Binohash: Transaction Introspection Without Softforks

Robin Linus
ZeroSync, Stanford University

Abstract

We present *Binohash*, a collision-resistant hash function for Bitcoin Script that enables limited transaction introspection without consensus changes. By exploiting the `FindAndDelete` quirk in legacy `OP_CHECKMULTISIG` combined with proof-of-work signature grinding, Binohash creates a transaction digest directly readable in Script. This enables covenant-like functionality for trustless chain introspection for protocols like BitVM bridges.

Our scheme achieves $W_1 + W_2$ bits of collision resistance with $\approx W_2 + 2.6$ bits of honest work, using a two-round nonce extraction protocol. With $W_1 = W_2 = 42$ bits, this provides $\sim$84 bits collision resistance with $\sim$44.6 bits honest work, with grinding cost of less than $50 on modern cloud GPUs.

1 Introduction

1.1 The Problem

Bitcoin Script cannot natively read transaction data (inputs, outputs, amounts, etc.). This limitation prevents trustless verification of transaction properties within Script, forcing protocols like BitVM bridges[1][2][3] to rely on trusted oracles or accept weaker security models.

1.2 The Goal

Create a collision-resistant hash function for Bitcoin transactions (which we call *Binohash*, named after the binomial coefficient $\binom{n}{t}$ central to its construction) that produces a digest with the following properties:

1. **Readable in Script** - Can be extracted and verified within Bitcoin Script
2. **Collision-resistant** - Computationally infeasible to find two transactions with the same Binohash
3. **Lamport-signable** - Can be Lamport-signed to import into BitVM verification

Use case: BitVM bridges need to verify transaction properties (e.g., *"did this peg-out transaction actually happen?"* or *"did the rollup sequencer actually publish the state diff?"*). By Lamport-signing the Binohash in Script, we can feed it into BitVM's off-chain verification without trusted oracles (i.e., without BitVM light clients that require existential honesty assumptions for safety).

1.3 Threat Model

The primary security goal of our scheme is to prevent an attacker from finding two different transactions txA and txB with the same Binohash such that they can execute txA onchain but feed txB into BitVM.

1.3.1 Attack Scenario

In BitVM bridge applications, the Binohash serves as a commitment to a specific transaction included in the blockchain. The threat model considers an adversary who attempts to:

1. Find a collision: create two different transactions txA and txB with the same Binohash
2. Execute txA onchain (the actual blockchain transaction)
3. Submit txB to BitVM's off-chain verification to prove false properties about the onchain execution

Impact: If successful, the attacker could prove properties about txB to BitVM (e.g., "the peg-out paid address X with amount Y") while actually executing txA onchain with different properties (e.g., paying a different address or amount). This breaks the connection between on-chain execution and off-chain verification.

1.4 Our Approach: Binohash

Since Script cannot directly compute transaction hashes (no `OP_TXHASH`), Binohash derives a transaction digest indirectly:

1. **Pin transaction fields** - Make TX modification expensive so the Binohash commits to the TX
2. **Derive a Binohash** - Find a nonce satisfying a PoW puzzle; this nonce becomes the Binohash

Key insight: By embedding signatures in the locking script and using `OP_CHECKMULTISIG`'s `FindAndDelete` behavior, different subset selections create different sighashes. The spender grinds through subsets until finding one that satisfies a puzzle constraint. The subset indices that worked become the Binohash.

1.5 Results

Binohash achieves:

- $\sim W_2 + 2.6$ **bits honest work** for honest spenders (4 pinning + 2 nonce rounds)
- $W_1 + W_2$ **bits collision resistance**
- **Binohash readable in Script** via subset indices in unlocking script
- **Consensus-valid** using only legacy Script opcodes

Concrete parameters: With $W_1 = W_2 = 42$ bits of work per signature:

- **$\sim$84 bits collision resistance**
- **$\sim$44.6 bits honest work** (feasible with GPUs)

1.6 Related Work

"Signing a Bitcoin Transaction with Lamport Signatures"[4]: Tries to achieve 45 bits collision resistance—insufficient for high-security applications.

ColliderScript/ColliderVM[5, 6]: Enable covenants via hash collisions but require impractical computational work (∼$50M electricity cost).

Our contribution: Practical transaction introspection with ∼83 bits collision resistance and ∼43.6 bits honest work.

2 Background & Primitives

Note: If you are familiar with the intricacies of Bitcoin Script and ECDSA DER encoding, you can skip this section.

Variable-length signatures have been explored extensively in prior Bitcoin research as a mechanism for introducing novel features[7, 8, 9, 10, 11].

2.1 Signature Puzzles

A signature puzzle is a proof-of-work constraint on an ECDSA signature that can be verified in Bitcoin Script. We exploit the variable size of ECDSA signatures to create verifiable PoW.

2.1.1 The ECDSA Formula

An ECDSA signature consists of a pair (R, s), where R is a curve point and s is a scalar.

The s-value is computed as:

$$s = k^{-1}(z + d \cdot r) \tag{1}$$

where:

- k is the private signing nonce
- z is the sighash (the hash of the message being signed. Bitcoin uses double SHA-256: $z = \text{SHA-256}(\text{SHA-256}(\text{message}))$)
- d is the private key
- r is the x-coordinate of $R = kG$, the public nonce

2.1.2 DER Encoding and Signature Size

Bitcoin ECDSA signatures use DER encoding and have variable size.

Format for Bitcoin ECDSA signatures:

```
0x30 [total-length] 0x02 [r-length] [r] 0x02 [s-length] [s] [sighash-flag]
```

Where:

- `0x30` = SEQUENCE tag (1 byte)

- `total-length` = length of remaining data (1 byte)
- `0x02` = INTEGER tag for r (1 byte)
- `r-length` = length of r value (1 byte)
- `[r]` = the r value (rlen bytes)
- `0x02` = INTEGER tag for s (1 byte)
- `s-length` = length of s value (1 byte)
- `[s]` = the s value (slen bytes)
- `[sighash-flag]` = the sighash flag (1 byte)

Size formula:

$$\text{Bitcoin signature size} = 7 + \text{rlen} + \text{slen} \tag{2}$$

The 7-byte overhead comes from the DER structure tags (0x30, 0x02, 0x02), length fields, and the sighash flag.

Non-negative encoding: Strict DER encoding[12] requires integers to be non-negative. If the most significant bit of r or s is 1, DER encoding prepends a zero-byte to that value, increasing its length by 1 byte.

Signature size range: The smallest possible Bitcoin ECDSA signature is 9 bytes. The largest is 74 bytes.

Key insight: Since signature size depends on the byte lengths of r and s, we can create a proof-of-work by grinding for signatures with small values, which can be verified in Script using `OP_SIZE`.

2.1.3 Smallest Possible R Value

The "smallest" curve point on secp256k1 with a known discrete logarithm is $R_{\min} = 2^{-1}G$ (the point corresponding to private key 2^{-1} modulo the curve order)[13]. Its x-coordinate is:

```
r_min = 0x3b78ce563f89a0ed9414f5aa28ad0d96d6795f9c63
```

This value is only 21 bytes when DER-encoded as an R value in a signature. Grinding for a smaller R value would require finding a point with an x-coordinate smaller than $r_{\min}$, which takes approximately 12 bytes = 96 bits of work (actually 97 bits, because if the first bit is 1, DER encoding prepends a zero byte, keeping the size at 21 bytes).

Use in signature puzzles: The spender is computationally constrained to use $R_{\min}$ as the nonce to achieve sufficiently small signatures. Since $R_{\min}$ has the smallest known x-coordinate (21 bytes), finding any other nonce with a comparably small R value is computationally infeasible. With this 21-byte R value, the signature size is $7 + 21 + 32 = 60$ bytes (if s is a regular 32 byte value), though typical grinding targets are 50–60 bytes depending on the proof-of-work requirement.

The signature equation then becomes

$$s = k^{-1}(z + d \cdot r) = 2(z + d \cdot r_{\min})$$

We define public key $P_{\min}$ and corresponding private key $d_{\min}$ such that $d_{\min} = r_{\min}^{-1}$, which reduces the signature formula to:

$$s = 2(z + d_{\min} \cdot r_{\min}) = 2(z + 1)$$

Thus, s becomes simply the sighash $z + 1$ shifted by one bit. This reduces the question of the signature being small to the sighash being small. Hence, no curve point arithmetic is required for grinding – only hashing.

2.1.4 Script Verification

Script can verify signature size using `OP_SIZE`:

```
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey> OP_CHECKSIGVERIFY
```

This checks that the signature is exactly $(60 - n)$ bytes, proving the signer did $W = 8n + 2$ bits of work.

Granularity: Since `OP_SIZE` returns the byte-size of stack elements, the proof-of-work has 8-bit granularity. For arbitrary-precision proof-of-work beyond byte boundaries, see [10].

Work parameter: For W bits of work, we need $n = \lceil (W - 2)/8 \rceil$ bytes of size reduction.

Derivation of $W = 8n + 2$: Requiring a signature of exactly $(60 - n)$ bytes means reducing the signature size by n bytes below the maximum 60-byte threshold. Since we use fixed $R_{\min}$ (21 bytes for r), the size reduction must come entirely from s. For s to encode in fewer bytes, it must have leading zero bits in its binary representation. Each byte saved requires 8 additional leading zero bits in the sighash z, plus 1 bit to ensure the higest non-zero byte starts with a zero (avoiding DER sign-bit padding), plus 1 additional bit to account for the bitshift of z in $s = 2(z+1)$ (i.e. the multiplication by 2). Thus, $W = 8n + 2$ bits of grinding work are required.

Examples:

- $W = 42$ bits ($n = 5$): verified with `OP_SIZE 55 OP_EQUALVERIFY`
- $W = 50$ bits ($n = 6$): verified with `OP_SIZE 54 OP_EQUALVERIFY`

2.2 ECDSA Key Recovery

ECDSA has a key recovery property: given a signature and message, we can recover the public key that would verify this signature.

How it works: Given (R, s) and message z, we can compute the public key P that satisfies the ECDSA verification equation. This allows creating arbitrary signature data and deriving the corresponding public key.

2.3 Minimum-Size Signatures

The minimum valid DER-encoded ECDSA signature is 9 bytes.

Structure of a 9-byte signature:

```
0x30 0x07 0x02 0x01 [r] 0x02 0x01 [s] [sighash-flag]
```

This corresponds to:

- Both r and s are 1 byte each (the smallest possible)
- With the 7-byte overhead, the total is 9 bytes

The values of r and s must be < 128 (MSB $= 0$). If either r or s has MSB $= 1$ (value ≥ 128), DER non-negative encoding would prepend a zero byte, making it 2 bytes instead of 1 byte, and the signature would be larger than 9 bytes.

Key property: Any 9-byte value that follows valid DER structure can be treated as a signature. We can then use ECDSA key recovery (from §2.2) to find a public key that would make this signature valid for a chosen message z.

2.4 SIGHASH Modes

Bitcoin has 6 sighash modes affecting what transaction data is hashed: `ALL`, `NONE`, `SINGLE`, `ANYONECANPAY|ALL`, `ANYONECANPAY|NONE`, `ANYONECANPAY|SINGLE`. Each mode commits to different transaction fields.

For legacy and SegWit v0 transactions, Bitcoin uses double SHA-256 for signature hashes.

2.4.1 The 6 Sighash Modes

Mode	Flag	Commits To	Omits
`ALL`	`0x01`	`nVersion`, all `prevout`, all `nSequence`, all outputs, `nLocktime`	Nothing (most restrictive)
`NONE`	`0x02`	`nVersion`, all `prevout`, current input's `nSequence`, `nLocktime`	All outputs; sets other inputs' `nSequence` to 0
`SINGLE`	`0x03`	`nVersion`, all `prevout`, current input's `nSequence`, one output (same index), `nLocktime`	Other outputs; sets other inputs' `nSequence` to 0
`ANYONECANPAY|ALL`	`0x81`	`nVersion`, one input `prevout`, one input `nSequence`, all outputs, `nLocktime`	Other inputs
`ANYONECANPAY|NONE`	`0x82`	`nVersion`, one input `prevout`, one input `nSequence`, `nLocktime`	All outputs, other inputs
`ANYONECANPAY|SINGLE`	`0x83`	`nVersion`, one input `prevout`, one input `nSequence`, one output, `nLocktime`	Other inputs, other outputs

2.4.2 The Sighash Flag Byte

There are 256 possible sighash flag byte values (0x00-0xFF). Only 6 are standard (`STRICTENC`) , but all 256 are consensus-valid and produce different sighashes (the flag byte is included in the sighash preimage).

Flag distribution by mode:

Mode	Standard Flag	Byte Values
ALL	0x01	120
NONE	0x02	4
SINGLE	0x03	4
ALL|ANYONECANPAY	0x81	120
NONE|ANYONECANPAY	0x82	4
SINGLE|ANYONECANPAY	0x83	4
Total	**6 standard**	**256 total**

Note: All 256 byte values (0x00-0xFF) are consensus-valid; only 6 are standard

2.4.3 The SIGHASH_SINGLE Bug

In pre-SegWit scripts, if `input_idx` $\geq$ `num_outputs` when using `SIGHASH_SINGLE`, the sighash computation returns the constant value `0x0000...0001` ($z = 1$) instead of computing the actual hash [14].

Use case 1: Excludes SIGHASH_SINGLE modes from puzzle grinding

We force the spender to trigger the bug by embedding a hardcoded signature check:

Construction:

1. Generate an arbitrary 9-byte DER signature with `SIGHASH_SINGLE` flag (from §2.3)
2. Use ECDSA key recovery (from §2.2) with $z = 1$ to compute the corresponding public key
3. Embed both in the locking script:

```
<minimum 9-byte signature>
<recovered pubkey for z=1> OP_CHECKSIGVERIFY
```

To pass this check, the spender must trigger the bug (`input_idx` $\geq$ `num_outputs`). When triggered, all `SIGHASH_SINGLE` signatures on that input have $z = 1$ (fixed).

Algebraic reasoning why SIGHASH_SINGLE becomes useless for grinding:

Recall from §2.3 that puzzle signatures use public key $P = r_{\text{min}}^{-1}G$ where r_{min} is the x-coordinate of $R_{\text{min}} = 2^{-1}G$. For any signature (R, s) with $z = 1$ and $R = kG$:

$$s = k^{-1}(1 + r_{\text{min}}^{-1} \cdot r) \tag{3}$$

where r is the x-coordinate of R.

Why grinding fails: To achieve a small signature size, the attacker would need both r and k^{-1} to be small. However, grinding a nonce k such that both r (the x-coordinate of kG) and k^{-1} are simultaneously small would require at least 90 bits of grinding work - far exceeding the work budget of the entire scheme.

No grinding freedom: With $z = 1$ fixed, varying the transaction or `scriptCode` doesn't change z. The core grinding mechanism is broken for `SIGHASH_SINGLE`.

The key insight: The spender is effectively limited to 4 modes for grinding: `ALL`, `NONE`, `ANYONECANPAY|ALL`, `ANYONECANPAY|NONE`.

Use case 2: Enables transaction-independent minimum-size signatures

The bug allows creating 9-byte signatures that are valid for any transaction (regardless of content) as long as `input_idx` $\geq$ `num_outputs`. We can hardcode such signatures as constants in scripts and use them later (§3.2) to create nonce pools for `FindAndDelete` grinding.

2.5 The `scriptCode` in Bitcoin signature hashing

In Bitcoin signature hashing, the `scriptCode` is the script fragment that a `CHECKSIG` or `CHECKMULTISIG` signature commits to (roughly: the part of the spending conditions being validated, starting after the most recently executed `OP_CODESEPARATOR`). In legacy (pre-SegWit) hashing, `FindAndDelete` removes the pushed signature bytes from this script before hashing (to avoid "signing itself"), whereas SegWit v0 (BIP-143[15]) defines `scriptCode` directly from the witness/redeem script with `OP_CODESEPARATOR` semantics and explicitly drops `FindAndDelete`.

2.6 `FindAndDelete` in `OP_CHECKMULTISIG`

`FindAndDelete` is a quirk in Bitcoin's `OP_CHECKMULTISIG` implementation that removes signature data from the `scriptCode` before computing sighashes. This behavior creates a mechanism for controllable sighash variation that is central to our scheme.

2.6.1 What is `FindAndDelete`?

When `OP_CHECKMULTISIG` verifies signatures, it performs the following for each signature check:

1. **Remove all signatures from `scriptCode`**: Before computing the sighash, Bitcoin runs a "find and delete" pass that removes *all pushed signature subscripts* (push opcode + signature bytes) provided to this `OP_CHECKMULTISIG` from the `scriptCode`. This means when verifying signature i, all signatures (not just signature i) are removed from `scriptCode`.
2. **Compute sighash with modified `scriptCode`**: The sighash is computed using this modified `scriptCode`. Finally, the signature is verified for this computed sighash

2.6.2 Historical Context

`FindAndDelete` was originally intended as a cleanup mechanism to prevent signatures from signing themselves. However, it was implemented in a way that has practical consequences: different signatures embedded in the locking script lead to different `scriptCodes`, which produce different sighashes.

This behavior has no known legitimate use case in standard Bitcoin scripts and is generally considered a historical artifact. However, we exploit it deliberately to create more space for grinding sighashes.

2.6.3 Future Consensus Changes

The original 2019 Great Consensus Cleanup proposal [16] by Matt Corallo suggested making `FindAndDelete` consensus-invalid. However, the current Great Consensus Cleanup (BIP-54) [17] leaves `FindAndDelete` and `OP_CODESEPARATOR` untouched at the consensus level. Instead, BIP-54 introduces a 2500 per-transaction limit on potentially-executed legacy sigops to address confiscation concerns, while keeping these features non-standard but consensus-valid.

2.7 Technical Constraints

The scheme operates under several constraints imposed by Bitcoin's script system:

Script type requirements:

- **ECDSA signatures:** Only available pre-Taproot
- **SIGHASH_SINGLE bug:** Only works in legacy scripts (pre-SegWit)
- **P2SH limitation:** Redeem scripts limited to 520 bytes, forcing bare script usage
- **Bare scripts:** Non-standard but consensus-valid, allow up to 10KB total size

Script limits:

- **201 non-push opcode limit**. Key pushes for `OP_CHECKMULTISIG` count as non-push opcodes
- **10KB maximum** for consensus-valid scripts

Standardness:

- Violates `SCRIPT_VERIFY_CONST_SCRIPTCODE` flag (`FindAndDelete` changes `scriptCode`)
- Violates `SCRIPT_VERIFY_CLEANSTACK` flag enforcing that the final stack must contain only a single true value. This is a standardness rule for legacy script. Our scheme pushes many elements (signatures, hashes) and leaves multiple items on the stack. Cleaning the stack would require many drop operations, pushing us over the 201 opcode limit.

The transaction cannot enter mempool on regular nodes and must be mined via `generateblock` RPC or modified nodes accepting non-standard transactions beyond the `acceptnonstdtxn` config flag.

3 Primitives

To achieve our goal of a collision-resistant, readable Binohash, we combine two primitives built from signature puzzles (as described in §2.1):

1. **Transaction Pinning** - Signature puzzles that bind the Binohash to the transaction
2. `FindAndDelete` **Nonce Space** - Signature puzzles that provide a digest that is readable in Script

3.1 Transaction Pinning

Definition: A technique to make transaction modification computationally expensive by requiring multiple signature puzzles, each with W_1 bits of work.

Construction:

```
//
// Unlocking Script
//
<sig_4> <sig_3> <sig_2> <sig_1>

//
// Locking Script
//

// Exclude SIGHASH_SINGLE using the bug
<sig_SINGLE> <pubkey_SINGLE> OP_CHECKSIGVERIFY

// Require 4 signature puzzles, each with W_1 bits of work
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_1> OP_CHECKSIGVERIFY
OP_CODESEPARATOR
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_2> OP_CHECKSIGVERIFY
OP_CODESEPARATOR
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_3> OP_CHECKSIGVERIFY
OP_CODESEPARATOR
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_4> OP_CHECKSIGVERIFY
```

Where $n = \lceil (W_1 - 2)/8 \rceil$ bytes and $W_1 = 8n + 2$ bits of work per signature.

Optimized version: An optimized 13-opcode construction is provided in Appendix A.

Economics:

- Honest spender: 4 signature puzzles, each requiring W_1 bits of work (using 4 different sighash modes)
- Attacker: Must redo ≥ 1 signature puzzle $= W_1$ bits of work when modifying the TX

Why different modes: Grinding two signatures for the same sighash mode requires $2 \cdot W_1$ bits of work, so spenders must use different modes – assuming $2W_1$ operations is intractable.

Key property: Removes all transaction-based degrees of freedom, forcing attackers to use only the `FindAndDelete` nonce space in later stages.

Concrete parameters: $W_1 = 42$ bits ($n = 5$ bytes) $\rightarrow$ each signature requires 42 bits, 4 total $= \sim$44 bits work.

3.1.1 How to Grind

Grinding is forced to be in the following order using the following fields. Otherwise, the proofs of work invalidate each other at some point in the process:

1. `NONE|ANYONECANPAY`: `nVersion`, `nSequence` (highest bit must be 1 to disable relative locktimes), `nLocktime` (must be lower than blocktime/blockheight)
2. `NONE`: the other inputs (grind an ancestor transaction's TXID here. Alternatively, iterate through permutations of multiple inputs)
3. `ALL|ANYONECANPAY`: `OP_RETURN` output data
4. `ALL`: the `nSequence` values of the other inputs (note: `NONE` sets all other inputs' `nSequence` to 0, so this mode must come after `NONE`)

3.2 FindAndDelete Nonce Space

Definition: A mechanism to create controllable sighash variation by embedding signatures in the locking script and selecting subsets.

Construction:

1. Embed n signatures in the locking script (the "nonce pool")
2. Spender provides t indices: $[i_1, i_2, \ldots, i_t]$
3. Script uses `OP_ROLL` to select those specific signatures
4. `OP_CHECKMULTISIG` processes them
5. `FindAndDelete` removes the selected signatures from `scriptCode`
6. Different subsets $\rightarrow$ different `scriptCode` $\rightarrow$ different sighash

Nonce space size: $\binom{n}{t}$ possible subsets $\rightarrow \log_2 \binom{n}{t}$ bits of entropy

Example: $n = 120$, $t = 8 \rightarrow \binom{120}{8} \approx 2^{40}$ possible sighashes

Key property: The subset indices are directly readable from the unlocking script, making them suitable as a Binohash (transaction digest).

Optimization: Use ECDSA key recovery (§2.2) and minimum-size signatures (§2.3) with the `SIGHASH_SINGLE` bug (§2.4.3) to create 9-byte transaction-independent signatures that can be hardcoded in the locking script. This reduces size from ~73 bytes to 9 bytes per signature, allowing more signatures to fit within script size limits.

4 Single Round Scheme

4.1 Overview

Two stages:

1. **Stage 1:** Transaction Pinning (4 signature puzzles $\times W_1$ bits each)
2. **Stage 2:** Nonce Extraction (1 `FindAndDelete` round $\times W_2$ bits)

4.2 Stage 1: Transaction Pinning

Uses Primitive 1 to pin the transaction with W_1 bits of work per signature.

Work: $\log_2(4 \times 2^{W_1}) = W_1 + 2$ bits total

4.3 Stage 2: Nonce Extraction (Single Round)

Uses Primitive 2 (`FindAndDelete` nonce space) in combination with a signature puzzle.

Setup:

- Embed n signatures in locking script (nonce pool)

- Puzzle signature requires W_2 bits of work
- Choose n, t such that $\log_2 \binom{n}{t} \approx W_2$ (matching the puzzle difficulty)

Spender's process:

1. Try different subsets of size t
2. For each subset:
   - Compute which signatures will be deleted by `FindAndDelete`
   - Compute resulting sighash
   - Try to create a puzzle signature satisfying the W_2-bit constraint
3. Iterate through subsets until finding one where puzzle signature can be created

Output: The t indices that worked become the Binohash

Work: W_2 bits

Example parameters: $n = 120$, $t = 8 \rightarrow \binom{120}{8} \approx 2^{39.6} \approx W_2$

4.4 `FindAndDelete` Entropy vs Puzzle Difficulty

Parameter choice: We set $E = W_2$ where:

- $E = \log_2 \binom{n}{t}$**:** Entropy from `FindAndDelete` combinations (bits)
- W_2**:** Puzzle difficulty from signature size constraint (bits)

Why $E = W_2$: Setting $E = W_2$ ensures each combination is tried exactly once on average, maximizing efficiency without wasting script space or grinding work.

Concrete example: For $W_2 = 42$ bits, we choose $n = 120$, $t = 8$:

- $\binom{120}{8} \approx 2^{39.6} \rightarrow E \approx 39.6 \approx W_2$
- Digest entropy: $\sim$40 bits per round
- Two rounds $\rightarrow$ $\sim$79 bits total digest entropy

4.5 Concrete Example: 10-Choose-5

To illustrate with small parameters providing $\sim$1 byte of nonce space ($n = 10$, $t = 5$):

Locking Script:

```
// Nonce pool: 10 signatures embedded in script
<sig_1> <sig_2> <sig_3> <sig_4> <sig_5>
<sig_6> <sig_7> <sig_8> <sig_9> <sig_10>

// CHECKMULTISIG setup
OP_0   // Dummy element because of the OP_CHECKMULTISIG bug

<6>   // Number of signatures

```

```
// Use OP_ROLL to select signatures by index
<10> OP_ROLL OP_ROLL  // Move sig[index_a] to top
<10> OP_ROLL OP_ROLL  // Move sig[index_b] to top
<10> OP_ROLL OP_ROLL  // Move sig[index_c] to top
<10> OP_ROLL OP_ROLL  // Move sig[index_d] to top
<10> OP_ROLL OP_ROLL  // Move sig[index_e] to top

<10> OP_ROLL  // Move puzzle sig to top
OP_SIZE <60-1> OP_EQUALVERIFY // The puzzle requires 1 byte of work

// Verify all 6 signatures (5 from pool + 1 puzzle)
<pubkey_puzzle>
<pubkey_1> <pubkey_2> <pubkey_3> <pubkey_4> <pubkey_5>
<pubkey_6> <pubkey_7> <pubkey_8> <pubkey_9> <pubkey_10>
<11>   // Number of public keys
OP_CHECKMULTISIG
```

Unlocking Script:

```
<signature_puzzle>
<index_e> <index_d> <index_c> <index_b> <index_a>
```

How it works:

1. Spender tries subsets: [1,2,3,4,5], [1,2,3,4,6], [2,4,5,7,9], etc.
2. For each subset, `OP_ROLL` picks those 5 signatures from the pool of 10
3. `FindAndDelete` removes them → unique sighash per subset
4. Spender grinds until finding a subset where the sighash produces a $\leq$59 byte puzzle signature
5. Winner (e.g., subset [2,4,5,7,9]): unlocking script = `<puzzle_sig> <9> <7> <5> <4> <2>`

Result: The indices [2,4,5,7,9] are the Binohash, readable directly from the unlocking script. With $\binom{10}{5} = 252$ subsets $\rightarrow \log_2(252) \approx 8$ bits of entropy ($\sim$1 byte).

4.6 Security Analysis

Honest spender work:

- Stage 1 (pinning): 4×2^{W_1} operations
- Stage 2 (nonce grinding): 2^{W_2} operations
- **Total:** $4 \times 2^{W_1} + 2^{W_2}$ (assuming $W_1 \approx W_2$, this is $\sim W_2 + 2.3$ bits)

Attacker sample cost (per collision attempt):

- Modify transaction: redo 1 pinning signature = 2^{W_1} operations
- Find valid nonce: grind puzzle signature = 2^{W_2} operations
- **Sample cost:** $2^{W_1} + 2^{W_2}$ (assuming $W_1 \approx W_2$, this is $\approx W_2 + 1$ bits)

Collision resistance:

- Binohash size: $\min(E, W_2)$ bits (from subset selection, limited by either combinations or puzzle difficulty)
- With $E \approx W_2$: Binohash size $\approx W_2$ bits
- Birthday bound [18]: need $\sim 2^{W_2/2}$ samples to find collision
- Total attacker work: $W_2/2 + (W_2 + 1) = 3W_2/2 + 1$ bits
- **Collision resistance:** $W_1 + W_2/2$ **bits** (or $\sim 3W_2/2$ bits when $W_1 = W_2$)

Concrete parameters: With $W_1 = W_2 = 42$ bits:

- Honest work: $\sim$44.3 bits
- Binohash: $\sim$40 bits (from $\binom{120}{8} \approx 2^{39.6}$)
- Attack cost: $\sim$63 bits
- **Collision resistance: $\sim$63 bits**

4.7 Limitation

$W_1 + W_2/2$ bits of collision resistance is insufficient for high-security applications requiring $\geq$80 bits. With $W_1 = W_2 = 42$, we achieve only $\sim$63 bits.

To reach 80 bits with single round: Would need $\log_2 \binom{n}{t} \geq 80$ bits, making $\binom{n}{t} \approx 2^{80}$, which is impractical (would need enormous amounts of grinding, and large n or t, exceeding script size limits).

Solution: Add a second round to Stage 2.

5 Two Round Scheme

5.1 Overview

Three stages:

1. **Stage 1:** Transaction Pinning (4 signature puzzles with $W_1 = 42$ bits each = $\sim$44 bits total)
2. **Stage 2 Round 1:** First nonce extraction ($W_2 = 42$ bits)
3. **Stage 2 Round 2:** Second nonce extraction ($W_2 = 42$ bits)

5.2 Stage 2: Nonce Extraction (Two Rounds)

Execute two `FindAndDelete` rounds with different signature pools, each with parameters (n, t):

- Embed n signatures in locking script
- Spender selects subset of t indices $\rightarrow \binom{n}{t}$ possible sighashes
- Grind puzzle signature with $W_2 = 42$ bits
- **Output:** t indices as nonce component

Combined Binohash: The concatenation of both sets of indices ($2t$ indices total)
Parameters: $n = 120$, $t = 8$ per round $\rightarrow \binom{120}{8} \approx 2^{39.6}$ per round $\rightarrow \sim$79 bits total Binohash

5.3 Security Analysis

Honest spender work:

- Stage 1 (pinning): 4×2^{W_1} operations ($\sim$44 bits for $W_1 = 42$)
- Stage 2 Round 1: 2^{W_2} operations (42 bits)
- Stage 2 Round 2: 2^{W_2} operations (42 bits)
- **Total:** $4 \times 2^{W_1} + 2 \times 2^{W_2}$ **operations** (for $W_1 = W_2 = 42$: $6 \times 2^{42} \approx 2^{44.58}$, i.e. $\sim$44.6 bits, feasible with GPU!)

Attacker sample cost (per collision attempt):

- Modify transaction: redo 1 pinning signature = 2^{W_1} operations (42 bits)
- Find valid nonce (both rounds): 2×2^{W_2} operations = 2^{W_2+1} (43 bits for $W_2 = 42$)
- **Sample cost:** $2^{W_1} + 2 \times 2^{W_2}$ **operations** (for $W_1 = W_2 = 42$: $3 \times 2^{42} \approx 2^{43.6}$, i.e. $\sim$43.6 bits)

Collision resistance:

- Binohash size per round: $\min(E, W_2)$ bits, where $E = \log_2 \binom{120}{8} \approx 39.6$ bits
- With $E \approx W_2 \approx 42$: each round contributes $\sim$40 bits
- Total Binohash size: $\sim$79 bits (two rounds of $\sim$40 bits each)
- Birthday bound: need $\sim 2^{(2E)/2} = 2^E \approx 2^{39.6}$ samples to find a collision
- Total attacker work (for $W_1 = W_2 = 42$): $39.6 + \log_2(3 \times 2^{42}) \approx 39.6 + 43.6 = \sim 83.2$ bits
- **Collision resistance:** $\sim$**84 bits** (with $W_1 = W_2 = 42$, rounding)

Why this works: Two rounds with $\sim$40 bits each provide $\sim$79 bits of Binohash entropy, requiring the birthday bound to be $\sim 2^{40}$ samples (instead of 2^{20} for single round with 40 bits). Combined with a per-sample cost of $\sim$43.6 bits (for $W_1 = W_2 = 42$), this yields $\sim$83-bit collision cost.

5.4 Work Binding Property

The construction ensures that:

- Cannot perform nonce grinding (Stage 2) without transaction pinning (Stage 1)
- Each collision attempt forces the attacker to redo pinning work plus nonce grinding for both rounds
- Work accumulates: sample cost $= 2^{W_1} + 2 \times 2^{W_2}$ operations

Key insight: The honest spender does ∼44.6 bits work total, while the attacker needs $\sim E \approx 39.6$ bits (birthday bound over the $\sim 2E$-bit digest) plus ~ 43.6 bits per sample (for $W_1 = W_2 = 42$), totaling ~ 83.2 bits to find collisions.

6 Implementation

6.1 Script Structure

For two-round scheme with $n = 120$, $t = 8$:

Total non-push opcodes: ∼199

- Stage 1 (pinning): 13 opcodes (see Appendix A)
- Stage 2 (two rounds): $R \times (11t + 4) + 2 = 186$ opcodes for $R = 2$, $t = 8$ (see Appendix B)

Total script size (locking script): ∼8 KB

- Stage 1 (pinning): ∼100 bytes (4 signatures, size checks, opcodes)
- Stage 2: $2n = 240$ minimum-size signatures (9 bytes each): $240 \times 10 = 2{,}400$ bytes (including push opcodes)
- Stage 2: $2n = 240$ `HASH160` values (20 bytes each): $240 \times 21 = 5{,}040$ bytes (including push opcodes)
- Stage 2: Loop and control flow for two rounds: $2 \times (168 + 76) = 488$ bytes
- Concrete total for $(n, t) = (120, 8)$: $100 + 7{,}928 = 8{,}028$ bytes (see formula below)

6.2 Parameters

Configuration	Pinning	Puzzle (per round)	n	t	Collision Resistance	Honest Work
Maximum	5 bytes	5 bytes	120	8	∼84 bits	∼44.6 bits
Higher W1	6 bytes	5 bytes	120	8	∼92 bits	∼52.6 bits

Note: The 201 opcode limit restricts us to $t = 8$. Increasing security beyond ∼84 bits requires increasing W_1 (pinning work) while keeping $W_2 = 42$ bits and $t = 8$.

Note: A signature that is $60 - n$ bytes short requires grinding $8n + 2$ bits of work due to DER non-negative encoding and the bitshift (i.e. the multiplication by 2) in $s = 2(z + 1)$.

6.2.1 Opcode Constraints and Parameter Selection

Asymmetry between W_1 and W_2:

- **Increasing W_1 (pinning work):** Costs zero opcodes
  - W_1 only affects the grinding difficulty (signature size constraint)
  - The 4 pinning signatures use the same opcodes regardless of W_1
  - Can easily increase to $W_1 = 49$ bits (6 bytes), 57 bits (7 bytes), etc.
- **Increasing W_2 (puzzle work):** Costs opcodes rapidly
  - Higher W_2 requires larger $E = \log_2 \binom{n}{t}$ to match (see §4.4)
  - Larger $\binom{n}{t}$ requires bigger n (more signatures in pool) or bigger t (more selected)
  - Each additional signature in the pool costs ∼1 opcode (for `OP_ROLL` selection)
  - **201 opcode limit severely constrains W_2**

Opcode count: Non-push opcodes = $R\times(11t+4)+2$, where pubkey arguments to `OP_CHECKMULTISIG` count as opcodes (see Appendix B)

Script size: Total bytes $\approx 100 + R \times (244 + 31n)$ for R rounds, or $\approx 100 + 488 + 62n$ for $R = 2$ rounds

Practical limit: With the 201 non-push opcode limit and 10KB script size limit:

- $W_2 = 42$ bits: $n = 120$, $t = 8 \to \binom{120}{8} \approx 2^{39.6} \to 199$ opcodes, 8,028 bytes
- $W_2 = 49$ bits: $n = 140$, $t = 9 \to \binom{140}{9} \approx 2^{48.5} \to 220$ opcodes, 9,368 bytes (exceeds opcode limit!)
- $W_2 = 57$ bits: $n = 160$, $t = 10 \to \binom{160}{10} \approx 2^{56.7} \to 242$ opcodes, 10,708 bytes (exceeds both limits!)

Design implication: With 199 total opcodes for $t = 8$, we are close to the 201 non-push opcode limit. The collision resistance is ∼84 bits with $W_1 = W_2 = 42$. Increasing to $t = 9$ would require 220 opcodes (exceeds limit). Further security increases must come from increasing W_1 alone (which costs zero opcodes), improving collision resistance while keeping $t = 8$.

6.3 Compact Signatures via ECDSA Key Recovery

As described in §2.2–§2.3, we use ECDSA key recovery with minimum-size 9-byte signatures to create compact dummy signatures instead of ∼73 bytes.

Savings: For $n = 120$ signatures per round × 2 rounds = 240 signatures:

- Normal: $240 \times 73 = 17{,}520$ bytes (exceeds 10KB limit!)
- Optimized: $240 \times 9 = 2{,}160$ bytes (fits comfortably)
- Total savings: ∼15.4 KB

6.4 Lamport Signature Integration for BitVM

Why not use regular Lamport or Winternitz signatures?

Traditional Lamport signatures require one hash preimage reveal per bit of the message. For an 80-bit Binohash, this would require:

- 80 `OP_HASH160` operations to verify the preimages
- 80 `OP_EQUALVERIFY` operations to check against commitments
- **Total:** 160 non-push opcodes (plus additional stack operations)

Script-optimized variants exist but remain opcode-intensive:

- **BitVM's 2-bit Lamport commitment**[19]: 5.5 opcodes per bit (11 opcodes for 2 bits)
- For 80 bits: $80 \times 5.5 = 440$ opcodes (exceeds Bitcoin's 201 limit)
- **Winternitz signatures**[20]: Cheaper in script size but over $2\times$ more expensive in opcodes

With Bitcoin's 201 non-push opcode limit, these schemes leave little room for the rest of the script logic (`FindAndDelete` operations, signature checks, etc.).

Our approach: Subset-based signatures

We use a variant of one-time signatures based on subset preimage revelation, similar to the HORS (Hash to Obtain Random Subset) scheme by Reyzin and Reyzin[21]:

1. **Setup:** Embed n hash commitments $H(\text{preimage}_1), \ldots, H(\text{preimage}_n)$ in the locking script
2. **Signing:** To sign message m, reveal preimages for a subset of t indices determined by m
3. **Verification:** Check that the revealed preimages hash to the committed values at the specified indices

Key efficiency: This requires only t hash checks instead of one per message bit.

Integration with `FindAndDelete`:

Our scheme naturally integrates with the `FindAndDelete` nonce extraction:

- The subset indices $[i_1, i_2, \ldots, i_t]$ from `FindAndDelete` are the message being signed
- We use the same indices for both `FindAndDelete` and subset signature
- For each selected index i, we reveal preimage$[i]$ alongside the dummy signature
- This effectively signs the nonce (the transaction digest)

The full flow:

1. **Extract Binohash in Script:**
   - The subset indices ($2t$ values from two rounds) are the Binohash
   - These are directly readable from the unlocking script

2. **Sign the Binohash:**
    - Locking script contains: hash commitments $H(\text{preimage}_1), \ldots, H(\text{preimage}_n)$ alongside dummy signatures
    - For each index i selected in `FindAndDelete`, reveal preimage[i]
    - Verify: `<preimage[i]> OP_HASH160 <i> OP_PICK <H(preimage[i])> OP_EQUALVERIFY`
    - The revealed preimages commit to the specific subset selection (the Binohash)
3. **Import into BitVM:**
    - The subset signature is verified in Script
    - BitVM's off-chain verification can check the signature and transaction properties

Example: For $n = 120$, $t = 8$, two rounds:

- Binohash: 16 indices (8 from each round)
- Subset signature: 16 preimages revealed, 16 hash checks
- **Opcodes: 48 non-push opcodes** (16×`OP_HASH160` + 16×`OP_EQUALVERIFY` + 16×`OP_ROLL`)

Comparison: Regular Lamport for 40 bits requires 220 opcodes. Our subset scheme for ∼40 bits: 48 opcodes. That's over **4× more efficient**.

Note: In production it might be relevant to ensure that the the preimages have a specific byte length.

7 Attacks and Mitigations

7.1 Attack: Skipping Transaction Pinning

Attack strategy:

1. Skip the full Phase 1 (transaction pinning) and only solve the `ANYONECANPAY|NONE` puzzle
2. Use the subset selection in Phase 2 to solve the puzzle with an `ANYONECANPAY|NONE` signature

Effect: The Binohash effectively only commits to the `ANYONECANPAY|NONE` sighash, which allows the attacker to arbitrarily change all outputs, other inputs, and most transaction fields.

Collision resistance is broken:

Given an honest transaction txA with Binohash D, the attacker can create a malicious transaction txB with the same Binohash D at very low cost:

- The attacker only needs to grind Phase 2 (the two nonce extraction rounds)
- Attack cost: 2×2^{W_2} operations (just the two puzzle signatures, ∼42 bits for $W_2 = 42$)
- No Phase 1 pinning required since `ANYONECANPAY|NONE` doesn't commit to outputs/other inputs

- The attacker can change outputs arbitrarily without invalidating the Binohash

This is a second preimage attack that costs only ∼42 bits, making it practical to find collisions. The security degrades from ∼83 bits to only ∼42 bits.

Mitigation in BitVM:

For BitVM bridge applications, this attack is not a concern:

1. The Binohash (subset indices) is Lamport-signed and fed into BitVM
2. BitVM verifies the full spending transaction, including which sighash modes were used and reject the proof if any phase 2 puzzle signature used `ANYONECANPAY|NONE` instead of `ALL`

Conclusion: The Binohash maintains ∼83 bits collision resistance. The attacker cannot cheaply find colliding transactions, because BitVM can enforce proper sighash mode usage during verification.

7.2 Attack: Sighash Byte Entropy

Problem: Any sighash byte value (0-255) is consensus-valid in Bitcoin's legacy script validation[1], giving attacker 8 bits of additional grinding per signature at negligible cost relative to pinning work.

With 2 puzzle signatures (one per round), the attacker gets $2 \times 8 = 16$ bits of extra degrees of freedom. The sighash byte variation allows trying 2^{16} different sighashes per transaction structure without changing the transaction body. This doesn't change the Binohash (subset indices) or its entropy, but reduces the cost to find which subsets produce valid puzzle signatures.

Impact on collision resistance:

- The sighash byte freedom reduces puzzle grinding work by 8 bits per signature (16 bits total for 2 rounds)
- Effective puzzle work: $W_2 - 8$ bits per round instead of W_2 bits
- The Binohash entropy itself ($2W_2$ bits) remains unchanged
- **Key insight:** When $W_1 = W_2$, pinning work dominates sample cost, so puzzle work reduction has minimal impact
- Sample cost with sighash entropy: $4 \times 2^{W_1} + 2 \times 2^{W_2-8} \approx 4 \times 2^{W_1}$ (when $W_1 = W_2 = 42$, since $4 \times 2^{42} \gg 2 \times 2^{34}$)
- Without entropy: $4 \times 2^{W_1} + 2 \times 2^{W_2} = 6 \times 2^{W_1}$ (when $W_1 = W_2$)
- Cost ratio: $6/4 = 1.5$
- **Estimated reduction: ∼0.6 bits** (factor of 1.5 in sample cost $\rightarrow \log_2(1.5) \approx 0.585$ bits)

[1]The sighash type is included in the signature-hash computation as specified in Bitcoin's `SignatureHash` implementation. While only specific types (ALL/NONE/SINGLE with optional ANYONECANPAY) have defined semantics, consensus does not reject unknown values.

Mitigation in BitVM: Enforcing Canonical Sighash Flags

In BitVM applications, we enforce that honest transactions use canonical `SIGHASH_ALL` (0x01). BitVM receives the full signed transaction including witness data and can directly inspect the sighash flag byte appended to each signature. BitVM verifies the transaction off-chain and rejects proofs using non-canonical flags. Script cannot inspect individual bytes within signatures after `OP_SIZE`, making off-chain verification the only enforcement mechanism.

Effect on collision resistance:

In the BitVM enforcement model, collision-finding attacks must generate two types of samples with different constraints. The threat model: an attacker searches for a collision (txA, txB) where both have the same Binohash:

- **txA (onchain):** posted on blockchain → can use any flags → retains 16 bits of sighash byte freedom
- **txB (BitVM):** verified off-chain in BitVM → must use canonical 0x01 flags → no sighash byte freedom

Both transactions are generated by the attacker. The goal: execute txA on-chain but verify txB in BitVM with Binohash(txA) = Binohash(txB).

Analysis:

For a birthday attack on a $2W_2$-bit Binohash, the attacker must generate samples from both distributions until finding a collision.

Sample costs (with $W_1 = W_2 = 42$):

- **txA sample:** $4 \times 2^{W_1} + 2 \times 2^{W_2-8} \approx 4 \times 2^{42} = 2^{44}$ ($\sim$44 bits)
- **txB sample:** $4 \times 2^{W_1} + 2 \times 2^{W_2} = 6 \times 2^{42} \approx 2^{44.6}$ ($\sim$44.6 bits)

The txB samples are $\sim$0.6 bits more expensive due to no sighash byte freedom.

Optimal attack strategy:

The attacker minimizes total cost by generating:

- $N_A = \sqrt{2^{2W_2} \times \frac{3}{2}} \approx 2^{W_2+0.29}$ txA samples (more, since they're cheaper)
- $N_B = \sqrt{2^{2W_2} \times \frac{2}{3}} \approx 2^{W_2-0.29}$ txB samples (fewer, since they're more expensive)

where the allocation balances work: $N_A \times \text{cost}_A = N_B \times \text{cost}_B$, subject to collision constraint $N_A \times N_B \approx 2^{2W_2}$.

Total collision cost with mitigation (cost ratio $r = 1.5$):

- Cost $= 2\sqrt{1.5} \times 2^{W_1+W_2} \approx 2^{W_1+W_2+1.29}$ (since $\log_2(2\sqrt{1.5}) \approx 1.29$)

Without mitigation (both can use sighash bytes, so cost ratio $r = 1$):

- Both samples cost $4 \times 2^{W_1} = 4 \times 2^{42}$
- Cost $= 2 \times 2^{W_1+W_2} = 2^{W_1+W_2+1}$

Concrete parameters: With $W_1 = W_2 = 42$:

- Without mitigation: $2^{84+1} = 2^{85}$ bits (baseline from Section 5.3 assuming honest canonical usage)
- With sighash entropy attack (no BitVM enforcement): effectively ∼84.4 bits[2]
- With BitVM enforcement: $2^{84+1.29} \approx 2^{85.3}$ bits
- **Net improvement over unmitigated attack: ∼0.9 bits** (making txA samples 1.5× more expensive)

Conclusion: When $W_1 = W_2$, enforcing canonical flags on the BitVM-verified transaction partially mitigates the sighash entropy attack by creating cost asymmetry (ratio 1.5×), forcing the attacker into a slightly more expensive asymmetric birthday attack that approaches the baseline security of ∼85 bits.

8 Limitations & Open Problems

8.1 Grinding Performance & Cost

8.1.1 No Elliptic Curve Operations Required

A key advantage: grinding signature puzzles requires no elliptic curve or field arithmetic operations.

Recall from §2.3 that with $R_{\min} = 2^{-1}G$ fixed and $d = r_{\min}^{-1}$, we have:

$$s = 2(z + 1) \tag{4}$$

To check if a signature satisfies the puzzle (size exactly $60 - n$ bytes), we only need to verify that the sighash z has $(8n + 2)$ leading zero bits. The grinding process is:

1. Compute sighash z for the current subset selection (double SHA-256 hashing)
2. Check if z has $(8n + 2)$ leading zero bits
3. If yes, construct $s = 2(z + 1)$ and return signature $(R_{\min}, s)$
4. If no, try a different subset

No need for: EC point operations, modular arithmetic, signature verification during grinding. Only double SHA-256 hashing to compute z.

[2]The ∼85 bits baseline assumes participants use canonical flags. Unrestricted sighash grinding reduces effective security by ∼0.6 bits at the sample level, translating to ∼0.3 bits at the collision level.

8.1.2 Hardware Considerations & Performance

Differences from Bitcoin mining:

- **Large preimages:** Sighash preimages are ∼9KB (not 80-byte block headers)
- **Complex variation:** Must delete 8 random 10-byte (9-byte signature + 1 byte push opcode) substrings from a 120 × 10 byte section (`FindAndDelete`), not just varying a single nonce field
- **Memory intensive:** Requires manipulating large `scriptCode` buffers

ASIC viability: Standard Bitcoin mining ASICs cannot be used directly, despite both using double SHA-256. Mining ASICs are optimized for:

- Fixed 80-byte preimages with only nonce variation
- Double SHA-256 (SHA-256(SHA-256(x))) on fixed-size inputs
- Not for large variable-length preimages (∼9KB) with complex `FindAndDelete` manipulations

GPU performance: GPUs are well-suited for this workload due to:

- Efficient double SHA-256 implementations
- Ability to handle variable-length inputs
- Parallelizable subset enumeration

Measured performance:

- **MacBook M1** (7 GPU cores): ∼1 MH/s for ∼9KB preimages with `FindAndDelete` operations
- **Cloud GPU** (10× RTX 4090): ∼320 MH/s (rental cost: $3.22/hour on vast.ai)

Midstate optimization for Round 2: The second nonce extraction round benefits from midstate optimization. Of the ∼9KB sighash preimage, approximately 7KB is fixed (transaction data, pinning signatures, and Round 1 selections). By pre-computing the SHA-256 midstate of these fixed bytes, Round 2 grinding only needs to hash the remaining ∼2KB that varies with each subset selection. This significantly increases the effective hashrate for Round 2 compared to Round 1.

Time estimates for $W_2 = 42$ bits (single transaction):

- **10× RTX 4090** (∼320 MH/s): $2^{42}/(320 \times 10^6) \approx$ **3.8 hours**
- With midstate optimization for Round 2, effective time is reduced further

Scaling options:

- **Multi-GPU:** Linear scaling with number of GPUs
- **Cloud GPU rental:** Cost-effective parallel grinding via vast.ai or similar providers
- **FPGAs:** Custom FPGA implementations could provide additional speedup for the `FindAndDelete` + SHA256 pipeline

8.1.3 Cost Analysis

Configuration	**Basic**	**High Security**
Parameters	$W_1 = W_2 = 42$	$W_1 = W_2 = 50$
Honest work	∼44.6 bits	∼52.6 bits
Measured performance (10× RTX 4090 @ 320 MH/s):		
Time	∼3.8 hours	∼40 days
Cloud cost (on vast.ai)	<$15	∼$3,100
Reference (MacBook M1 @ 1 MH/s):		
Time	∼50 days	∼35 years

Feasibility: The basic configuration ($W_1 = W_2 = 42$) is highly practical with cloud GPUs, costing under $15 and completing in under 4 hours. With midstate optimization for Round 2, effective costs are even lower. High security configurations ($W_1 = W_2 = 50$) remain feasible for high-value applications at approximately $3,000 in cloud compute.

Room for optimization: These estimates use a naive implementation with room for improvement:

- **Script size:** The locking script can be optimized to reduce preimage size (e.g., using `OP_CODESEPARATOR` to truncate the `scriptCode`)
- **Midstate-friendly structure:** Restructuring the script to maximize the fixed prefix would improve midstate optimization benefits
- **GPU miner:** Our GPU implementation is unoptimized; significant speedups are possible through better memory access patterns, kernel optimization, and batched subset enumeration

8.2 Non-Standard Transaction Mining

As explained in §2.7, these transactions:

- Cannot enter standard mempools
- Require direct mining via `generateblock` RPC
- Or mining pools running modified nodes

This creates a liveness assumption: miners must be willing to include non-standard transactions. However, this assumption is increasingly supported by concrete infrastructure and industry commitments.

Private Mempool Services. In February 2024, Marathon Digital Holdings launched Slipstream [22], a private mempool service for direct submission of non-standard transactions that comply with consensus rules but are excluded from standard mempools.

Mining Pool Support for BitVM. In May 2025, three major mining pools (Antpool, F2Pool, and SpiderPool) representing approximately 36–40% of Bitcoin's total hashrate committed to mining non-standard transactions for BitVM protocols [23]. This demonstrates substantial industry willingness to support such protocols in production.

These developments indicate that the liveness assumption required for non-standard transaction mining is not merely theoretical but backed by concrete partnerships, significant hashrate commitments, and deployed infrastructure. For protocols like those using Binohash, this substantially reduces the practical barrier to adoption.

9 Conclusion

We presented Binohash, a collision-resistant hash function for Bitcoin transactions that enables transaction introspection in Bitcoin Script using `FindAndDelete` grinding. Our two-round scheme achieves ∼84 bits of collision resistance with ∼44.6 bits honest work (feasible with GPU acceleration).

Key contributions:

1. Three fundamental primitives: transaction pinning with signature puzzles, `FindAndDelete` nonce space, and an opcode-efficient variant of Lamport signatures based on subset selections.
2. Efficient two-round protocol achieving 80+ bit security
3. Concrete construction using only legacy Script opcodes
4. Detailed analysis of signature grinding

While requiring legacy bare scripts and incurring high computational costs, this technique enables permissionless data availability for protocols like BitVM bridges without consensus changes. The resulting transaction is roughly 10 kB in size, which costs about $40 in fees at current market rates. Proof-of-concept transactions demonstrating a 4-byte Binohash have been executed on both Testnet4 [24] and Bitcoin Mainnet [25].

Future work:

- Optimize script size and opcode count
- Integration with BitVM bridge implementations
- A variant that violates fewer hardcoded standardness rules
- Improve collision resistance to at least 100 bits

10 Acknowledgments

We thank Liam Eagen, Ethan Heilman, Antoine Poinsot, Super Testnet, Adam Borco, and Ekrem Bal for valuable discussions and feedback on this work. Furthermore, we thank the Slipstream team for mining our test transactions. We apologize to Pieter Wuille for abusing the Bitcoin Script interpreter so badly.

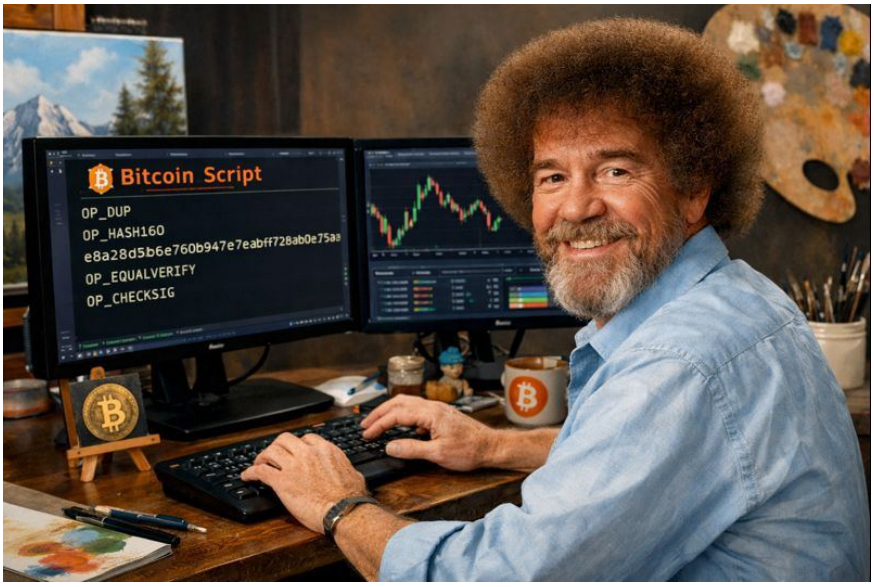

Figure 1: *"We don't have bugs, we have happy little accidents."*

References

[1] R. Linus, "BitVM: Compute anything on Bitcoin," 2023. [Online]. Available: https://bitvm.org/bitvm.pdf

[2] R. Linus, L. Aumayr, A. Pelosi, Z. Avirikioti, and M. Maffei, "BitVM2: Bridging Bitcoin to second layers," 2024. [Online]. Available: https://bitvm.org/bitvm_bridge.pdf

[3] R. Linus, "BitVM3," 2025. [Online]. Available: https://bitvm.org/bitvm3.pdf

[4] E. Heilman *et al.*, "Signing a Bitcoin transaction with Lamport signatures," Bitcoin Development Mailing List, 2024. [Online]. Available: https://groups.google.com/g/bitcoindev/c/mR53go5gHIk

[5] E. Heilman, V. Kolobov, A. Levy, and A. Poelstra, "ColliderScript: Covenants in Bitcoin via 160-bit hash collisions," Cryptology ePrint Archive, Report 2024/1802, 2024. [Online]. Available: https://eprint.iacr.org/2024/1802

[6] "ColliderVM: Stateful computation on Bitcoin without fraud proofs," Cryptology ePrint Archive, Report 2025/591, 2025. [Online]. Available: https://eprint.iacr.org/2025/591.pdf

[7] G. Maxwell, "CoinSwap: Transaction graph disjoint trustless trading," BitcoinTalk Forum, 2013. [Online]. Available: https://bitcointalk.org/index.php?topic=321228.msg13072047#msg13072047

[8] A. Towns, "Better privacy with SNARKs," Lightning-dev mailing list, November 2015. [Online]. Available: https://diyhpl.us/~bryan/irc/bitcoin/bitcoin-dev/linuxfoundation-pipermail/lightning-dev/2015-November/000344.txt

[9] "Proof of work in Bitcoin script," GitHub, coins/bitcoin-scripts, 2020. [Online]. Available: https://github.com/coins/bitcoin-scripts/blob/master/proof-of-work.md

[10] A. Borco, "Bitcoin PoW-locked outputs," GitHub repository, 2024. [Online]. Available: https://github.com/adambor/btc-pow-locked-outputs

[11] VzxPLnHqr, "sig-pow: Proof of work via ECDSA signature length," GitHub repository, 2022. [Online]. Available: https://github.com/VzxPLnHqr/sig-pow

[12] "BIP 66: Strict DER encoding," Bitcoin Improvement Proposal. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki

[13] N. Heninger, T. Scholl, and D. Shumow, "48ce563f89a0ed9414f5aa28ad0d96d6795f9c62," Crypto 2019 Rump Session, August 2019. [Online]. Available: https://www.youtube.com/watch?v=NGLR2N4EK58

[14] P. Todd, "SIGHASH_SINGLE bug," BitcoinTalk Forum, 2013. [Online]. Available: https://bitcointalk.org/index.php?topic=260595.0

[15] "BIP 143: Transaction signature verification for version 0 witness program," Bitcoin Improvement Proposal. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki#no-findanddelete

[16] M. Corallo, "Great consensus cleanup (original 2019 proposal)," 2019. [Online]. Available: https://github.com/TheBlueMatt/bips/blob/cleanup-softfork/bip-XXXX.mediawiki

[17] A. Poinsot, "BIP-54: The great consensus cleanup," Bitcoin Improvement Proposals, 2024. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0054.md

[18] G. Yuval, "How to swindle rabin," *Cryptologia*, vol. 3, no. 3, pp. 187–189, 1979.

[19] BitVM Contributors, "2-bit Lamport commitment," GitHub, 2024. [Online]. Available: https://github.com/BitVM/bitvm-js/blob/main/docs/opcodes/2-bit_commitment.md

[20] Robin Linus, "Winternitz implementation in Bitcoin Script," GitHub, 2024. [Online]. Available: https://github.com/BitVM/bitvm-js/blob/4d45e52ef11259d121b0915aead92ae11b8cd51a/run/examples/winternitz.js

[21] L. Reyzin and N. Reyzin, "Better than BiBa: Short one-time signatures with fast signing and verifying," in *Proceedings of the 7th Australasian Conference on Information Security and Privacy (ACISP 2002)*, 2002. [Online]. Available: https://www.cs.bu.edu/~reyzin/papers/one-time-sigs.pdf

[22] Marathon Digital Holdings, "Marathon digital holdings launches slipstream," Press Release, February 2024. [Online]. Available: https://ir.mara.com/news-events/press-releases/detail/1343/marathon-digital-holdings-launches-slipstream

[23] BitLayer Labs, "Bitlayer joins forces with Antpool, F2Pool, and SpiderPool to supercharge Bitcoin DeFi," CoinDesk, May 2025. [Online]. Available: https://www.coindesk.com/business/2025/05/27/bitlayer-joins-forces-with-antpool-f2pool-and-spiderpool-to-supercharge-bitcoin-defi

[24] "Binohash proof-of-concept transaction on testnet4," Bitcoin Testnet4, 2025, 4-byte Binohash output demonstration. [Online]. Available: https://mempool.space/testnet4/tx/dc3d81543402a83c2fa99dcf558b5aae25c83c1a203e6a1a13e0b2a54f3f60ca

[25] "Binohash proof-of-concept transaction on bitcoin mainnet," Bitcoin Mainnet, 2025, 4-byte Binohash output demonstration. [Online]. Available: https://mempool.space/tx/203d82a95ac621439e4f35e5d59476e3058579482fe5396244c3b6a36ab95f30

[26] Chick3nman, "Hashcat v6.2.6 benchmark on the Nvidia RTX 4090," GitHub Gist, 2022, rIPEMD160 (mode 6000): 36.2 GH/s. [Online]. Available: https://gist.github.com/Chick3nman/32e662a5bb63bc4f51b847bb422222fd

A Optimized Transaction Pinning (13 Opcodes)

This appendix provides an optimized construction for transaction pinning (§3.1) that uses only 13 non-push opcodes by eliminating `OP_CODESEPARATOR` opcodes and the explicit SIGHASH_SINGLE exclusion.

A.1 Key Insight

By carefully choosing the discrete logarithms of the pinning pubkeys, we can ensure that no two puzzles can be solved with the same sighash value z. This eliminates the need for:

- `OP_CODESEPARATOR` to separate the signature verification contexts
- Explicit SIGHASH_SINGLE exclusion (handled implicitly by the dummy signatures)

A.2 Pubkey Construction

For pinning work parameter $W_1 = 8n + 2$ bits, choose discrete logarithms:

$$d_i = -(2^{256-W_1+i}) \cdot r_{\min}^{-1}$$

where:

- $i \in \{1, 2, 3, 4\}$ is the puzzle index
- $r_{\min}$ is the minimum valid r-value (see §2.1)
- Each d_i requires a different leading bit pattern in the sighash z

The pubkeys are then $P_i = d_i G$.

Property: Given the fixed $R = r_{\min} G$, solving puzzle i requires finding z such that

$$s = 2(z + d_i \cdot r_{\min}) = 2(z - 2^{256-W_1+i})$$

is small, which requires that z has W_1 many leading zeros, except for the i-th highest bit, which has to be 1. So the discrete logs d_i are chosen such that the required z values have distinct high-order bits, making it impossible to use the same z for multiple puzzles.

A.3 Script

```
//
// Unlocking Script
//
<sig_4> <sig_3> <sig_2> <sig_1>

//
// Locking Script (12 non-push opcodes)
//

// Exclude sighash_single
<9-byte signature sighash_single>
<recovered pubkey for z=1> OP_CHECKSIGVERIFY
```

```
// Puzzle 1: W_1 bits of work
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_1> OP_CHECKSIGVERIFY

// Puzzle 2: W_1 bits of work
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_2> OP_CHECKSIGVERIFY

// Puzzle 3: W_1 bits of work
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_3> OP_CHECKSIGVERIFY

// Puzzle 4: W_1 bits of work
OP_SIZE <60 - n> OP_EQUALVERIFY <pubkey_4> OP_CHECKSIGVERIFY
```

Listing 1: Optimized transaction pinning (13 opcodes)

Opcode count: $4 \times (\text{OP\_SIZE} + \text{OP\_EQUALVERIFY}) + 5 \times \text{OP\_CHECKSIGVERIFY} = 13$ opcodes

B Single-Round Protocol Implementation

This appendix provides the Bitcoin Script implementation of the single-round protocol described in Section 5.2, including a Lamport signature of the digest. The code uses the `bitcoin-script` Rust crate syntax.

B.1 Locking Script

```
script! {
    // Push hashes in reverse order so index 0 selects hash[0]
    for hash in hashes.iter().rev() {
        { hash.to_vec() }
    }

    // Push signatures in reverse order so index 0 selects sig[0]
    for sig in signatures.iter().rev() {
        { sig.clone() }
    }

    // Push dummy element for OP_CHECKMULTISIG bug
    OP_0

    // For i in [0..t] repeat
    for i in 0..t {
        // Move <index_i> to top of the stack
        { 2 * n - i } OP_ROLL
        // Sanitize <index_i> to prevent out-of-bound reads
        { n - i } OP_MIN
        // Copy index
        OP_DUP
        // Move hash[index_i] to top of the stack using: index + offset
        { n + 1 } OP_ADD
        OP_ROLL
        // Move <preimage_i> to top of stack
        { 2 * n + t - 2 * i } OP_ROLL
        // Hash the preimage and verify it matches
        OP_HASH160
```

```
        OP_EQUALVERIFY
        // Move <signature_i> to top of stack
        OP_ROLL
    }

    // Multisig check: (t+1) signatures with (t+1) pubkeys
    { 2 * n + 1 } OP_ROLL

    // Verify puzzle signature size is 60 - work bytes
    OP_SIZE
    { 60 - work } OP_EQUALVERIFY

    { t + 1 } // M = t+1 signatures

    // Move to the top the t pubkeys provided in scriptSig
    for _ in 0..t {
        { 2 * n + 2 } OP_ROLL
    }

    // Push the puzzle pubkey
    { puzzle_pubkey.to_vec() }

    { t + 1 } // N = t+1 pubkeys
    OP_CHECKMULTISIG
}
```

Listing 2: Single-round locking script

B.2 Unlocking Script

```
//
// Unlocking Script
//
script! {
    // Push puzzle signature
    { puzzle_sig.to_vec() }

    // Push pubkeys in reverse order so they align with signature
    // selection order
    for pubkey in pubkeys.iter().rev() {
        { pubkey.clone() }
    }

    // Push preimages
    for preimage in preimages {
        { preimage.clone() }
    }

    // Push indices
    for &index in indices {
        { index }
    }
}
```

Listing 3: Single-round unlocking script

C Application: BitVM Bridge with Rollup Data Availability

C.1 The Problem

BitVM bridges cannot natively verify the state of external systems (rollups, sidechains). When an operator initiates a peg-out claiming "Alice withdrew from the rollup, I paid her, now I take bridge funds," BitVM has no way to verify the rollup state. Traditional solutions rely on trusted oracles or light client fraud proofs with existential honesty assumptions.

Binohash enables cryptographically binding operators to rollup state published on Bitcoin's blockchain.

C.2 The Mechanism

The scheme uses Bitcoin as a Data Availability (DA) layer. Rollup state diffs are published in a sequence of transactions, where the latest transaction's TXID (call it X) commits to all prior state via prevout references. This transaction includes a "read output" spendable by the operator.

Figure 2 shows the transaction structure. The operator posts collateral in AssertTx with a timeout clause. To avoid losing collateral, they must execute a transaction spending *both*:

- The Binohash output (containing transaction introspection script)
- The read output from the DA chain (TXID = X)

The resulting Binohash H commits to the entire transaction, including all prevouts. Since the read output has TXID X, the Binohash H transitively commits to the entire rollup state history. The operator Lamport-signs H in Script, feeding it into BitVM's verification.

C.3 Proof Statement

The operator proves to BitVM: *"There exists a transaction with Binohash H spending read output TXID X, where X commits to rollup state S, and S authorizes my peg-out."*

Using SNARK verification, this separates public inputs (H, X) from private witness (full state S). The operator proves knowledge of valid state without revealing it on-chain.

C.4 Security

Collision resistance prevents equivocation: A dishonest operator attempting to execute txA on-chain (spending read output X_A) but submit txB to BitVM (claiming state X_B) with the same Binohash H must find a collision—two different transactions with the same Binohash. This requires ∼83 bits of work (from §7), making it computationally infeasible.

D Polyglot Digest Optimization

The dummy signatures and HORS hash commitments can be merged into a single "polyglot digest," reducing script size and opcode count.

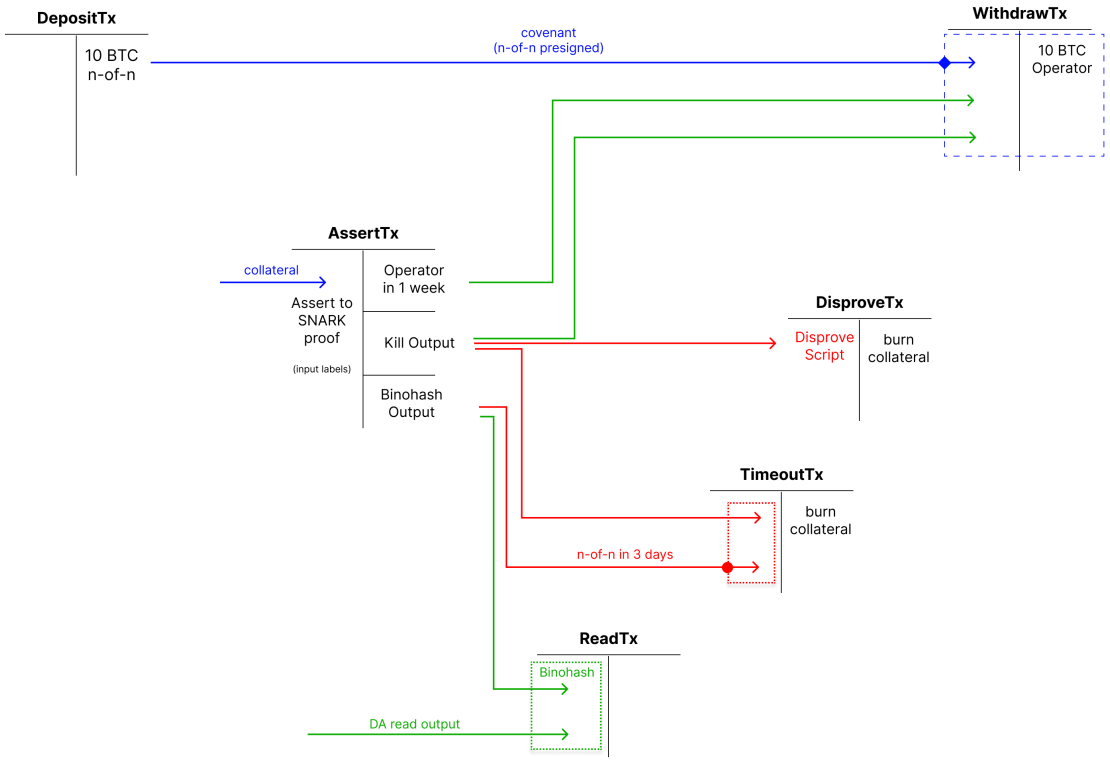

Figure 2: BitVM bridge architecture. Operator must execute ReadTx within 3 days or lose collateral via TimeoutTx. ReadTx commits to rollup state by spending the read output, making the Binohash transitively commit to all published data.

Construction: Grind 20-byte preimages P_i such that `RIPEMD160`(P_i) is a valid 20-byte DER-encoded ECDSA signature. The preimage P_i serves as the HORS preimage, while its hash serves as a dummy signature (with the corresponding public key derived via ECDSA key recovery).

DER constraints: A 20-byte value is a valid DER signature if it matches the structure `30 [len] 02 [r-len] [r] 02 [s-len] [s] [sighash]` with appropriate length fields and r, s values having MSB < 128. Multiple valid (r-len, s-len) combinations exist, reducing grinding difficulty to $\sim$46 bits per polyglot.

Benefits:

- Eliminates separate dummy signatures (saves $R \times n \times 10$ bytes)
- Reduces opcode count from $R \times (11t + 4) + 2$ to $R \times (9t + 4) + 2$
- Enables $t = 10$ (vs. $t = 8$), allowing smaller $n = 90$ (vs. $n = 120$) for equivalent entropy
- Total savings: $\sim$3.7 KB (from 7.4 KB to 3.8 KB for signature/hash data)
- Smaller script $\rightarrow$ smaller sighash preimage $\rightarrow$ faster Binohash grinding
- Smaller transaction $\rightarrow$ lower fees

Cost: For $n = 90$ per round (180 total polyglots): $180 \times 2^{46} \approx 2^{53.5}$ RIPEMD160 hashes. At $\sim$36 GH/s per RTX 4090 [26], a 10-GPU cluster computes this in $\sim$9 hours ($\sim$\$30 on vast.ai). This

is a one-time cost per script instance, whereas Binohash grinding is only required for executed instances—relevant for BitVM bridges where many contracts are created but few are executed.

Drawback: The polyglot preimages are secrets (HORS signing keys). Unlike Binohash grinding which only reveals public data, polyglot grinding must be performed on trusted hardware—cloud GPUs would learn the preimages and could forge signatures.