NeRO: BitVM2-Based Optimistic Verifiable Computation on Bitcoin

October 31, 2024

Distributed Lab

distributedlab.com/

github.com/distributed-lab/nero

Plan

1. Advanced Bitcoin Script
   - What is Bitcoin Script? Basic Notation
   - Non-Native Verifications
   - Demystifying Math behind BitVM Groth16
2. BitVM2
   - Shard Splitting
   - Protocol
3. BitVM2 Pitfalls
   - Splitting Mechanism
   - BitVM-friendliness

Advanced Bitcoin Script

What Bitcoin Script is for?

Recall

Bitcoin Script is a scripting language used in Bitcoin to specify conditions on how the UTXO can be spent.

What Bitcoin Script is for?

Recall

Bitcoin Script is a scripting language used in Bitcoin to specify conditions on how the UTXO can be spent.

Example

The standard `pay-to-pubkey-hash` looks as follows.
`scriptPubKey`:

Script: `OP_DUP OP_HASH160` $\langle H(\mathrm{pk}) \rangle$ `OP_EQUALVERIFY OP_CHECKSIG`

What Bitcoin Script is for?

Recall

Bitcoin Script is a scripting language used in Bitcoin to specify conditions on how the UTXO can be spent.

Example

The standard `pay-to-pubkey-hash` looks as follows.

`scriptPubKey`:

Script: `OP_DUP OP_HASH160` $\langle H(\mathsf{pk}) \rangle$ `OP_EQUALVERIFY OP_CHECKSIG`

As a `scriptSig`, the user provides $\{ \langle \sigma \rangle \ \langle \mathsf{pk} \rangle \}$.

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` $\|$ `scriptPubKey`:

Script: $\langle\sigma\rangle$ $\langle\mathsf{pk}'\rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathsf{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` $\|$ `scriptPubKey`:

Script: $\langle\sigma\rangle$ $\langle \mathsf{pk}' \rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathsf{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathsf{pk}' \rangle$ $\langle \mathsf{pk}' \rangle$ `OP_HASH160` $\langle H(\mathsf{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

How the Bitcoin Script is executed

Consider the pay-to-pubkey-hash's scriptSig || scriptPubKey:

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ OP_DUP OP_HASH160 $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle \mathrm{pk}'\rangle$ OP_HASH160 $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle H(\mathrm{pk}')\rangle$ $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

How the Bitcoin Script is executed

Consider the pay-to-pubkey-hash's scriptSig || scriptPubKey:

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ OP_DUP OP_HASH160 $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle \mathrm{pk}'\rangle$ OP_HASH160 $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle H(\mathrm{pk}')\rangle$ $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ OP_CHECKSIG

How the Bitcoin Script is executed

Consider the pay-to-pubkey-hash's scriptSig || scriptPubKey:

Script: $\langle\sigma\rangle$ $\langle\mathsf{pk}'\rangle$ OP_DUP OP_HASH160 $\langle H(\mathsf{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle\mathsf{pk}'\rangle$ $\langle\mathsf{pk}'\rangle$ OP_HASH160 $\langle H(\mathsf{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle\mathsf{pk}'\rangle$ $\langle H(\mathsf{pk}')\rangle$ $\langle H(\mathsf{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle\mathsf{pk}'\rangle$ OP_CHECKSIG

Script: $\langle 1\rangle$

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` || `scriptPubKey`:

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ OP_DUP OP_HASH160 $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle \mathrm{pk}'\rangle$ OP_HASH160 $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle H(\mathrm{pk}')\rangle$ $\langle H(\mathrm{pk})\rangle$ OP_EQUALVERIFY OP_CHECKSIG

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ OP_CHECKSIG

Script: $\langle 1\rangle$

Note

One can spend the UTXO iff the output is OP_1.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

✓ Hash Preimage Verification and Timelocks.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

✓ Hash Preimage Verification and Timelocks.

✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

- ✓ Hash Preimage Verification and Timelocks.
- ✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).
- ✓ Threshold/Multisignatures.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

✓ Hash Preimage Verification and Timelocks.

✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).

✓ Threshold/Multisignatures.

✓ Combination of those.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

✓ Hash Preimage Verification and Timelocks.

✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).

✓ Threshold/Multisignatures.

✓ Combination of those.

Question

Can we implement some non-native verifications? For example, zk-SNARKs (Groth16, fflonk), zk-STARKs, BLS Signatures?

Can we do more?

✓ Groth16 is already implemented.

Can we do more?

- ✓ Groth16 is already implemented.
- ✓ fflonk is already implemented.

Can we do more?

✓ Groth16 is already implemented.

✓ fflonk is already implemented.

✗ zk-STARK cannot be currently implemented (requires `OP_CAT` for Fiat-Shamir transformation and Merkle Trees). Yet, assuming `OP_CAT`, the **Circle STARK** is implemented!

Can we do more?

✓ Groth16 is already implemented.

✓ fflonk is already implemented.

✗ zk-STARK cannot be currently implemented (requires OP_CAT for Fiat-Shamir transformation and Merkle Trees). Yet, assuming OP_CAT, the **Circle STARK** is implemented!

✓ Any discrete-log-based protocol that does not involve hashing (typically requiring concatenation) can be implemented: Σ-protocols, Bulletproofs, BLS Signatures.

Can we do more?

✓ Groth16 is already implemented.

✓ fflonk is already implemented.

✗ zk-STARK cannot be currently implemented (requires `OP_CAT` for Fiat-Shamir transformation and Merkle Trees). Yet, assuming `OP_CAT`, the **Circle STARK** is implemented!

✓ Any discrete-log-based protocol that does not involve hashing (typically requiring concatenation) can be implemented: Σ-protocols, Bulletproofs, BLS Signatures.

Note

In other words, currently, it is *theoretically possible* to build a Groth16 zk-SNARK verification of proof π in a form

Script: $\langle\pi\rangle$ $\langle$public statement$\rangle$ `OP_CHECKGROTH16`

Demystifying Math behind BitVM Groth16

✓ Arithmetic is allowed only over `u32` integers.

[1] With dropping variations such as `OP_1ADD` and such.

Demystifying Math behind BitVM Groth16

- ✓ Arithmetic is allowed only over u32 integers.
- ✓ From arithmetic, one only[1] has OP_ADD, OP_SUB, OP_NEGATE, OP_ABS, OP_LESSTHAN, OP_GREATERTHAN, OP_BOOLAND, OP_BOOLOR.

[1] With dropping variations such as OP_1ADD and such.

Demystifying Math behind BitVM Groth16

- ✓ Arithmetic is allowed only over u32 integers.
- ✓ From arithmetic, one only[1] has OP_ADD, OP_SUB, OP_NEGATE, OP_ABS, OP_LESSTHAN, OP_GREATERTHAN, OP_BOOLAND, OP_BOOLOR.
- ✓ Flow control is very limited: only OP_IFs/OP_ELSEs are allowed. No for/while loops, but almost full control of compile-time stack movement.

[1] With dropping variations such as OP_1ADD and such.

Demystifying Math behind BitVM Groth16

- ✓ Arithmetic is allowed only over `u32` integers.
- ✓ From arithmetic, one only[1] has `OP_ADD`, `OP_SUB`, `OP_NEGATE`, `OP_ABS`, `OP_LESSTHAN`, `OP_GREATERTHAN`, `OP_BOOLAND`, `OP_BOOLOR`.
- ✓ Flow control is very limited: only `OP_IF`s/`OP_ELSE`s are allowed. No for/while loops, but almost full control of compile-time stack movement.

Question

(Almost) any zk-SNARK requires working over large finite fields (with a bit-size of 254). How do we even push a 254-bit big integer?

[1] With dropping variations such as `OP_1ADD` and such.

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Numbers $x_0, x_1, \ldots, x_{n-1}$ are called **limbs**, where n is the **limb-size** of x in base b.

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Numbers $x_0, x_1, \ldots, x_{n-1}$ are called **limbs**, where n is the **limb-size** of x in base b.

Idea #1

If b is small enough, we can publish individual limbs $x_0, \ldots, x_{n-1}$ that constitute the whole number x.

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Numbers $x_0, x_1, \ldots, x_{n-1}$ are called **limbs**, where n is the **limb-size** of x in base b.

Idea #1

If b is small enough, we can publish individual limbs $x_0, \ldots, x_{n-1}$ that constitute the whole number x.

Idea #2

Since we want to minimize the number of limbs, we take the largest b possible (with $b = 2^t$ for convenience). Thus, we set $b := 2^{30}$.

Representing large integers

Example

Consider the following 254-bit integer:

$$x = (\texttt{0xbe48fffd2a6f534dc} \\ \texttt{5b6a6901840fc0fb65827e6} \\ \texttt{efd22a8063cded681f5f7b2})$$

Representing large integers

Example

Consider the following 254-bit integer:

$$x = (\texttt{0xbe48fffd2a6f534dc}$$
$$\texttt{5b6a6901840fc0fb65827e6}$$
$$\texttt{efd22a8063cded681f5f7b2})$$

To add this integer to the stack, one uses the following script:

Script:

```
OP_PUSHBYTES_2 ⟨e40b⟩ OP_PUSHBYTES_4 ⟨a9f4ff23⟩
OP_PUSHBYTES_4 ⟨c54d532f⟩ OP_PUSHBYTES_4 ⟨06a4a92d⟩
OP_PUSHBYTES_4 ⟨fbc00f04⟩ OP_PUSHBYTES_4 ⟨9b9f6019⟩
OP_PUSHBYTES_4 ⟨802ad22f⟩ OP_PUSHBYTES_4 ⟨5a7bf318⟩
OP_PUSHBYTES_4 ⟨b2f7f501⟩
```

Representing large integers

Example

Consider the following 254-bit integer:

$$x = (\texttt{0xbe48fffd2a6f534dc}$$
$$\texttt{5b6a6901840fc0fb65827e6}$$
$$\texttt{efd22a8063cded681f5f7b2})$$

To add this integer to the stack, one uses the following script:

Script:

```
OP_PUSHBYTES_2 ⟨e40b⟩ OP_PUSHBYTES_4 ⟨a9f4ff23⟩
OP_PUSHBYTES_4 ⟨c54d532f⟩ OP_PUSHBYTES_4 ⟨06a4a92d⟩
OP_PUSHBYTES_4 ⟨fbc00f04⟩ OP_PUSHBYTES_4 ⟨9b9f6019⟩
OP_PUSHBYTES_4 ⟨802ad22f⟩ OP_PUSHBYTES_4 ⟨5a7bf318⟩
OP_PUSHBYTES_4 ⟨b2f7f501⟩
```

Note: One needs **9 limbs** to represent a 254-bit integer.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

1. On step i, calculate $t \leftarrow x_i + y_i + \text{carry}$ (start with zero carry).

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

1. On step i, calculate $t \leftarrow x_i + y_i + \text{carry}$ (start with zero carry).
2. If $t < 2^{30}$, set $z_i \leftarrow t$, carry $\leftarrow 0$.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

1. On step i, calculate $t \leftarrow x_i + y_i + \text{carry}$ (start with zero carry).
2. If $t < 2^{30}$, set $z_i \leftarrow t$, carry $\leftarrow 0$.
3. If $t \geq 2^{30}$, set $z_i \leftarrow t - 2^{30}$, carry $\leftarrow 1$.

BitInt Addition: Bitcoin Script

Algorithm 7: Adding two integers assuming with no overflow

Input : Two integers on the stack: $\{\ \langle x_{\ell-1}\rangle \ \dots\ \langle x_0\rangle\ \langle y_{\ell-1}\rangle\ \dots\ \langle y_0\rangle\ \}$

Output: Result of addition $z = x + y$ in a form $\{\ \langle z_{\ell-1}\rangle\ \dots\ \langle z_0\rangle\ \}$

$\{$ `OP_ZIP` $\}$; `/* Convert current stack` $\{\ \langle x_{\ell-1}\rangle\ \dots\ \langle x_0\rangle\ \langle y_{\ell-1}\rangle\ \dots\ \langle y_0\rangle\ \}$ `to the form` $\{\ \langle x_{\ell-1}\rangle\ \langle y_{\ell-1}\rangle\ \dots\ \langle x_0\rangle\ \langle y_0\rangle\ \}$ `*/`

$\{\ \langle\beta\rangle\ \}$; `/* Push base to the stack */`

$\{$ `OP_LIMB_ADD_CARRY OP_TOALTSTACK` $\}$

for $\_ \in \{0,\dots,\ell-3\}$ **do**

    `/* At this point, stack looks as` $\{\ \langle x_n\rangle\ \langle y_n\rangle\ \langle\beta\rangle\ \langle c\rangle\ \}$ `. We need to add carry` c `and call OP_LIMB_ADD_CARRY */`

    $\{$ `OP_ROT` $\}$

    $\{$ `OP_ADD` $\}$

    $\{$ `OP_SWAP` $\}$

    $\{$ `OP_LIMB_ADD_CARRY OP_TOALTSTACK` $\}$

end

`/* At this point, again, stack looks as` $\{\ \langle x_n\rangle\ \langle y_n\rangle\ \langle\beta\rangle\ \langle c\rangle\ \}$ `. We need to drop the base, add carry, and conduct addition, assuming overflowing does not occur */`

$\{$ `OP_NIP OP_ADD, OP_ADD` $\}$

`/* Return all limbs to the main stack */`

for $\_ \in \{0,\dots,\ell-2\}$ **do**

    $\{$ `OP_FROMALTSTACK` $\}$

end

BigInt Multiplication

Problem

Given two 254-bit integers x and y, find 508-bit $z := x \times y$.

BigInt Multiplication

Problem

Given two 254-bit integers x and y, find 508-bit $z := x \times y$.

Algorithm 2: Double-and-add method for integer multiplication

Input : x, y — two integers being multiplied
Output : Result of the multiplication $x \times y$

Decompose y to the binary form: $(y_0, y_1, \ldots, y_{N-1})_2$

BigInt Multiplication

Problem

Given two 254-bit integers x and y, find 508-bit $z := x \times y$.

Algorithm 3: Double-and-add method for integer multiplication

Input : x, y — two integers being multiplied

Output : Result of the multiplication $x \times y$

Decompose y to the binary form: $(y_0, y_1, \dots, y_{N-1})_2$

$r \leftarrow 0$

$t \leftarrow x$

BigInt Multiplication

Problem

Given two 254-bit integers x and y, find 508-bit $z := x \times y$.

Algorithm 4: Double-and-add method for integer multiplication

---

Input : x, y — two integers being multiplied
Output : Result of the multiplication $x \times y$

```
Decompose y to the binary form: (y_0, y_1, ..., y_{N-1})_2
r ← 0
t ← x
for i ∈ {0, ..., N − 1} do
    if y_i = 1 then
        r ← r + t
    end
    t ← 2 × t
end
```

BigInt Multiplication

Problem

Given two 254-bit integers x and y, find 508-bit $z := x \times y$.

Algorithm 5: Double-and-add method for integer multiplication

Input : x, y — two integers being multiplied
Output : Result of the multiplication $x \times y$

```
Decompose y to the binary form: (y_0, y_1, ..., y_{N-1})_2
r ← 0
t ← x
for i ∈ {0, ..., N − 1} do
    if y_i = 1 then
        r ← r + t
    end
    t ← 2 × t
end
```

Return : Integer r

Other Primitives to Implement...

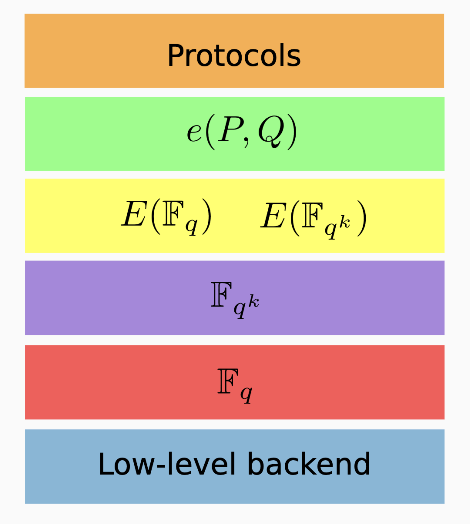

Figure: Primitives to implement

What is the problem then?

If Groth16 is already implemented, what is the reason we are here?

What is the problem then?

If Groth16 is already implemented, what is the reason we are here?

The thing is... Currently, fflonk verification script is **875MB** in size, while Groth16 takes **1.3GB** (after our and Alpen Labs optimization using *w*-window multiplication)... See this post for more details.

The current Bitcoin mainnet restriction is roughly 4MB (while the practical limitation is about 200-400kB). What to do?

Optimizing Big Integer Multiplication on Bitcoin: Introducing w-windowed Approach

Dmytro Zakharov[1], Oleksandr Kurbatov[1], Manish Bista[2] and Belove Bist[2]

[1] Distributed Lab dmytro.zakharov@distributedlab.com, ok@distributedlab.com
[2] Alpen Labs manish@alpenlabs.io, belove@alpenlabs.io

Figure: Our paper on optimizing big integer multiplication

Questions before moving on?

BitVM2

Core Idea

Suppose our script is represented as a function f. Our input (`ScriptSig/witness`) is x, while the output is $y = f(x)$.

Core Idea

Suppose our script is represented as a function f. Our input (`ScriptSig/witness`) is x, while the output is $y = f(x)$.

Note

Although BitVM2's primary goal is implementing the Groth16 verifier (so f is the ZKP verification function), we believe the concept is easily generalizable to any f.

Core Idea

Suppose our script is represented as a function f. Our input (`ScriptSig/witness`) is x, while the output is $y = f(x)$.

Note

Although BitVM2's primary goal is implementing the Groth16 verifier (so f is the ZKP verification function), we believe the concept is easily generalizable to any f.

Idea #1

We do not need to compute y from x. Instead, the **operator** publishes x, y (f is publically known as the part of the protocol), and if $y \neq f(x)$, anyone can punish the operator.

Core Idea

Suppose our script is represented as a function f. Our input (`ScriptSig/witness`) is x, while the output is $y = f(x)$.

Note

Although BitVM2's primary goal is implementing the Groth16 verifier (so f is the ZKP verification function), we believe the concept is easily generalizable to any f.

Idea #1

We do not need to compute y from x. Instead, the **operator** publishes x, y (f is publically known as the part of the protocol), and if $y \neq f(x)$, anyone can punish the operator.

?!

However, doesn't check $y \neq f(x)$ involve calculating f as a whole?

Shards Splitting

Idea #2

We can ease the challenger's burden by **splitting the function** f **into subchunks**. In other words, suppose $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$. Then, the operator can calculate the **intermediate states**:

$$z_1 = f_1(z_0),\ z_2 = f_2(z_1),\ z_3 = f_3(z_2), \dots,\ z_n = f_n(z_{n-1})$$

Where z_0 is x and z_n **must** be y.

Shards Splitting

Idea #2

We can ease the challenger's burden by **splitting the function** f **into subchunks**. In other words, suppose $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$. Then, the operator can calculate the **intermediate states**:

$$z_1 = f_1(z_0),\ z_2 = f_2(z_1),\ z_3 = f_3(z_2), \ldots,\ z_n = f_n(z_{n-1})$$

Where z_0 is x and z_n **must** be y.

Idea #3

If $y \neq f(x)$, that means that for some shard, $z_j \neq f_j(z_{j-1})$.

Shards Splitting

Idea #2

We can ease the challenger's burden by **splitting the function** f **into subchunks**. In other words, suppose $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$. Then, the operator can calculate the **intermediate states**:

$$z_1 = f_1(z_0),\ z_2 = f_2(z_1),\ z_3 = f_3(z_2), \dots,\ z_n = f_n(z_{n-1})$$

Where z_0 is x and z_n **must** be y.

Idea #3

If $y \neq f(x)$, that means that for some shard, $z_j \neq f_j(z_{j-1})$.

Why this is useful?

✓ Disproving $z_j \neq f_j(z_{j-1})$ is **much** easier than $y \neq f(x)$.

Shards Splitting

Idea #2

We can ease the challenger's burden by **splitting the function** f **into subchunks**. In other words, suppose $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$. Then, the operator can calculate the **intermediate states**:

$$z_1 = f_1(z_0),\ z_2 = f_2(z_1),\ z_3 = f_3(z_2), \dots,\ z_n = f_n(z_{n-1})$$

Where z_0 is x and z_n **must** be y.

Idea #3

If $y \neq f(x)$, that means that for some shard, $z_j \neq f_j(z_{j-1})$.

Why this is useful?

- ✓ Disproving $z_j \neq f_j(z_{j-1})$ is **much** easier than $y \neq f(x)$.
- ✓ For stack-based languages, $f_1 \circ f_2 = f_2 \parallel f_1$.

Shards Splitting: Example

Example

Consider a fairly simple program f:

$$f(a, b) = 25a^2 b^2 (a + b)^2$$

Shards Splitting: Example

Example

Consider a fairly simple program f:

$$f(a, b) = 25a^2b^2(a + b)^2$$

Its implementation (assuming OP_MUL is implemented):

Script: $\langle a \rangle$ $\langle b \rangle$ OP_2DUP OP_ADD OP_MUL OP_MUL OP_DUP OP_DUP OP_ADD OP_DUP OP_ADD OP_ADD OP_DUP OP_MUL

Shards Splitting: Example

Example

Consider a fairly simple program f:

$$f(a, b) = 25a^2b^2(a+b)^2$$

Its implementation (assuming `OP_MUL` is implemented):

Script: $\langle a \rangle$ $\langle b \rangle$ `OP_2DUP OP_ADD OP_MUL OP_MUL OP_DUP OP_DUP OP_ADD OP_DUP OP_ADD OP_ADD OP_DUP OP_MUL`

Let us split the function into three shards f_1, f_2, and f_3:

$$f_1(x, y) = xy(x+y), \quad f_2(z) = 5z, \quad f_3(w) = w^2$$

Shards Splitting: Example (cont.)

Example

This way, it is fairly easy to see that $f(a,b) = f_3 \circ f_2 \circ f_1(a,b)$. In turn, in Bitcoin script we can represent f as $f_1 \parallel f_2 \parallel f_3$:

Script:

$\langle a \rangle$ $\langle b \rangle$ `OP_2DUP OP_ADD OP_MUL OP_MUL`	// $xy(x+y)$
`OP_DUP OP_DUP OP_ADD OP_DUP OP_ADD OP_ADD`	// $5z$
`OP_DUP OP_MUL`	// w^2

Shards Splitting: Example (cont.)

Example

This way, it is fairly easy to see that $f(a,b) = f_3 \circ f_2 \circ f_1(a,b)$. In turn, in Bitcoin script we can represent f as $f_1 \parallel f_2 \parallel f_3$:

Script:

```
⟨a⟩ ⟨b⟩ OP_2DUP OP_ADD OP_MUL OP_MUL          // xy(x + y)
OP_DUP OP_DUP OP_ADD OP_DUP OP_ADD OP_ADD     // 5z
OP_DUP OP_MUL                                 // w^2
```

Suppose $a = 2$, $b = 3$. Then, intermediate states are:

$$
\begin{aligned}
z_0 &= (2,3) && \text{// Script input} \\
z_1 &= f_1(z_0) = 2 \times 3 \times (2+3) = 30 \\
z_2 &= f_2(z_1) = 5 \times 30 = 150 \\
z_3 &= f_3(z_2) = 150^2 = 22500 && \text{// Script output}
\end{aligned}
$$

Naive Version

1. Operator splits the program f into shards $f_1, \ldots, f_n$ with intermediate states $z_0, \ldots, z_n$ and commitments $\sigma_0, \ldots, \sigma_n$.
2. Operator creates an **Assert Transaction** that can be spent in $n+1$ different ways (taptree):
   - $((j+1)^{\text{th}})$ **DisproveScript**$[j]$: Challenger shows $z_{j+1} \neq f_j(z_j)$.
   - $((n+1)^{\text{th}})$ **Payout**: `LockTimeVerify` + `CheckSig`.

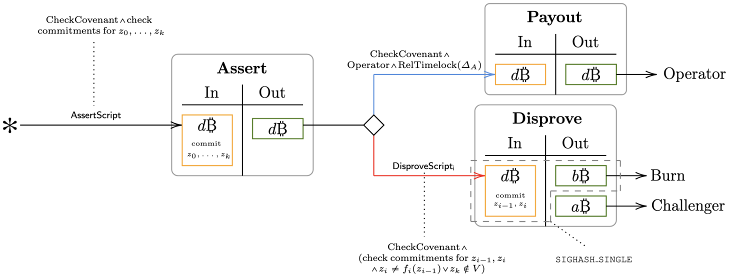

Figure: BitVM2 Naive Version from the original paper

"Super-Optimistic" Version

Operator creates a **Claim Tx** with commitments, and Challenger publishes the **Challenge Tx** in case of suspicion. Rest is the same.

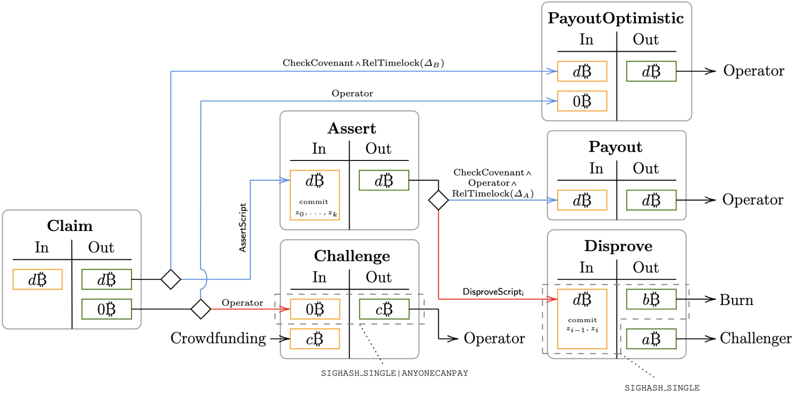

Figure: BitVM2 Optimized Version from the original paper

BitVM2 Pitfalls

Main Problem

How do we implement the `DisproveScript`?

Main Problem

How do we implement the `DisproveScript`?

Script:

$\langle z_{j-1} \rangle$ `OP_DUP` $\langle \sigma_{j-1} \rangle$ $\langle \mathsf{pk}_{j-1} \rangle$ `OP_WINTERNITZVERIFY`

$\langle z_j \rangle$ `OP_DUP` $\langle \sigma_j \rangle$ $\langle \mathsf{pk}_j \rangle$ `OP_WINTERNITZVERIFY`

$\langle f_j \rangle$ `OP_EQUAL` `OP_NOT`

Main Problem

How do we implement the `DisproveScript`?

Script:

$\langle z_{j-1} \rangle$ `OP_DUP` $\langle \sigma_{j-1} \rangle$ $\langle \mathsf{pk}_{j-1} \rangle$ `OP_WINTERNITZVERIFY`

$\langle z_j \rangle$ `OP_DUP` $\langle \sigma_j \rangle$ $\langle \mathsf{pk}_j \rangle$ `OP_WINTERNITZVERIFY`

$\langle f_j \rangle$ `OP_EQUAL` `OP_NOT`

pk's and z's are stored in the `scriptPubKey`, while σ's (Winternitz signatures) are provided by the challenger in the `witness`.

Main Problem

How do we implement the `DisproveScript`?

Script:

$\langle z_{j-1}\rangle$ `OP_DUP` $\langle \sigma_{j-1}\rangle$ $\langle \mathsf{pk}_{j-1}\rangle$ `OP_WINTERNITZVERIFY`

$\langle z_j\rangle$ `OP_DUP` $\langle \sigma_j\rangle$ $\langle \mathsf{pk}_j\rangle$ `OP_WINTERNITZVERIFY`

$\langle f_j\rangle$ `OP_EQUAL` `OP_NOT`

pk's and z's are stored in the `scriptPubKey`, while σ's (Winternitz signatures) are provided by the challenger in the `witness`.

Main Problem

1. Each z_j is a collection of `u32` elements.

Main Problem

How do we implement the `DisproveScript`?

Script:

$\langle z_{j-1} \rangle$ `OP_DUP` $\langle \sigma_{j-1} \rangle$ $\langle \mathsf{pk}_{j-1} \rangle$ `OP_WINTERNITZVERIFY`
$\langle z_j \rangle$ `OP_DUP` $\langle \sigma_j \rangle$ $\langle \mathsf{pk}_j \rangle$ `OP_WINTERNITZVERIFY`
$\langle f_j \rangle$ `OP_EQUAL` `OP_NOT`

pk's and z's are stored in the `scriptPubKey`, while σ's (Winternitz signatures) are provided by the challenger in the `witness`.

Main Problem

1. Each z_j is a collection of `u32` elements.
2. This collection cannot be aggregated (e.g., $H(z_{j,1} \parallel z_{j,2} \parallel \ldots)$).

Main Problem

How do we implement the `DisproveScript`?

Script:

$\langle z_{j-1} \rangle$ `OP_DUP` $\langle \sigma_{j-1} \rangle$ $\langle \mathrm{pk}_{j-1} \rangle$ `OP_WINTERNITZVERIFY`
$\langle z_j \rangle$ `OP_DUP` $\langle \sigma_j \rangle$ $\langle \mathrm{pk}_j \rangle$ `OP_WINTERNITZVERIFY`
$\langle f_j \rangle$ `OP_EQUAL` `OP_NOT`

pk's and z's are stored in the `scriptPubKey`, while σ's (Winternitz signatures) are provided by the challenger in the `witness`.

Main Problem

1. Each z_j is a collection of `u32` elements.
2. This collection cannot be aggregated (e.g., $H(z_{j,1} \parallel z_{j,2} \parallel \dots)$).
3. Thus, every stack element must be signed separately.

Main Problem

How do we implement the `DisproveScript`?

Script:

$\langle z_{j-1} \rangle$ `OP_DUP` $\langle \sigma_{j-1} \rangle$ $\langle \mathsf{pk}_{j-1} \rangle$ `OP_WINTERNITZVERIFY`
$\langle z_j \rangle$ `OP_DUP` $\langle \sigma_j \rangle$ $\langle \mathsf{pk}_j \rangle$ `OP_WINTERNITZVERIFY`
$\langle f_j \rangle$ `OP_EQUAL` `OP_NOT`

pk's and z's are stored in the `scriptPubKey`, while σ's (Winternitz signatures) are provided by the challenger in the `witness`.

Main Problem

1. Each z_j is a collection of `u32` elements.
2. This collection cannot be aggregated (e.g., $H(z_{j,1} \parallel z_{j,2} \parallel \dots)$).
3. Thus, every stack element must be signed separately.
4. Signing each element costs roughly 1kB (!!!)

Splitting

Now, how do we actually implement splitting?

Splitting

Now, how do we actually implement splitting?

Idea #1

Fix shard size L. Take the first L opcodes. If not all `OP_IF`s are closed, add opcodes till they are closed. Repeat until the end.

Splitting

Now, how do we actually implement splitting?

Idea #1

Fix shard size L. Take the first L opcodes. If not all `OP_IF`s are closed, add opcodes till they are closed. Repeat until the end.

Problem: Although we might make all shards of size $\approx L$, the intermediate state sizes can still be large.

Splitting

Now, how do we actually implement splitting?

Idea #1

Fix shard size L. Take the first L opcodes. If not all `OP_IF`s are closed, add opcodes till they are closed. Repeat until the end.

Problem: Although we might make all shards of size $\approx L$, the intermediate state sizes can still be large.

Example

u32 multiplication costs roughly 4.5kB in Bitcoin Script. Splitting:

Shard number	Shard Size	# Elements in state	Estimated Cost
1	623B	37	37kB
2	640B	32	32kB
3	640B	27	27kB
4	640B	22	22kB
5	640B	17	17kB
6	640B	12	12kB
7	627B	3	3kB

Ideology

Core Idea

1. Making the taptree larger does not cost almost anything. Therefore, we might make shards as small as we want them to be.

Ideology

Core Idea

1. Making the taptree larger does not cost almost anything. Therefore, we might make shards as small as we want them to be.
2. We should care not only for making shards small but, more importantly, intermediate state sizes smaller.

Ideology

Core Idea

1. Making the taptree larger does not cost almost anything. Therefore, we might make shards as small as we want them to be.
2. We should care not only for making shards small but, more importantly, intermediate state sizes smaller.
3. ... which is impossible to do automatically; only **manually**.

Ideology

Core Idea

1. Making the taptree larger does not cost almost anything. Therefore, we might make shards as small as we want them to be.
2. We should care not only for making shards small but, more importantly, intermediate state sizes smaller.
3. ... which is impossible to do automatically; only **manually**.

Definition

A function f is called **BitVM-friendly** if:

- It can be split into the shards $f_1, \ldots, f_n$ of relatively small size.

Ideology

Core Idea

1. Making the taptree larger does not cost almost anything. Therefore, we might make shards as small as we want them to be.
2. We should care not only for making shards small but, more importantly, intermediate state sizes smaller.
3. ... which is impossible to do automatically; only **manually**.

Definition

A function f is called **BitVM-friendly** if:

- It can be split into the shards $f_1, \dots, f_n$ of relatively small size.
- The intermediate states $\{z_j\}_{0 \leq j \leq n}$ contain a small number of elements, making the commitment cheap enough.

Square Fibonacci Sequence

Let us consider one non-trivial BitVM-friendly script.

Square Fibonacci Sequence

Let us consider one non-trivial BitVM-friendly script.

Problem Statement

Fix integer q and two integers x_0, x_1. Define the sequence

$$x_{j+2} = x_{j+1}^2 + x_j^2 \pmod{q}$$

Square Fibonacci Sequence

Let us consider one non-trivial BitVM-friendly script.

Problem Statement

Fix integer q and two integers x_0, x_1. Define the sequence

$$x_{j+2} = x_{j+1}^2 + x_j^2 \pmod{q}$$

Define $f(a, b)$ to be x_{1000} with $x_0 = a, x_1 = b$.

Square Fibonacci Sequence

Let us consider one non-trivial BitVM-friendly script.

Problem Statement

Fix integer q and two integers x_0, x_1. Define the sequence

$$x_{j+2} = x_{j+1}^2 + x_j^2 \pmod q$$

Define $f(a, b)$ to be x_{1000} with $x_0 = a, x_1 = b$.

Observe that having (x_j, x_{j+1}), it is easy to get (x_{j+1}, x_{j+2}):

Square Fibonacci Sequence

Let us consider one non-trivial BitVM-friendly script.

Problem Statement

Fix integer q and two integers x_0, x_1. Define the sequence

$$x_{j+2} = x_{j+1}^2 + x_j^2 \pmod q$$

Define $f(a, b)$ to be x_{1000} with $x_0 = a, x_1 = b$.

Observe that having (x_j, x_{j+1}), it is easy to get (x_{j+1}, x_{j+2}):

Script: `OP_DUP OP_SQUARE` $\langle 2 \rangle$ `OP_ROLL OP_SQUARE OP_ADD`

Square Fibonacci Sequence (cont.)

Total script:

Script:

repeat 1000 times
```
    OP_DUP OP_SQUARE ⟨2⟩ OP_ROLL OP_SQUARE OP_ADD
```
end
```
OP_SWAP OP_DROP
```

Square Fibonacci Sequence (cont.)

Total script:

Script:

repeat 1000 times

  OP_DUP OP_SQUARE ⟨2⟩ OP_ROLL OP_SQUARE OP_ADD

end

OP_SWAP OP_DROP

Is it BitVM-friendly? Yes! Make 1001 shards:

- **Shards 1 . . . 1000:** $\{$ OP_DUP OP_SQUARE ⟨2⟩ OP_ROLL OP_SQUARE OP_ADD $\}$.
- **Shard 1001:** $\{$ OP_SWAP OP_DROP $\}$

Square Fibonacci Sequence (cont.)

Total script:

Script:

```
repeat 1000 times
    OP_DUP OP_SQUARE ⟨2⟩ OP_ROLL OP_SQUARE OP_ADD
end
OP_SWAP OP_DROP
```

Is it BitVM-friendly? Yes! Make 1001 shards:

- **Shards 1 . . . 1000:** { OP_DUP OP_SQUARE ⟨2⟩ OP_ROLL OP_SQUARE OP_ADD } .
- **Shard 1001:** { OP_SWAP OP_DROP }

Intermediate State Size: 2 integers.

Square Fibonacci Sequence (cont.)

Total script:

Script:

```
repeat 1000 times
    OP_DUP OP_SQUARE ⟨2⟩ OP_ROLL OP_SQUARE OP_ADD
end
OP_SWAP OP_DROP
```

Is it BitVM-friendly? Yes! Make 1001 shards:

- **Shards $1 \ldots 1000$:** $\{$ OP_DUP OP_SQUARE ⟨2⟩ OP_ROLL OP_SQUARE OP_ADD $\}$.
- **Shard 1001:** $\{$ OP_SWAP OP_DROP $\}$

Intermediate State Size: 2 integers.

Question: What if we wanted to compute the 1000000^{th} element?

Big Integer Multiplication

Algorithm 6: Double-and-add method for integer multiplication

---

Input : x, y — two integers being multiplied
Output : Result of the multiplication $x \times y$

```
Decompose y to the binary form: (y_0, y_1, ..., k_{N-1})_2
r ← 0
t ← x
for i ∈ {0, ..., N − 1} do
    if y_i = 1 then
        r ← r + t
    end
    t ← 2 × t
end
```

Return : Integer r

Question: Suppose we use the automatic splitting. Would that be BitVM-friendly?

Big Integer Multiplication (cont.)

Suppose for concreteness that we multiply two 254-bit integers.

Big Integer Multiplication (cont.)

Suppose for concreteness that we multiply two 254-bit integers.

At each for-loop step, we need to store the binary decomposition of one integer, which consists of 254 elements.

Big Integer Multiplication (cont.)

Suppose for concreteness that we multiply two 254-bit integers.

At each for-loop step, we need to store the binary decomposition of one integer, which consists of 254 elements.

We need to sign (commit to) each one. Meaning we have **254kB** for any shard **at least**. Although the multiplication algorithm itself costs roughly 100kB.

Big Integer Multiplication (cont.)

Suppose for concreteness that we multiply two 254-bit integers.

At each for-loop step, we need to store the binary decomposition of one integer, which consists of 254 elements.

We need to sign (commit to) each one. Meaning we have **254kB** for any shard **at least**. Although the multiplication algorithm itself costs roughly 100kB.

How this can be fixed?

BitVM-friendly Big Integer Multiplication

Algorithm 7: BitVM-friendly double-and-add method

Input : x, y — two u32 integers being multiplied, N — bitsize of y.
Output : Result of the multiplication $x \times y$

$r \leftarrow 0$
$t \leftarrow x$
for $i \in \{0, \ldots, N\}$ **do**
    Start the shard i
    Decompose y into the binary form: $y = (y_0, \ldots, y_{N-1})_2$
    if $y_i = 1$ **then**
        $r \leftarrow r + t$
    end
    $t \leftarrow 2 \times t$
    Recover y back to the original form: $y \leftarrow \sum_{i=0}^{N-1} y_i 2^i$.
    End shard i
end
Return : Integer r

Thank you for your attention

♥

distributedlab.com

github.com/distributed-lab/nero