Practical Implementation of Pairing-Based zkSNARK in Bitcoin Script

Federico Barbacovi[1], Enrique Larraia[1], Paul Germouty[⋆], and Wei Zhang[1]

nChain
`f.barbacovi@nchain.com`, `e.larraia@nchain.com`, `germouty.paul@orange.fr`, `w.zhang@nchain.com`

Abstract. Groth16 is a pairing-based zero-knowledge proof scheme that has a constant proof size and an efficient verification algorithm. Bitcoin Script is a stack-based low-level programming language that is used to lock and unlock bitcoins. In this paper, we present a practical implementation of the Groth16 verifier in Bitcoin Script deployable on the mainnet of a Bitcoin blockchain called BSV. Our result paves the way for a framework of verifiable computation on Bitcoin: a Groth16 proof is generated for the correctness of an off-chain computation and is verified in Bitcoin Script on-chain. This approach not only offers privacy but also scalability. Moreover, this approach enables smart contract capability on Bitcoin which was previously thought rather limited if not non-existent.

Keywords: Bitcoin · Smart Contract · Zero-Knowledge Proof.

1 Introduction

Zero-knowledge proofs (ZKPs) have been widely adopted to enhance blockchain technology. For example, zCash [36] and Firo [16] use ZKPs for user privacy, and Ethereum uses ZKPs for scalability [14]. Bitcoin, on the other hand, is thought to have limitations that make ZKP integration much more difficult, one of which is the Bitcoin scripting language. Due to its stack-based structure and primitive set of opcodes, it is rather difficult to implement the complex mathematical functions required by most ZKPs.

Despite these limitations, there are many projects trying to integrate ZKPs and Bitcoin. They determined to make Bitcoin more scalable and more capable for smart contracts, ultimately making Bitcoin more economically sustainable and viable even with the diminishing block reward through halving. For examples, B^2 Network [4] and Merlin Chain [23] are working on ZK Rollups to scale Bitcoin; Bitlayer [9] takes a BitVM [26] approach to create a computational layer for Bitcoin, while LumiBit [21] adapts ZKEVM to achieve the same. In [27], ZeroSync has compressed the bootstrapping process (initial block download) into a single ZKP verification using a ZKP-circuit-friendly language called Cairo [13], thus dramatically reducing the time required to start a Bitcoin node. However,

[⋆] Formerly nChain researcher. Work done while at nChain.

in all the examples mentioned above, practically verifying ZKP on-chain has not yet been realised.

sCrypt [31] has implemented ZKP verification (Groth16, [19]) in Bitcoin Script on the BSV, a version of Bitcoin, achieving a script size of 1.2MB [32,33]. However, their implementation is not practical, and it falls short for three reasons. First, it cannot be deployed on the BSV mainnet without a collaborating miner. This is because of a policy that restricts the script size to 500kB [25], and non-policy-compliant transactions can only be accepted by other miners if the collaborating miner successfully mines a block. Second, it is not a faithful implementation of the Groth16 verifier as they hard-code in the script data which should be revealed at the point of spending, thus greatly limiting the applicability of their code. Third, their approach of using a compiler that converts TypeScript to Bitcoin Script generally leads to scripts of non-optimal size.

Our main contributions are:

– an implementation of bilinear pairings in Bitcoin Script, which has size[1] of 293.6kB, and that can be readily used on the BSV mainnet;
– an implementation of Groth16 verification in Bitcoin Script, which has size[2] of 466kB, and that can be readily used on the BSV mainnet;
– an analysis of the trade-off between script size and execution time caused by large number arithmetics (the smaller the script, the larger the numbers, hence the longer execution time);
– a significant reduction in transaction fees for on-chain ZKP verification as the transaction size is significantly reduced.

To achieve this level of optimisation, we use a combination of different techniques, each providing a significant reduction in script size:[3]

– stack management: being aware of positions of elements on the stacks and identifying the best arrangement of data elements and operations that results in the smallest script;
– no computation of inverses: designing the script to verify a candidate inverse instead of computing it;
– sparseness: working with field elements represented by polynomials that have many zero coefficients [30];
– seed choice: choosing an elliptic curve with seed having the smallest Hamming weight.

[1] All the sizes are cumulative of the locking and unlocking script size.

[2] The size reported here is for one public input, in Section 4 we also report the size for the Groth16 verifier with two public inputs.

[3] It is difficult to pinpoint where exactly the reductions come from, as they are a combination of all the techniques we employed. However, in the body of the text we provide rough estimations for each of the techniques we employ.

Our scripts are publicly available on GitHub.[4]. In the repository, readers can find a Python code used to generate our scripts,[5] another Python code used to generate the test data for benchmarking purposes, and the references to examples of on-chain transactions.

Our results enable, for the first time, practical ZKP verification on the mainnet of a Bitcoin version called BSV. [6] While Ethereum uses ZKPs for scalability, Bitcoin can also use them to enable smart contracts with greater flexibility. That is, in theory, one can run any computation off-chain and generate a ZKP, which is verified on chain, that the computation was done correctly. For example, the computation can be the validation of a token rule set, the enforcement of a financial contract, or the execution of instructions based on business logic and workflows.

The paper is structured as follows. In Section 2, we recall the preliminary notions we need in the rest of the paper, and we introduce the notation we use for Bitcoin scripts. In Section 3, we detail our implementation of the Optimal Ate Pairing and of the Groth16 verifier instatiated over the curve BLS12-381. Finally, in Section 4 we evaluate our scripts according to script size and execution time, and we compare the cost of verifying a ZKP on BSV and on Ethereum.

2 Preliminaries

2.1 Bitcoin

The Bitcoin blockchain [28] parses block data x into an ordered set of transactions $x := (\mathsf{tx}_1, \dots, \mathsf{tx}_n)$. Each transaction specifies a list of inputs and outputs. An output of a transaction is *spent* if it is referenced as an input of a valid transaction. An output can only be spent once.

Bitcoin uses a non-Turing complete, stack-based programming language in which spending conditions can be coded into *locking* scripts contained in outputs. Each input of a transaction contains an *unlocking* script, with the arguments needed to execute the locking script from the output referenced by the input. The spending is accepted if the execution terminates with true.

We think of the subroutines that make up a locking script as the implementations of functions $f(x_1, \dots, x_n)$, and of the elements in the unlocking script as the values $(\tilde{x}_1, \dots, \tilde{x}_n)$ over which the functions are evaluated. The unlocking script is then the implementation of a predicate (a function that returns true or false) whose calculation requires the evaluation of various functions (the subroutines) on the values supplied in the unlocking script.

[4] https://github.com/nchain-innovation/zkscript_package

[5] We generate our scripts as outputs of Python functions, so that they can be composed and shuffled around in an easy way. The approach we take is similar to that of BitVM [12], but they use Rust in place of Python.

[6] For our optimisations to work on BTC, we need large integer arithmetic.

Opcode	Operation
`OP_1SUB`	$[\![x_0 - 1]\!] \leftarrow [\![x_0]\!]$ `OP_1SUB`
`OP_DEPTH`	$[\![x_n]\!], \dots, [\![x_0]\!], [\![n+1]\!] \leftarrow [\![x_n]\!], \dots, [\![x_0]\!]$ `OP_DEPTH`
`OP_PICK`	$[\![x_n]\!], \dots, [\![x_0]\!], [\![x_i]\!] \leftarrow [\![x_n]\!], \dots, [\![x_0]\!], [\![i]\!]$ `OP_PICK`
`OP_EQUAL`	$[\![x_0 == x_1]\!] \leftarrow [\![x_0]\!], [\![x_1]\!]$ `OP_EQUAL`
`OP_VERIFY`	Pop x_0; fail if x_0 is false, otherwise, continue $\leftarrow [\![x_0]\!]$ `OP_VERIFY`
`OP_EQUALVERIFY`	`OP_EQUALVERIFY` = `OP_EQUAL` `OP_VERIFY`

Table 1. Opcodes used in this paper

Script execution A script is a sequence of opcodes and data objects. It is evaluated in reverse Polish notation by the Bitcoin Script engine, starting by pushing to the stack the arguments specified in the unlocking script.

The set of opcodes available for scripting depends on the implementation of Bitcoin. We will use the BSV implementation [11] because it supports large numbers (of size up to 10kB) and has the widest opcode support for arithmetic operations.[7] We list the opcodes used in this paper in Table 1; note that they are not all the ones needed in our implementations.

We introduce some notation that will be used throughout this work:

– $\langle\!\langle l \rangle\!\rangle$ denotes data hard-coded in the locking script. Thus, we will write

$$[\texttt{foo}] := \langle\!\langle l \rangle\!\rangle [\texttt{bar}]$$

to denote that the locking script [`foo`] consists of a subroutine script [`bar`] and hard-coded data $\langle\!\langle l \rangle\!\rangle$.

– $[\![x]\!]$ denotes data on top of the stack, i.e., the data we would get if we popped an element from the stack. More generally, $[\![x_0]\!], \dots, [\![x_n]\!]$ means that x_n is the data on top of the stack, x_{n-1} is the data below x_n (second from the top), and x_0 is the data buried at depth $n+1$ in the stack.

– $[\![y_0]\!], \dots, [\![y_m]\!] \leftarrow [\![x_0]\!], \dots, [\![x_n]\!]$ [`foo`]: Before executing [`foo`], the top $n+1$ elements of the stack are $[\![x_0]\!], \dots, [\![x_n]\!]$, and after executing [`foo`] the top $m+1$ elements are $[\![y_0]\!], \dots, [\![y_m]\!]$.

2.2 Pairings

Bilinear pairings are the building block of many important cryptographic primitives. The most efficient instantiation of a bilinear pairing is the Optimal Ate pairing [22], which is defined as a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ such that

$$e(n[A],[B]) = e([A],n[B]) = e([A],[B])^n$$

[7] BTC only allows number up to 4 bytes and has disabled various opcodes needed for arithmetic operations [1].

for any $[A] \in \mathbb{G}_1$, $[B] \in \mathbb{G}_2$ and $n \in \mathbb{Z}$. Here, $\mathbb{G}_1$ and $\mathbb{G}_2$ are subgroups of the group of points on some elliptic curves.

The value of e on $(P,Q) \in \mathbb{G}_1 \times \mathbb{G}_2$ is computed in two steps. First, we compute $\mathsf{miller}(P,Q) \in \mathbb{G}_T$, the output of the Miller loop [24], and then we perform a final exponentiation $\mathsf{miller}(P,Q)^\eta$, where η is an exponent depending on the instantiation of $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$. We set $e(P,Q) := \mathsf{miller}(P,Q)^\eta$.

In this paper, we instantiate the Optimal Ate Pairing over BLS12 curves [8] for their robustness against some known attacks [5–7,20] and their efficiency [3].

BLS12 curves BLS12 curves have the form $y^2 = x^3 + b \mod q$, where b is a parameter of the curve, and $q = (u-1)^2(u^4 - u^2 + 1)/3 + u$ is a prime dependent on a seed u. We write $E_{b,u}(\mathbb{F}_{q^n})$ to denote set of points of the BLS12 curve with parameters b and u over $\mathbb{F}_{q^n}$.

For BLS12-381, we have $u = -(2^{63} + 2^{62} + 2^{60} + 2^{57} + 2^{48} + 2^{16})$ and $b = 4$. When instantiating the Optimal Ate Pairing over this curve, $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are cyclic groups of order $r = u^4 - u^2 + 1$, $\mathbb{G}_1$ is a subgroup of $E_{b,u}(\mathbb{F}_q)$, while $\mathbb{G}_T = \mathbb{F}_{q^{12}}$. To construct $\mathbb{G}_2$, we set $\mathbb{F}_{q^2} = \mathbb{F}_q[t]/(1+t^2)$, and then $\mathbb{G}_2$ is a subgroup of $E_{b',u}(\mathbb{F}_{q^2})$, where $b' = (1+t)b \in \mathbb{F}_{q^2}$.

2.3 zkSNARKs

Circuits and NP relations. Let $\mathsf{C} : \mathbb{F}_r^{\ell+h} \to \{0,1\}$ be a polynomial-size arithmetic circuit over a finite field $\mathbb{F}_r$. The NP relation $\mathcal{R}_\mathsf{C}$ for C is defined as

$$\mathcal{R}_\mathsf{C} := \left\{(\boldsymbol{a}; \boldsymbol{w}) \in \mathbb{F}_r^\ell \times \mathbb{F}_r^h \mid \mathsf{C}(\boldsymbol{a}, \boldsymbol{w}) = 1\right\}.$$

The vector $\boldsymbol{a} = (a_1, \ldots, a_\ell)$ is the statement of the relation, sometimes also called the instance or public input, and the vector $\boldsymbol{w}$ is the witness or private input. The language associated to $\mathcal{R}_\mathsf{C}$ is $\mathcal{L}_\mathsf{C} := \{\boldsymbol{a} \in \mathbb{F}_r^n \mid \exists \boldsymbol{w} \in \{0,1\}^h \text{ s.t. } (\boldsymbol{a}; \boldsymbol{w}) \in \mathcal{R}_\mathsf{C}\}$.

zkSNARKs A preprocessing, zero-knowledge, succinct, non-interactive, argument system of knowledge (zkSNARK[8]) for $\mathcal{R}_\mathsf{C}$ is a triplet of algorithms $\Pi := (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ such that Setup takes as input a security parameter λ and the description of the circuit C, and it outputs a pair of keys pk and vk. The prover Prove takes pk, the statement $\boldsymbol{a}$ and the witness $\boldsymbol{w}$ and outputs a proof $\pi \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$, that is $\pi \leftarrow \mathsf{Prove}(pk, \boldsymbol{a}, \boldsymbol{w})$. The verifier Verify takes vk, $\boldsymbol{a}$, π and it either accepts or rejects, that is $\{\mathsf{true}, \mathsf{false}\} \leftarrow \mathsf{Verify}(vk, \boldsymbol{a}, \pi)$. The proof purportedly is for the statement "$\boldsymbol{a} \in \mathcal{L}_\mathsf{C}$".

Groth16 Groth16 [19] follows the linear interactive proof paradigm [10] with security in the generic group model, and can thus be instantiated with pairings. Given $\pi = ([A]_1, [B]_2, [C]_1)$, the result of Verify on the public input

[8] In this paper, we use ZKP and zkSNARK interchangeably. It is understood that there is a subtle difference which is not relevant to this paper.

$\boldsymbol{a} = (a_1, \dots, a_\ell)$ and the proof π is the result of an equation of pairings:

$$e([A]_1, [B]_2) = e([\alpha]_1, [\beta]_2) \cdot e\left(\sum_{i=0}^{\ell} a_i [P_i]_1, [\gamma]_2\right) \cdot e([C]_1, [\delta]_2) \tag{1}$$

where $[\alpha]_1, [\beta]_2, [\gamma]_2, [\delta]_2$ and the $[P_i]_1$'s[9] are part of vk, and $a_0 = 1$.

Note that the equation (1) depends on C only via the number of public inputs. This means that a Bitcoin Script implementation of Verify will be independent of the complexity of the calculations happening in C, and will scale only according to the number of public inputs. Furthermore, the size of the proof π is fixed to that of three groups element, once again independent of C.

These properties make Groth16 the best candidate for implementing zk-SNARK verification on-chain as it induces the least transaction size and computational complexity, resulting in low fees and fast execution time.

3 Implementation of Pairings and Groth16 in Script

We now outline our implementation of the Optimal Ate Pairing and of the Groth16 verifier in Bitcoin Script. First, we break down the scripts into subroutines. That is, we look at the operations required to compute the Optimal Ate Pairing and to verify a Groth16 proof, and we decompose these operations into simpler ones that we efficiently implement. Second, in constructing the subroutines we assume that the spender (the prover in the zkSNARK framework) supplies to the script (the verifier) additional input data to simplify the proof verification. The script will then verify that the data supplied is the one purported to be, and use it if it is, or fail otherwise. See Section 3.1 for an example of the input data supplied by the spender.

Remark 1. In Bitcoin, the prover/verifier framework of zkSNARKs is transposed to that of a payer who constructs the locking script (the Groth16 verifier) and of a spender who shows a zero-knowledge proof to prove they have the right to spend. While in zkSNARKs only the verifier carries the burden of verifying the proof, in our setup we assume that also the prover performs part of the computation required to verify the proof, so to reduce the size of the Bitcoin Script Groth16 verifier. This means that there is an additional burden on the prover, which is quantified by the amount of work required to compute the product of the three Miller loops in (6). This added burden is acceptable as it is easier to increase efficiency of off-chain computations rather than optimising script size.

The Optimal Ate Pairing is computed as $e(P, Q) = \mathsf{miller}(P, Q)^\eta$, see Section 2.2. We implement it by first implementing the Miller loop, and then the final exponentiation. That is, we construct a script [`millerLoop`] that computes

[9] The $[P_i]$'s are the evaluations of the QAP polynomials of the public inputs corresponding to the R1CS system of C on the verifier pre-computed challenge (the so-called *toxic waste* generated in the setup), [18].

the function $(P,Q) \mapsto \mathsf{miller}(P,Q)$, and a script $[\texttt{finalExponentiation}]$ that computes the function $(-) \mapsto (-)^{\eta}$. More precisely, on input data

$$\begin{aligned} [\texttt{inputMillerLoop}] &= [\![\mathsf{aux}_{\texttt{miller}}]\!], [\![P]\!], [\![Q]\!] \\ [\texttt{inputFinalExponentiation}] &= [\![\mathsf{aux}_{\mathsf{exp}}]\!], [\![x_0]\!] \end{aligned} \tag{2}$$

where $x_0 \in \mathbb{F}_{q^{12}}$, and $\mathsf{aux}_{\mathsf{miller}}$, $\mathsf{aux}_{\mathsf{exp}}$ is auxiliary data, they compute

$$\begin{aligned} [\![\mathsf{miller}(P,Q)]\!] &\leftarrow [\texttt{inputMillerLoop}]\ [\texttt{millerLoop}] \\ [\![x_0^{\eta}]\!] &\leftarrow [\texttt{inputFinalExponentiation}]\ [\texttt{finalExponentiation}] \end{aligned} \tag{3}$$

Then the script implementing the Optimal Ate Pairing is

$$[\texttt{pairing}] = [\texttt{millerLoop}]\ [\texttt{finalExponentiation}] \tag{4}$$

Indeed, on input $[\texttt{inputPairing}] = [\![\mathsf{aux}_{\mathsf{exp}}]\!], [\![\mathsf{aux}_{\texttt{miller}}]\!], [\![P]\!], [\![Q]\!]$, we get

$$[\![e(P,Q)]\!] \leftarrow [\texttt{inputPairing}]\ [\texttt{pairing}]$$

We approach the Groth16 verifier in a similar way. By rearranging (1) from Section 2.3 using the bilinear properties of e and its definition, we see that Groth16 verification entails verifying the following equation

$$\begin{aligned} &\left(\mathsf{miller}([A]_1,[B]_2)\cdot \mathsf{miller}\left(\sum_{i=0}^{\ell} a_i[P_i]_1, -[\gamma]_2\right)\cdot \mathsf{miller}([C]_1,-[\delta]_2)\right)^{\eta} \\ &= e([\alpha]_1,[\beta]_2) \end{aligned} \tag{5}$$

Hence, we need a script $[\texttt{multiScalarMultiplication}]$ that computes the function $(a_1,\dots,a_\ell) \mapsto \sum_{i=0}^{\ell} a_i[P_i]_1$, and a script $[\texttt{tripleMillerLoop}]$ that computes the product of the three Miller loops in (5). Then, the Groth16 verifier is[10]

$$\begin{aligned} [\texttt{groth16Verifier}] = {} & [\texttt{multiScalarMultiplication}] \\ & [\texttt{tripleMillerLoop}]\ [\texttt{finalExponentiation}] \\ & \langle\!\langle e([\alpha]_1,[\beta]_2)\rangle\!\rangle\ \texttt{OP\_EQUALVERIFY} \end{aligned} \tag{6}$$

We now detail the challenges to implement the subroutines the make up $[\texttt{pairing}]$ and $[\texttt{groth16Verifier}]$, and our proposed solutions.

3.1 Optimising the Miller loop

The value $\mathsf{miller}(P,Q)$ for curves in the BLS12 family is computed according to algorithm (1), where $\mathrm{ev}_{\ell_{T,Q}}(P)$ denotes the evaluation of the line through T and Q at P.

[10] Note that $e([\alpha]_1,[\beta]_2)$ can be hard-coded because $[\alpha]_1,[\beta]_2$ are part of vk and are known before the proof is generated.

Algorithm 1 Miller Loop

```
Inputs: $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$, $u = \sum_{i=0}^{n} u_i 2^i$, $u_i \in \{-1, 0, 1\}$, $u_n \neq 0$
Output: $\mathsf{miller}(P, Q) \in \mathbb{G}_T$
  $\mathsf{out} \leftarrow 1$
  if $u_n = 1$ then
    $T \leftarrow Q$
  else
    $T \leftarrow -Q$
  end if
  for $i = n-1, \dots, 0$ do
    $\mathsf{out} \leftarrow \mathsf{out}^2$
    $T \leftarrow 2T$
    if $u_i = 1$ then
      $\mathsf{out} \leftarrow \mathsf{out} \cdot \mathrm{ev}_{\ell_{T,Q}}(P)$
      $T \leftarrow T + Q$
    else
      $\mathsf{out} \leftarrow \mathsf{out} \cdot \mathrm{ev}_{\ell_{T,-Q}}(P)$
      $T \leftarrow T - Q$
    end if
  end for
```

Seed choice To obtain the most efficient implementation of [`millerLoop`], we seek to minimise the length of the loop and the cost of performing the operations in each iteration. The length of the loop and the number of operations performed can be minimised by choosing a curve whose seed u has small Hamming weight (number of non-zero bits) and bit-length. Our choice is BLS12-381, for which u has bit-length 64 and Hamming weight equal to 6, see Section 2.2.

Sparseness. The number of operations performed in the Miller loop can be further reduced by leveraging *sparseness* as explained by Scott in [29]. Both out and the line evaluations belong to the finite field extension $\mathbb{F}_{q^{12}}$, but many of the coefficients of the line evaluations are zero (that is why Scott calls them sparse). Leveraging this knowledge, we reduce the size of the script required to multiply two line evaluations from 1kB (the size of our implementation of multiplication over $\mathbb{F}_{q^{12}}$, to 150 bytes (the size of our script for the multiplication of two sparse elements).

Remark 2. When implementing [`tripleMillerLoop`] for (6), instead of computing $\mathsf{miller}([A]_1, [B]_2)$, $\mathsf{miller}\left(\sum_{i=0}^{\ell} a_i [P_i]_1, -[\gamma]_2\right)$ and $\mathsf{miller}([C]_1, -[\delta]_2)$ one after the other, we parallelise the computation. Namely, as evaluating $\mathsf{miller}(-,-)$ means executing algorithm (1), instead of repeating the loop three times, we go through the loop once, and at every iteration we carry out the computations required by each of the three terms appearing in (6). In this way, we can multiply together the sparse elements coming from the various line evaluations, thus amplifying the size optimisation resulting from leveraging sparseness.

Verifying the gradient. Finally, we look at reducing the cost of performing the various operations required by the Miller loop. To update the value of T, we sum points in $\mathbb{G}_2 \subset E_{b',u}(\mathbb{F}_{q^2})$, while to update `out`, we need to compute line evaluations. The most inefficient part of these operations is the calculation of the gradient of the line through two points on the curve. The inefficiency is due to the fact that computing the gradient requires inverting an element in a finite field, an operation whose cost in Bitcoin Script is substantial.[11]

To avoid the overhead of computing the gradient on-chain, we verify a candidate provided in the unlocking script as part of the auxiliary data $\texttt{aux}_{\texttt{miller}}$. Namely, every time we need the gradient of the line through two points $R_1, R_2 \in \mathbb{G}_2$, we expect the gadient $\lambda \in \mathbb{F}_{q^2}$ to be supplied in $\texttt{aux}_{\texttt{miller}}$, and we verify that λ is computed correctely by verifying

$$\lambda \cdot (x_{R_2} - x_{R_1}) = y_{R_2} - y_{R_1}$$

where $R_i = (x_i, y_i) \in \mathbb{F}_{q^2} \times \mathbb{F}_{q^2}$. Note that verification is very efficient, as once λ is verified, it can be used multiple times. Putting it into numbers, verifying the gradient instead of computing it on chain allows us to save roughly $3 \cdot \log(q) = 3 \cdot 381 \sim 1100$ bytes.

3.2 Optimising the final exponentiation

Final exponentiation is the same for [`pairing`] and [`groth16Verifier`], and it entails raising an element of $\mathbb{F}_{q^{12}}$ to the power η. To minimise the script size of [`finalExponentiation`], we follow the standard approach in the literature and split the final exponentiation in an easy and a hard part

$$[\texttt{finalExponentiation}] = [\texttt{easyExponentiation}]\ [\texttt{hardExponentiation}]$$

In the hard part, we leverage the Frobenius map $\mathbb{F}_{q^n} \to \mathbb{F}_{q^n}, z \to z^q$, to fix the cost of [`hardExponentiation`] to (roughly) that of performing five exponetiations to the power u.

For the easy part, we need to compute one Frobenius map, and to invert an element in $\mathbb{F}_{q^{12}}$. Instead of performing inversion on-chain, similarly to what we did in the Miller loop, we verify an inverse candidate supplied in the unlocking script as part of the auxiliary data $\texttt{aux}_{\texttt{exp}}$. Namely, as we need the inverse of an element $z \in \mathbb{F}_{q^{12}}$, we expect the inverse z' to be supplied in $\texttt{aux}_{\texttt{exp}}$, and on-chain we verify $z \cdot z' = 1 \in \mathbb{F}_{q^{12}}$. This allows us to fix the cost of [`easyExponentiation`] to constant (it is independent of the curve parameters).

Remark 3. Even if the Frobenius map entails raising an element to the power q, its implementation is of constant size because it only requires multiplying the components of $z \in \mathbb{F}_{q^n}$ by some constants.

[11] If $z \in \mathbb{F}_q$, then inverting z in Bitcoin Script requires $O(\log(q))$ operations using Fermat's Little Theorem.

3.3 Optimising the multi scalar multiplication

The hardest subroutine to optimise in (6) is [`multiScalarMultiplication`]. The reason is that this subroutine computes $\sum_{i=0}^{\ell} a_i[P_i]_1$, which depends both on circuit-specific values: the $[P_i]'s$, see Section 2.3, and on values supplied by the prover: the public inputs $a_1, \ldots, a_\ell$.

As the public inputs are supplied by the prover, they are not known when [`multiScalarMultiplication`] is constructed, and therefore the script must take into account the worst case scenario, namely, $a_i = r$.

The cost of computing $\sum_{i=0}^{\ell} a_i[P_i]_1$ scales linearly with ℓ, which is unfortunate as a single multiplication $a_i[P_i]_1$ costs about 35kB via double-and-add (and verifying the gradient as in Section 3.1). To optimise the size of the script we use a standard trick: pass the ℓ public inputs a_i of C as witness and a hash of them as public input. This makes the size of the script independent of the number of public inputs at the cost of increasing the computational burden of the prover.

More specifically, let $H : \{0,1\}^* \to \mathbb{F}_r^d$ be a cryptographic hash function where $\mathbb{F}_r$ is the field over which C is defined. Then, the augmented relation for which we prove satisfiability is

$$\mathcal{R}' := \left\{ ((h_1, \ldots, h_d); (a_1, \ldots, a_\ell, \boldsymbol{w})) \;\middle|\; \begin{matrix} \mathsf{C}(a_1, \ldots, a_\ell, \boldsymbol{w}) = 1 \\ (h_1, \ldots, h_d) = H(a_1, \ldots, a_\ell) \end{matrix} \right\}.$$

In this way, we keep the size of the script fixed to that of a Groth16 verifier for a circuit with d public inputs. Indeed, in a proof for relation $\mathcal{R}'$ the prover supplies d public inputs $h_1, \ldots, h_d$ for which the circuit corresponding to $\mathcal{R}'$ verifies that $(h_1, \ldots, h_d)$ is the digest of $(a_1, \ldots, a_\ell)$, the original public inputs that are now passed as private inputs, and that $\mathsf{C}(a_1, \ldots, a_\ell, \boldsymbol{w}) = 1$. The public inputs $h_1, \ldots, h_d$ are a commitment to the public inputs $a_1, \ldots, a_\ell$.

For example, if $\ell > 2$ we can set H to be the (vector) Pedersen hash over the JubJub curve [34], whose base field is the scalar field $\mathbb{F}_r$ of BLS12-381. A Pedersen hash digest is just a single group element of JubJub $h = (h_1, h_2) \in \mathbb{F}_r^2$. Thus, $d = 2$ and the verification script only needs to compute $h_1[P_1'] + h_1[P_2']$, instead of an ℓ-multi scalar multiplication.

3.4 Subroutine-independent optimisations

In this section we detail some optimisations that we apply to all the subroutines appearing in (4) and (6).

Stack management Stacks are data structures equipped only with push and pop operations, which means that we can only access the top element of the stack. This property makes storage and retrieval of temporary variables a task with great impact on script size.

During script execution, the Bitcoin Script Engine has two stacks at its disposal, the main stack, also referred to as the stack, and the altstack. One can only push and pull elements from the altstack, which is why it is customary to use it to store variables. We take a different approach, we use the bottom of the stack instead. As the depth of the stack can be obtained with the opcode OP_DEPTH, the bottom of the stack can be thought to have a fixed position, and can be used to store variables.

The variable we need more often in [pairing] and [groth16Verifier] is q, which we store at the bottom of the stack, and fetch with the following script

$$[\texttt{fetch}_q] = \texttt{OP\_DEPTH OP\_1SUB OP\_PICK} \tag{7}$$

In this way, we save ~ 50 bytes compared to pushing q to the stack every time we need it.

Remark 4 (Make fetching secure). As there is no way to efficiently push an element to the bottom of the stack, we assume q is supplied in the unlocking script as part of the auxiliary data. To ensure that it is the one we assume it to be, i.e., the parameter q of BLS12-381, we use the following script

$$[\texttt{verify}_q] = \texttt{OP\_DEPTH OP\_1SUB OP\_PICK} \langle\langle q \rangle\rangle \texttt{ OP\_EQUALVERIFY}$$

Arithmetic over finite fields All the subroutines in (4) and (6) require arithmetic over (a finite field extension of) $\mathbb{F}_q$. The biggest impact of finite field arithmetic on script size comes from modulo operations by q. To efficiently mod by q, we employ two techniques.

First, as taking the residue class modulo q is a homomorphism $\mathbb{Z} \to \mathbb{F}_q$, instead of taking a modulo after every operation, we do it only once in a while. A similar approach was taken in [17], but we improve it by using the *modulo threshold*, i.e., the upper bound on the size of the numbers during script execution, as a parameter of the script. Tuning this parameter we have a trade-off between script size and execution time, see Section 4.

Second, we batch modulo operations, so that q must be fetched only once. We explain the technique in the case of addition, but it can be applied to any other operation. As elements of $\mathbb{F}_{q^n}$ are given by tuples $(z_1, \dots, z_n)$ of elements in $\mathbb{F}_q$, computing $(z_1, \dots, z_n) + (\tilde{z}_1, \dots, \tilde{z}_n)$ means computing $z_i + \tilde{z}_i \mod q$ for $i = 1, \dots, n$. Being Bitcoin Script a stack-based language, we must compute each component $z_i + \tilde{z}_i \mod q$ and place it on top of the stack. Instead of sequentially computing $z_i + \tilde{z}_i \mod q$ for $i = 1, \dots, n$, we compute $z_i + \tilde{z}_i$ for $i = n, \dots, 1$, place them on the altstack, and then sequentially take the modulo of each element. With this technique, we save $(n - 1)$ bytes for every modulo operation.

Remark 5 (Preventing overflows). As we remarked in Section 2, the BSV implementation supports large numbers. However, policy restrictions dictate that the numbers must fit in 10kB. To avoid overflows, we proceed as follows. As the operations executed in [groth16Verifier] are fixed, and we know the largest size of the input data fed to the script, when constructing the script we keep track of the size of the numbers we are working with. For example, if we multiply two numbers of bit size at most $|q|$, we know that the result has bit size at most $2|q|$. Then, in the script we reduce modulo q before the numbers overflow.

We go one step further: we introduce a modulo threshold variable that is supplied at the point of script construction and that dictates when to perform modulo operations. See Section 4.1 for more information.

4 Script benchmarking

We now benchmark our scripts according to three metrics: script size, script execution time, and the cost of publishing a transaction with the script on-chain. Based on the first two metrics, we select the optimal modulo threshold, see Section 4.1, and then we compare the monetary cost of executing our script to that of executing an equivalent script on Ethereum, see Section 4.2.

4.1 Script size and execution time

When constructing the [`pairing`] and [`groth16Verifier`], we can choose the threshold after which modulo operations are carried out. Namely, we can choose the largest size the numbers can reach during script execution before we mod by q and bring them back to $\mathbb{F}_q$. Changing the threshold for modulo operations allows us to strike a balance between script size and execution time. Indeed, the more often we mod by q, the bigger the script size, but the lower the execution time of the script, as it will work with smaller numbers.

Below, we plot the threshold for the modulo operations against script size and execution time, respectively, for [`pairing`] and [`groth16Verifier`] with one and two public parameters, i.e., $\ell = 1$ and $\ell = 2$. We run our tests in a BSV regtest v1.0.8 on a processor Intel Core i7, 2.6 Ghz, 6-Core.

While BSV can support transactions with arbitrary script size and execution time, and with numbers of length up to 750kB, current policy restrictions impose that the locking script is at most 500kB, that it executes in at most 1 second, and that the numbers[12] must fit in 10kB [25].

Figure 1 shows that the size of [`pairing`] decreases rapidly when the modulo threshold increases from 50 bytes (which implies that we mod by q after every operation) to 2kB. Further increases of the modulo threshold result in small further decreases of the script size, but at the cost of a higher execution time. In particular, when the modulo threshold reaches 4kB, the execution time approaches the policy threshold of 1 second. As there is not a big difference in either script size or execution time when the modulo threshold passes from 2kB to 3kB, we take a conservative stance and choose 2kB as the optimal modulo threshold for the [`pairing`]. This choice results in a script of size 286kB and with execution time of circa 0.47s.

Figure 2 shows that the size of [`groth16Verifier`] behaves similarly. Namely, the size of the script decreases rapidly when the modulo threshold increases from

[12] By numbers we mean elements on the stack that are used in mathematical operations.

Fig. 1. Size and execution time of [`pairing`] as functions of the modulo threshold

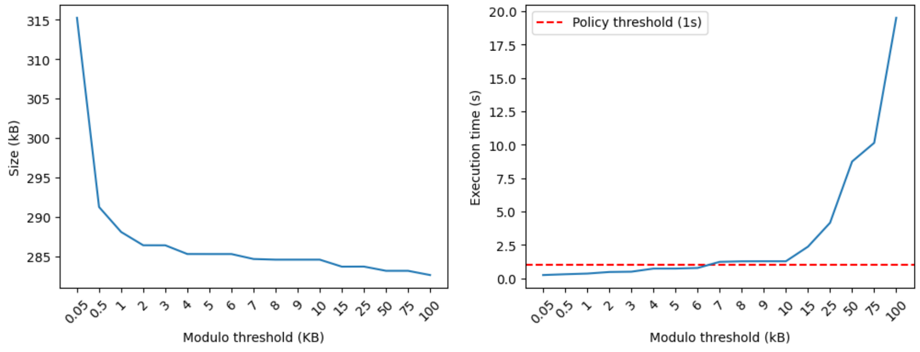
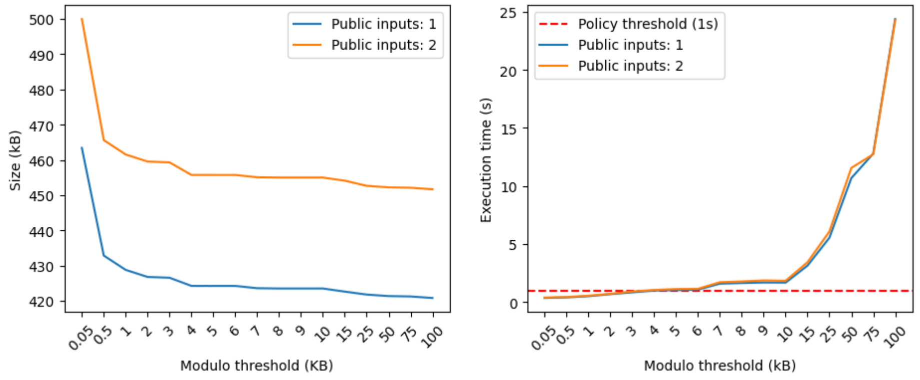


Fig. 2. Size and execution time of [`groth16Verifier`] as functions of the modulo threshold



50 bytes to 2kB, but further increases do not decrease the script size too much. For Groth16, we approach the consensus threshold of 1s execution time when the modulo threshold approaches 3kB. We choose as optimal modulo threshold 2kB, which results in script of sizes 426kB and 460kB, for one and two public inputs, respectively, and with execution times of circa 0.67s and 0.70s, respectively.

Remark 6. In the Figures 1 and 2, we focus on the size of the locking script because the modulo threshold does not affect the size of the unlocking script. However, in Section 4.2 we will take into account both locking and unlocking script, as the cost of publishing of script on-chain depends on both.

4.2 Monetary cost

In this section, we compare the transaction fees for the Optimal Ate Pairing and the Groth16 verifier on BSV and Ethereum, respectively. For simplicity, the comparison only focuses on the cost for the computation to be done by the nodes in a network. We do not take other factors such as the cost to maintain the respective network or the time it takes for the transaction to be confirmed and published.

The results presented in Table 2 and Table 3 make it apparent that as of May 2024, it is much cheaper to execute bilinear pairings and the Groth16 verifier on BSV than on Ethereum.[13]

In BSV, a miner executes a script $[\mathtt{S}] = [\mathtt{unlock}][\mathtt{lock}]$ if there is a transaction $\mathsf{tx}_{\mathsf{lock}}$ with an output $\mathsf{txOut}_{\mathsf{lock}}$ that has $[\mathtt{lock}]$ as locking script, and there is a transaction $\mathsf{tx}_{\mathsf{unlock}}$ with an input $\mathsf{txIn}_{\mathsf{unlock}}$ that spends $\mathsf{txOut}_{\mathsf{lock}}$ and that has $[\mathtt{unlock}]$ as unlocking script. In this situation, consensus requires a miner to execute $[\mathtt{S}]$, regardless of what operations are contained in it.[14] Thus, we model the cost of executing a script in BSV as the dollar value of the transaction fees required to publish $[\mathtt{S}]$ on-chain.

From the analysis of Section 4.1, we see that the optimal modulo thresholds for the Optimal Ate Pairing and the Groth16 verifier are given by 2kB in both cases. The script size for $[\mathtt{pairing}]$ and $[\mathtt{groth16Verifier}]$ can be read off from Figure 1 and Figure 2, and are 286kB for $[\mathtt{pairing}]$, and 426kB, 460kB for $[\mathtt{groth16Verifier}]$ with $\ell = 1$, $\ell = 2$, respectively; the size of $[\mathtt{unlockPairing}]$, see (2), is 7.6kB, while the size of the unlocking script of $[\mathtt{groth16Verifier}]$ with $\ell = 1$ is 40kB, and with $\ell = 2$ is 60kB.

To estimate fee rates on the BSV blockchain and the BSVUSD conversion rate, we download data from WhatsOnChain.com [35]. We consider the average fee rate and the exchange rate for the period going from 15/03/2024 to 12/06/2024. We trim the series by removing the values below the 5th percentile and above the 95th percentile. Then, we take the 25th, 50th, and 75th percentile from the time series of fee rates as estimates of low, medium and high fee rates. They come out to be: 57, 79 and 120 sats/kB, respectively. As estimate of the exchange rate, we take the average price of the trimmed time series, which is \$73 per BSV. The results of the calculations are presented in Table 2 for the Optimal Ate Pairing, and in Table 3 for Groth16 verifier.

The cost to execute an Ethereum contract is proportional to the computational complexity of the underlying code, with computational units measured in terms of gas. Hence, the cost of executing a contract in Ethereum is given by how many units of gas it requires, and gas cost at the time of execution.

Since EIP-1108 [2], the cost of executing one pairing is of 79000 $(= 34000 + 45000)$ gas units, whereas the cost of executing the Groth16 verifier is of 153150 $(= 34000 \cdot 3 + 45000 + 6150$, see [2]) gas units for one public statement, and 159300 for two public statements. We estimate the cost of executing the scripts with the same method as for BSV fee rates. We download the data from Etherscan.io [15], and we calculate three fee rates: low: 11 gwei/gas,[15] medium: 16 gwei/gas, and

[13] Note that Ethereum uses a different curve. However, their implementation of pairings is state-of-the-art, so the comparison made here can be considered fair.

[14] The only exception is if the script does not abide by the policies set forth by the miner. For the purpose of this analysis we assume that $[\mathtt{S}]$ satisfies such policies.

[15] 1 gwei equates to 10^{-9} ETH.

Blockchain \ Fee rate	Low	Medium	High
BSV	\$0.012	\$0.016	\$0.025
Ethereum	\$2.95	\$4.29	\$6.43

Table 2. Cost of executing the Optimal Ate Pairing in BSV and Ethereum.

	Groth16 verifier ($\ell = 1$)			Groth16 verifier ($\ell = 2$)		
Blockchain \ Fee rate	Low	Medium	High	Low	Medium	High
BSV	\$0.018	\$0.024	\$0.037	\$0.0019	\$0.026	\$0.040
Ethereum	\$5.78	\$8.32	\$12.48	\$5.95	\$8.65	\$12.97

Table 3. Cost of executing the Groth16 verifier in BSV and Ethereum.

high: 24 gwei/gas, as well as an estimated exchange rate of \$3394 per ETH.[16] The results of the calculations are presented in Table 2 for the Optimal Ate Pairing, and in Table 3 for the Groth16 verifier.

5 Conclusion

We have demonstrated not only that it is practical to implement the Groth16 verifier in Bitcoin Script, but also that the cost of executing it is much cheaper than that of executing an equivalent script in Ethereum, see Section 4. As part of our future work, we plan to implement more pairing-based cryptographic primitives in Bitcoin Script so that the Bitcoin blockchain can leverage this fruitful area of cryptography to its full strength.

References

1. Bitcoin (BTC) Wiki, Script. `https://en.bitcoin.it/wiki/Script`
2. Antonio Salazar Cardozo, Zachary Williamson: EIP-1108: Reduce alt_bn128 precompile gas costs," Ethereum Improvement Proposals, no. 1108, May 2018. `https://eips.ethereum.org/EIPS/eip-1108`
3. Aranha, D.F., El Housni, Y., Guillevic, A.: A survey of elliptic curves for proof systems. Designs, Codes and Cryptography **91**(11), 3333–3378 (2023)
4. B Squared: Zero-Knowledge Proof Verification Commitment for ZK-Rollup on Bitcoin. `https://docs.bsquared.network/zpvc`
5. Barbulescu, R., Duquesne, S.: Updating key size estimations for pairings. Journal of cryptology **32**, 1298–1336 (2019)

[16] Note that, even if the exchange rate of BSV per ETH were 1:1, then executing the scripts on BSV would still be much cheaper. For example, at the price of \$3394 per BSV, the Optimal Ate Pairing at high fee rate would cost only \$1.16.

6. Barbulescu, R., Gaudry, P., Guillevic, A., Morain, F.: Improving nfs for the discrete logarithm problem in non-prime finite fields. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 129–155. Springer (2015)
7. Barbulescu, R., Gaudry, P., Joux, A., Thomé, E.: A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–16. Springer (2014)
8. Barreto, P.S., Kim, H.Y., Lynn, B., Scott, M.: Efficient algorithms for pairing-based cryptosystems. In: Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22. pp. 354–369. Springer (2002)
9. Bit Layer: Bitlayer: A Bitcoin Computational Layer Architecture Based on the BitVM Paradigm. `https://static.bitlayer.org/Bitlayer-Technical-Whitepaper.pdf`
10. Bitansky, N., Chiesa, A., Ishai, Y., Ostrovsky, R., Paneth, O.: Succinct non-interactive arguments via linear interactive proofs. In: Sahai, A. (ed.) Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings
11. Bitcoin SV: `https://github.com/bitcoin-sv/bitcoin-sv`
12. BitVM: Official BitVM implementation in Rust. `https://github.com/BitVM/BitVM`
13. Cairo: The Cairo Programming Language. `https://book.cairo-lang.org/title-page.html#the-cairo-book`
14. Ethereum Org: ZERO-KNOWLEDGE ROLLUPS. `https://ethereum.org/en/developers/docs/scaling/zk-rollups/`
15. Etherscan: etherscan.io. `https://etherscan.io/`
16. Firo: `https://firo.org`
17. franchfrog42: zkBaguette applied to zkSnarks. `https://github.com/frenchfrog42/zk-hackaton`
18. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings
19. Groth, J.: On the size of pairing-based non-interactive arguments. In: Advances in Cryptology - EUROCRYPT 2016
20. Kim, T., Barbulescu, R.: Extended tower number field sieve: A new complexity for the medium prime case. In: Annual international cryptology conference. pp. 543–571. Springer (2016)
21. LumiBit: LumiBit's ZK-EVM. `https://lumibit.gitbook.io/lumibit-gitbook/overview/lumibit-101/lumibits-zk-evm`
22. Matsuda, Seiichi and Kanayama, Naoki and Hess, Florian and Okamoto, Eiji: Optimised Versions of the Ate and Twisted Ate Pairings. In: Cryptography and Coding. pp. 302–312. Springer Berlin Heidelberg (2007)
23. Merlin Chain: ZK Rollup on Bitcoin. `https://docs.merlinchain.io/merlin-docs/zk-rollup-on-bitcoin`
24. Miller, V.S.: The weil pairing, and its efficient calculation. Journal of Cryptology **17**(4), 235–261 (Sep 2004). `https://doi.org/10.1007/s00145-004-0315-8`
25. Patrick Fromberg: BSV Consensus Limits. `https://github.com/bitcoin-sv/bitcoin-sv/wiki/Consensus-Limits`

26. R. Linus: BitVM: Computing Anything on Bitcoin. `https://bitvm.org/bitvm.pdf`
27. Robin Linus and Lukas George: ZeroSync: Introducing Validity Proofs to Bitcoin. `https://zerosync.org/zerosync.pdf`
28. S. Nakamoto: Bitcoin: A Peer-to-Peer Electronic Cash System
29. Scott, M.: Pairing implementation revisited. Cryptology ePrint Archive, Paper 2019/077 (2019), `https://eprint.iacr.org/2019/077`, `https://eprint.iacr.org/2019/077`
30. Scott, M.: A note on twists for pairing friendly curves. `http://indigo.ie/~mscott/twists.pdf`
31. sCrypt Inc: `https://scrypt.io`
32. sCrypt Inc: sCrypt Transaction. `https://test.whatsonchain.com/tx/2396a4e52555cdc29795db281d17de423697bd5cbabbcb756cb14cea8e947235`
33. sCrypt Inc: Tutorial 5: Zero Knowledge Proofs. `hhttps://docs.scrypt.io/tutorials/zkp`
34. Sean Bowe: `https://github.com/zkcrypto/jubjub`
35. TAAL: whatsonchain.com. `https://whatsonchain.com/`
36. zCash: `https://z.cash`