementation costs 599 fewer non-linear constraints. In Figure 4.2 we see that the differences are a lot more apparent when the input size increases. The average reduction in run time for the proof setup from 10 to 50 inputs is 33.4% and from 60 to 100 we observe an average reduction of 32.6%. Overall, the average reduction in run time for proof set up phase is 33%. For the proof generation phase, there is not as much of a reduction with a 17.6% average reduction from 10 to 50 inputs and a 16.6% reduction from 60 to 100 inputs. Overall, the average reduction in the run time of the proof generation phase is 17.1%.

To add to this, the final key sizes are not reduced in the new implementation. On average, across input sizes 10 to 100, the size of the final key produced increases by 21%.

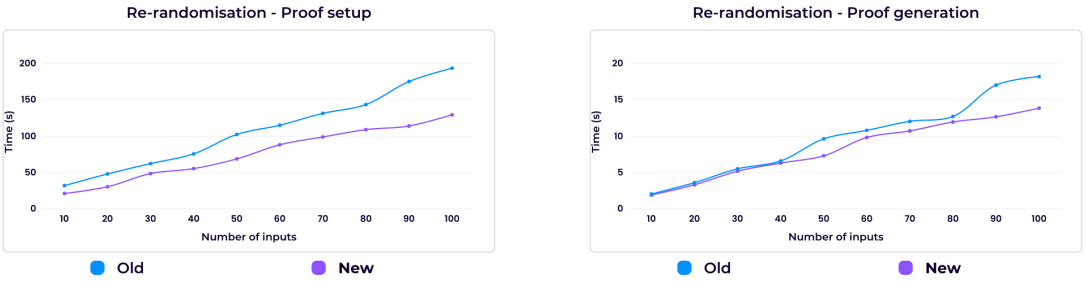

Figure 4.3: Times taken to execute re-randomisation with varying input sizes

4.4 Re-randomisation

Similarly to homomorphic encryption, the core improvement to re-randomisation in the new implementation is due to the more efficient circuit for scalar multiplication of arbitrary on-curve points.

As a result of the heavy reduction in the number of non-linear constraints produced by the new implementation's `Scal.Mult.Any` we have that the number of non-linear constraints produced by re-randomisation in the new implementation is reduced by 576, a similar reduction as in homomorphic encryption.

We illustrate the run times of the re-randomisation circuits in both the previous and new implementations in Figure 4.3. The average reduction in run times for the proof setup phase from 10 to 50 inputs is 30.6% with a 28% average reduction from 60 to 100 inputs. Overall, the average reduction in run time for proof setup is 29.3%. For the proof generation phase, the average reduction from 10 to 50 inputs is 15.2% and is 9.8% from 60 to 100 inputs. Overall, the reduction in run time for the proof generation phase is 12.5%.

Regarding the difference in final key sizes, there is an average increase of 15.4%.

Re-rand.	Sub-function	Non-linear constraints
Previous	**Total**	**3089**
	`Bitify`	253
	`Scal. Mult. Any`	2301
	`Scal. Mult. Gen.`	523
	`Tw. Edw. Addition` (x2)	12
New	**Total**	**2513**
	`Scal. Mult. Any`	1757
	`Scal. Mult. Gen.`	750
	`Mont. Addition` (x2)	6

Table 4.3: Number of non-linear constraints produced by re-randomisation in both implementations

5 Further discussion

Beyond what was considered in this paper, there are a number of interesting avenues to pursue. This includes using a hybrid of both the new and previous implementations due to the fact that the previous implementation has a more efficient circuit for scalar multiplication of a pre-chosen generator. Together with the more efficiently written circuit for scalar multiplication of an arbitrary on-curve point given in the new implementation, one would have an implementation in which the cost of homomorphic encryption is reduced by around 450 non-linear constraints along with a reduction of around 230 in the number of non-linear constraints produced by re-randomisation.

Another, perhaps more natural, development to make would be to implement the sliding-windowed method for elliptic curve scalar multiplication. As noted earlier, this was done in iden3's implementation, but their circuit for scalar multiplication of an arbitrary on-curve point is 'bloated' in the sense that it produces far more non-linear constraints than one would expect. Using the same method but in a more efficient way could prove useful in further reducing the cost of elliptic curve scalar multiplication, reducing the cost of homomorphic encryption and re-randomisation along with it.

Regarding the analysis of the efficiency of these circuits, one could more robustly describe the rate at which this new implementation's circuits reduce their costs. For example, with more benchmarks, one might be able to conclude that the reduction in run times grows linearly, quadratically, exponentially etc. with respect to the number of inputs. This would help to understand how these circuits would perform on larger scales.

Appendix

All programs relevant to this report can be found at [6]. The most relevant programs being `doubleAndAddAny`, `doubleAndAddGenerator`,

`homEnc` and `re-Rand` in the `new-implementation` directory and `escalarmulany`, `escalarmulfix`, `homEnc` and `re-Rand` in the `previous-implementation` directory.

References

[1] Jens Groth. On the size of pairing-based non-interactive arguments. *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326, 2016.

[2] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*, pages 223–238, 1999.

[3] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964, 2020.

[4] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 4:220, 2016.

[5] Barry WhiteHat, Jordi Baylina, and Marta Bellés. Baby jubjub elliptic curve. *Ethereum Improvement Proposal, EIP2494*, 29, 2020.

[6] Dewi. Project repository. `https://gitlab.com/ecc-snark/zkp-circuits/-/tree/dewi`. [Online; accessed 14.03.2023].

[7] Mohammed Alghazwi and Fatih Turkmen. Decentralized private genome analysis using verifiable off-chain computation. *Under review*, 2022.

[8] Circom. Circuit compiler for zk proving systems. `https://github.com/iden3/circom`. [Online; accessed 14.03.2023).

[9] Circom. Circomlib. `https://github.com/iden3/circomlib`. [Online; accessed 14.03.2023).