Quantum-Safe Bitcoin Transactions Without Softforks

Avihu Mordechai Levy*
StarkWare

April 9, 2026

Abstract

We present QSB, a Quantum Safe Bitcoin transaction scheme that requires no changes to the Bitcoin protocol and remains secure even in the presence of Shor's algorithm. Our construction modifies BINOHASH (Linus, 2026), replacing its non-quantum-safe component—the signature-size-based proof-of-work puzzle—with a hash-to-sig puzzle whose security depends only on the pre-image resistance of RIPEMD-160. The resulting scheme achieves ~118-bit second pre-image resistance under the Shor threat model (roughly half under Grover's algorithm), at an estimated off-chain GPU cost of a few hundred dollars. The scheme fits within Bitcoin's legacy script constraints of 201 opcodes and 10,000 bytes.

1 Introduction

Bitcoin transactions are not quantum-safe. The primitive that binds the components of a transaction—its inputs, outputs, and spending conditions—is an ECDSA signature (pre-Taproot) or a Schnorr signature (post-Taproot), neither of which is post-quantum secure. This means that even spending conditions which are themselves quantum-safe can be subverted: if the signature scheme securing the transaction is broken, the entire transaction—including its outputs and other inputs—can be forged.

Consider, for example, a UTXO whose spending condition is knowledge of a hash preimage, which is quantum-safe on its own. A spender broadcasts a transaction revealing the preimage. An adversary who can break the signature scheme can take that preimage, construct a different transaction t' spending the same UTXO to an address they control, sign t' with a forged signature, and broadcast it instead.[1][2]

1.1 Related Work

Two recent works bring us close to the goal of quantum-safe Bitcoin transactions. The first is Heilman's "Signing a Bitcoin Transaction with Lamport Signatures (no OP_CAT)" [1], followed by the work of Robin Linus—BINOHASH [2]—on which the present scheme is based.

While these two works differ significantly in their constructions and security properties, they both rely on a common high-level approach. Inside the Script:

1. **Derive a cryptographically strong identifier** of the spending transaction—call it D.

*avihu@starkware.co, socials: @avihu28

[1] Under Taproot, the situation is strictly worse: every Taproot output is spendable via the key path, meaning knowledge of the private key corresponding to the internal public key (which aggregates the taptweak and the script tree root) suffices to spend it, regardless of the script conditions in the taptree [3]. A quantum adversary capable of computing discrete logarithms could therefore spend any Taproot output directly.

[2] Even if the transaction is broadcast privately or mined directly, other miners might have a rational incentive to fork the block including that transaction and create a competing chain paying this transaction to them (this is apart from potential dependency on trust here).

2. **Verify a Lamport signature** over D.

Since Lamport signatures are post-quantum secure, and they sign a cryptographically strong identifier of the transaction, it is not possible to modify the transaction without producing a new Lamport signature—which the attacker cannot forge, even with quantum computing capabilities.

1.1.1 Remaining Issues

There are two main issues with the existing schemes:

1. Reliance on signature size check. This is the central issue. Both schemes rely on a signature proof-of-work (PoW) puzzle based on signature size [4], verified via `OP_SIZE`. This puzzle assumes a specifically small ECDSA r value for which the discrete logarithm is known. Currently, a well-known minimal value r_{min} is used. However, an adversary with knowledge of discrete logarithms—as a quantum computer would provide—could compute the smallest possible r value ($r = 1$), breaking the `OP_SIZE` check for all transactions created with the current r_{min}.[3]

2. Sighash flag uncertainty. Since the sighash flag is not directly readable from Script, both works acknowledge potential attacks arising from this limitation. BINOHASH discusses this in detail [2, Section 7.1], identifying `ANYONECANPAY|NONE` as a potential attack vector that could break the security guarantees.

1.2 Our Contribution

In this work, we modify the BINOHASH scheme to address both issues identified in Section 1.1.1. We replace the signature-size-based PoW puzzle with a hash-to-sig puzzle: the script hashes a transaction-bound public key via RIPEMD-160 and checks whether the output is a valid DER signature—an event with probability $\sim 2^{-46}$, independent of any elliptic curve hardness assumption. Since the puzzle's signature is hardcoded with `SIGHASH_ALL`, the sighash flag uncertainty is also eliminated.

We further introduce *bonus keys*—subset selections that contribute to the search space without requiring Lamport verification—to bridge the gap between the fixed 2^{46} puzzle target and the available opcode budget. The resulting scheme achieves ~118-bit pre-image resistance and fits within 201 opcodes, at an off-chain GPU cost of a few hundred dollars.

1.3 Discussion

While this article describes a solution that works today for quantum-safe Bitcoin transactions, it should be treated as a last-resort measure for several reasons:

- Both the off-chain computational cost and the on-chain transaction cost do not scale to the full range of users, amounts, and throughput that Bitcoin targets (also, these are non-standard transactions, though with some security tradeoffs we might be able to bring them closer to standard).
- Apart from the cost, the user experience for generating the transaction is more complex than standard Bitcoin usage (see Section 3.5).
- The scheme does not yet cover all Bitcoin use cases (e.g., Lightning Network channels).

[3] Paradoxically, once $r = 1$ is found, it could be used *going forward* to strengthen these schemes: the PoW puzzle with $r = 1$ would be quantum-safe, since there is no smaller r to exploit. But all transactions created before that point, using the current r_{min}, would be vulnerable.

We believe improvements can be made on all three fronts, though it would not be the most straightforward path. To the extent that the quantum threat is believed to be real, it remains necessary to continue the ongoing effort to research and implement the best possible solution for Bitcoin—one that is maximally efficient, user-friendly, and answers Bitcoin's needs, through protocol-level changes.

2 Background

We assume familiarity with Bitcoin Script, ECDSA signatures, and DER encoding; for a thorough treatment of these topics, see BINOHASH [2], Sections 2–3. Our scheme builds extensively on the BINOHASH construction, and we strongly recommend reading that paper prior to this one. While we summarize the relevant components in the following sections, many of the deeper design choices and optimizations—particularly around FindAndDelete, minimum-size signatures, and the SIGHASH_SINGLE bug—are explained more fully there.

2.1 Goals

We aim to construct:

1. A **cryptographically strong identifier** of the spending transaction—with both second pre-image and collision resistance—that is readable within Bitcoin Script.

2. A **Lamport signature** over that identifier, verifiable entirely within Script.

2.2 Constraints

For multiple reasons—including the reliance on ECDSA (rather than Schnorr), the FindAndDelete mechanism (removed in SegWit), and the SIGHASH_SINGLE bug (absent in SegWit's sighash algorithm)—neither BINOHASH nor our construction can use Taproot or SegWit. We are therefore restricted to legacy (pre-SegWit) script execution, which imposes severe constraints. The two hardest, and therefore most important, are:

1. **Opcode limit.** The number of non-push opcodes in a script may not exceed 201. Every opcode above `OP_16` counts toward this limit, including opcodes in non-executed branches of `OP_IF`/`OP_ELSE` [2, Section 6].

2. **Script size limit.** The total script size may not exceed 10,000 bytes [2, Section 6].

For a full discussion of additional constraints—including stack element size limits, stack depth, and transaction relay considerations—see BINOHASH [2, Section 6]. Furthermore, as noted in BINOHASH, the script size exceeds the 520-byte limit for P2SH redeem scripts, so the transaction must use a bare script output (scriptPubKey contains the script directly).

These constraints are extremely demanding in the context of our two goals: achieving a cryptographically strong transaction identifier requires a high number of security bits, while the Lamport signature imposes a per-bit overhead in both opcodes and script size.

2.3 Threat Model

2.3.1 Quantum Computing Assumptions

Our threat model assumes an adversary with access to a cryptographically relevant quantum computer. We distinguish between the two classes of cryptographic primitives affected:

Elliptic curve cryptography (catastrophic vulnerability). ECDSA and Schnorr signatures rely on the hardness of the elliptic curve discrete logarithm problem (ECDLP). Shor's algorithm [6] solves ECDLP in polynomial time on a quantum computer, rendering these schemes completely insecure. For the secp256k1 curve used in Bitcoin, a sufficiently large quantum computer could recover private keys from public keys, forge arbitrary signatures, and compute discrete logarithms of any curve point. This is a total break: the adversary gains *unlimited* forgery capability, not merely a reduction in work factor.

Hash functions (moderate weakening). Cryptographic hash functions such as SHA-256, RIPEMD-160, and HASH160 are affected by Grover's algorithm [7], which provides a quadratic speedup for unstructured search. This reduces the effective security of an n-bit hash from 2^n to $2^{n/2}$ for pre-image attacks, and from $2^{n/2}$ to $2^{n/3}$ for collision attacks. Importantly, this is a *weakening*, not a break: a 160-bit hash retains approximately 80 bits of pre-image resistance and approximately 53 bits of collision resistance against a quantum adversary. For our scheme, which relies on RIPEMD-160 (160 bits), this provides a comfortable security margin.

Practical timeline considerations. While the timeline for large-scale fault-tolerant quantum computers remains uncertain, there is broad consensus that breaking elliptic curve cryptography is likely to be feasible significantly earlier than breaking hash functions to a practically exploitable degree. Shor's algorithm requires a number of logical qubits roughly proportional to the key size, while Grover's algorithm—though polynomial in qubits—offers only a quadratic speedup and is difficult to parallelize. This asymmetry motivates our approach: we aim to eliminate *all* reliance on elliptic curve hardness for security, while retaining hash-based primitives whose quantum-reduced security levels remain adequate.

2.3.2 Security Requirements

In the context of quantum-safe transactions, there are two distinct security properties we aim to achieve. They differ significantly in severity and in the assumptions they make about the adversary.

1. Second pre-image resistance (critical). This is the primary security requirement. Suppose an honest spender broadcasts a transaction T whose digest is $D = H(T)$. We require that it is infeasible for a malicious actor—even one equipped with a quantum computer—who observes T and its digest D to construct a different transaction T' such that $H(T') = D$. If such a T' could be found, the attacker could reuse the Lamport signature revealed in T to authorize T', completely breaking the quantum safety of the scheme.

2. Collision resistance (secondary). Here we consider a potentially malicious spender who controls the transaction creation process. We require that the spender cannot find two distinct transactions T and T' such that $H(T) = H(T')$. This property is not essential for the spender's own quantum safety—the spender has no incentive to attack their own transaction—but it prevents equivocation: the spender cannot present one transaction to a counterparty while secretly holding another, equally valid transaction with a different set of outputs. This is particularly relevant for protocols such as BitVM [2] that rely on the uniqueness of the digest for off-chain dispute resolution.

2.4 Binohash Overview

Readers already familiar with Binohash *[2] may skip to Section 3, where we present our modifications.*

At a high level, to create a valid spending transaction with BINOHASH, the spender must complete two steps of proof-of-work:

Step 1: Transaction pinning. Force a W_1-bit PoW puzzle for every change to the spending transaction. This means the spender must perform 2^{W_1} additional work each time any part of the transaction is modified.

Step 2: Compute the Binohash digest. Find a valid hash digest D for the transaction. A valid digest is constructed from two rounds of "n-choose-t" subset selections over signatures embedded in the script—hence the name BINOHASH. The correct combination of signatures results in solving an additional W_2-bit PoW puzzle per round. The selected subsets form the digest D, which is also Lamport-signed within the script.

The Lamport signature uses a HORS (Hash to Obtain Random Subset) scheme [9]: for each of the n possible indices, the locking script contains a hash commitment $H(pre_i)$. To sign a particular subset (the digest), the spender reveals the preimages pre_i for the selected indices. The script verifies each preimage by hashing it and comparing to the commitment. Since preimages are secret until revealed, the spender commits to a specific digest at signing time.

The total work for the honest spender is therefore $2^{W_1} + 2 \cdot 2^{W_2}$. The digest D has $2W_2$ bits of entropy.

Second pre-image resistance. Both steps contribute equally to this property. Given a transaction T with $H(T) = D$, finding a different transaction T' with $H(T') = D$ costs 2^{W_1} per attempt (due to pinning), and the expected number of attempts to match D is 2^{2W_2} (the digest space). The total cost is $2^{W_1} \cdot 2^{2W_2}$.

Collision resistance and the role of pinning. The value of Step 1 (pinning) becomes apparent when we consider collision resistance. Due to the birthday attack, finding two transactions with the same digest costs only 2^{W_2} samples (the square root of the 2^{2W_2} digest space). However, generating each sample still requires 2^{W_1} work for pinning. The total collision cost is therefore $2^{W_1+W_2}$, and the W_1 contribution from pinning is the dominant factor.

Additionally, pinning is extremely cheap in terms of both opcodes and script size, while still providing valuable security. Why not increase W_1 further? Because the pinning cost is imposed on the honest spender as well. The optimal strategy is to set W_1 roughly equal to the work the spender already performs to find the digest, which is approximately $2 \cdot 2^{W_2}$. This balances the cost borne by the spender against the security gained.

2.5 Why a Binomial Structure?

The opcode constraint is extremely severe. Achieving a target security of $\sim$100 bits—with a digest that is also Lamport-signed—within a budget of 201 non-push opcodes is a formidable challenge.

The digest information must be provided by the unlocking script (the witness). If it were supplied as many independent elements, the cost of processing and signing each one would be prohibitive. Conversely, reproducing many elements in the locking script from a few witness-provided values is equally expensive: flow-control opcodes (`OP_IF`/`OP_ELSE`) count toward the 201-opcode limit even in non-executed branches, making dispatch logic impractical.

We are therefore forced to encode the digest in a *small number of witness elements*, and to verify and sign it with correspondingly few operations. This naturally leads to a binomial structure: the witness provides t indices selecting a subset from a pool of n items embedded in

the locking script. The number of possible subsets, $\binom{n}{t}$, grows rapidly even for small t, providing the entropy we need. For example, $\binom{150}{8} \approx 2^{42}$ with just 8 selections.

2.6 Primitives

To perform proof of work over a transaction that is checkable in Script, we need two ingredients:

1. **A method to verify PoW in Script.** Given a candidate solution, the script must be able to check that sufficient work was performed. This is the role of the *signature size puzzle* (Section 2.6.1).

2. **A method to iterate over candidates.** The spender must be able to try many candidates until one satisfies the PoW check. This iteration itself splits into two distinct mechanisms:

   (a) **Iteration outside the script.** The spender modifies transaction parameters—such as `nLocktime`, output ordering, or `OP_RETURN` data—and re-checks the PoW puzzle in Script for each variant. Since these modifications happen outside the script, they are *not* Lamport-signable. This is essentially what is used for *transaction pinning* (Step 1): the spender iterates over transaction variants off-chain, and the script verifies the PoW for whichever variant is broadcast.

   (b) **Iteration inside the script.** Here, different iterations are invoked by different data provided in the unlocking script (witness). Crucially, this data *is* Lamport-signable within the script, which is what makes it suitable for the digest (Step 2). For this to be practical, we need an efficient method to iterate over many candidates with as few opcodes and bytes as possible. This is the role of the *FindAndDelete mechanism* (Section 2.6.2).

2.6.1 Signature Size Puzzle

The signature size puzzle (see, e.g., Bor [4] and Linus [2]) exploits the fact that a DER-encoded ECDSA signature's total byte length is determined by the lengths of its two components, r and s. If r is fixed, then the total signature size is governed solely by the length of s, and finding an s shorter than a target threshold requires brute-force search—a proof of work. The way to effectively fix r is by imposing an extreme constraint on the total signature size, practically forcing the spender to use the smallest known r value; any larger r would make it infeasible to meet the size target.

Concretely, the puzzle works as follows:

1. **Force a fixed r via size constraint.** The locking script enforces a signature size threshold so tight that only signatures using the smallest known r value—$r_{\min}$, whose x-coordinate is approximately 21 bytes—can satisfy it. The discrete logarithm of the corresponding curve point is known, allowing anyone to compute signatures with this specific r.

2. **Grind for small s.** To solve the puzzle, the spender searches over transactions (or nonces) until the resulting ECDSA signature—computed with the known $r_{\min}$—has an s value short enough that the total DER-encoded signature falls below the target size.

3. **Verify via `OP_SIZE`.** Bitcoin Script cannot parse the internal structure of a signature, but it can measure its total byte length using `OP_SIZE`. By checking that the signature is shorter than a threshold, the script implicitly verifies that sufficient grinding was performed.

This primitive is elegant and cheap: verification requires only `OP_SIZE` and an `OP_LESSTHAN` or `OP_EQUALVERIFY`—just 2–3 opcodes.

2.6.2 FindAndDelete and Sighash Variation

The signature size puzzle gives us a way to verify proof of work on a single signature. But for Step 2—computing the digest—we need to iterate over many candidates *inside* the script, in a way that is Lamport-signable. The mechanism that makes this possible is *FindAndDelete*—a legacy quirk of how `OP_CHECKMULTISIG` computes sighashes in pre-SegWit Bitcoin.

When `OP_CHECKMULTISIG` verifies a signature, it first computes the sighash—the double-SHA256 of the transaction data that the signature commits to. Crucially, the sighash is computed over the *scriptCode*, which is the locking script with all signature data removed via a process called FindAndDelete: before hashing, the script interpreter scans the scriptCode and deletes every occurrence of each signature being verified.

Binohash exploits this as follows. The locking script embeds n dummy signatures (e.g., $n = 150$). The spender selects a subset of t of them and passes them to `OP_CHECKMULTISIG`. FindAndDelete removes exactly those t signatures from the scriptCode before computing the sighash. A different subset of t signatures leads to a different scriptCode, and therefore a different sighash.

This gives the spender $\binom{n}{t}$ distinct sighashes to try—all from a single, fixed transaction. For each sighash, the spender checks whether the corresponding PoW puzzle is satisfied. The subset that solves the puzzle becomes the digest.

The dummy signatures are constructed to be transaction-independent using the `SIGHASH_SINGLE` bug: when `SIGHASH_SINGLE` is used for an input whose index exceeds the number of outputs, Bitcoin sets the signed message to $z = 1$, regardless of the transaction content [2]. This allows the dummy signatures and their corresponding public keys (recovered via ECDSA key recovery with $z = 1$) to be precomputed once and reused across all transactions.
FindAndDelete was removed in SegWit (BIP 143 [5]), which is one of the reasons this construction requires legacy (pre-SegWit) script execution.

2.7 The Binohash Scheme

We now describe how the primitives above are assembled into the full Binohash construction. For a complete treatment, see [2].

2.7.1 Script Verification (Locking Script)

Step 1: Transaction pinning. The locking script verifies four signature size puzzles, each using a different sighash flag combination, to cover all possible transaction modification vectors.[4] The four distinct puzzles can be enforced either via `OP_CODESEPARATOR`—which changes the scriptCode used for sighash computation at each invocation—or via a more compact trick involving carefully chosen public keys; see [2, Appendix A]. Additionally, the `SIGHASH_SINGLE` flag is explicitly excluded via a check, preventing an attacker from using the $z = 1$ bug to bypass the PoW. This step costs 13 opcodes.

Step 2: Digest verification (two rounds). Each round performs the following:

1. **Lamport signature verification.** The spender reveals HORS preimages for the selected subset indices. The script verifies each preimage against its corresponding hash commitment (embedded in the locking script).

[4]Since Bitcoin Script cannot enforce which sighash flag the spender uses, the spender could choose any of the six available modes. Two of these are `SIGHASH_SINGLE` variants, which are eliminated by an explicit check that detects the $z = 1$ bug. The remaining four modes—`ALL`, `NONE`, `ALL|ANYONECANPAY`, and `NONE|ANYONECANPAY`—are each assigned a separate puzzle. By requiring all four puzzles to be solved, we ensure that regardless of which mode the spender uses, every transaction field is committed to by at least one puzzle. See [2, Section 5] for details.

2. **Dummy signature subset selection.** Based on the indices provided in the witness, the script uses `OP_ROLL` to select t dummy signatures from the pool of n.

3. **Multisig verification with puzzle signature.** The selected dummy signatures, together with the puzzle signature, are verified via `OP_CHECKMULTISIG`. FindAndDelete removes the selected dummies from the scriptCode, so the puzzle signature's sighash reflects the specific subset chosen.

4. **PoW check.** The puzzle signature's size is checked via `OP_SIZE` against the W_2 target.

2.7.2 Spending Procedure (Off-Chain)

To find a valid spending transaction, the spender performs the following:

1. **Solve the pinning puzzle.** Iterate over transaction parameters (e.g., `nLocktime`, output ordering) until all four pinning puzzles are satisfied. The transaction is now fixed.

2. **Search for the digest.** For each round, iterate over the $\binom{n}{t}$ possible dummy signature subsets. Each subset produces a different scriptCode via FindAndDelete, yielding a different sighash. Check whether the resulting puzzle signature satisfies the W_2-bit PoW target.

3. **Record the digest.** The subset that solves the puzzle becomes the digest for that round. Reveal the corresponding HORS preimages (Lamport signature).

4. **Repeat for both rounds.** If both rounds find a solution—the full digest is complete. If either round fails (none of the $\binom{n}{t}$ subsets solve the puzzle), return to step 1 with a modified transaction and try again.

2.8 Open Issues

Before presenting our scheme, we identify two issues with the BINOHASH construction as described above. The first is a fundamental quantum safety vulnerability—a direct consequence of the catastrophic weakness of elliptic curve cryptography under quantum computing (Section 2.3.1). The second is a sighash-related attack that, while not specific to quantum computing, must be addressed for standalone use outside of BitVM.

2.8.1 The Signature Size Puzzle Is Not Quantum Safe

The signature size puzzle (Section 2.6.1) relies on the assumption that the spender is forced to use the smallest known r value, $r_{\min}$, when constructing the puzzle signature. This assumption holds classically: finding a different curve point with a comparably small x-coordinate requires approximately 2^{96} work [2, Section 4].

As discussed in Section 2.3.1, a quantum computer running Shor's algorithm [6] can compute the discrete logarithm of *any* elliptic curve point in polynomial time. This includes points with very small x-coordinates—in particular, the point with $r = 1$ (or the smallest valid r on the curve). Armed with this value, the adversary can produce signatures with a shorter r than $r_{\min}$, allowing them to satisfy the `OP_SIZE` check with significantly less work on s—or none at all.

Concretely: the PoW difficulty is determined by how short s must be to bring the total signature size under the target threshold. If the target requires W bits of grinding on s when using $r_{\min}$, then an adversary who finds a curve point with an r value just $\lceil W/8 \rceil$ bytes shorter than $r_{\min}$ gains enough headroom to satisfy the size check with virtually any s, reducing the

PoW to zero. For typical targets of $W \approx 40$, this means finding an r just 5 bytes shorter than $r_{\min}$ suffices to break the puzzle entirely.[5]

Note that this vulnerability is fundamentally different from the weakening of hash functions by Grover's algorithm. While Grover reduces hash security by a square root factor, the result remains computationally hard (e.g., RIPEMD-160 retains $\sim 2^{80}$ pre-image resistance). Shor's attack on the signature size puzzle, by contrast, is a *total break*: the PoW guarantee collapses entirely, regardless of the target difficulty.

2.8.2 Sighash Flag Uncertainty

The second issue concerns the sighash flag byte appended to each signature. As discussed in Section 2.7, the pinning stage uses four separate puzzles to cover all non-`SINGLE` sighash modes. However, this protection applies only to the pinning signatures.

The puzzle signatures in Step 2 (the digest rounds) are produced by the spender via PoW grinding. The spender is free to choose any sighash flag for these signatures. In particular, they could use `ANYONECANPAY|NONE`, which commits to neither the outputs nor the other inputs of the transaction. As Linus observes [2, Section 7.1], this allows an attacker to take a valid puzzle signature—which committed to a specific set of outputs—and reuse it in a completely different transaction with different outputs, provided the input being spent remains the same.

In the BitVM context, this attack is mitigated by off-chain verification: the BitVM protocol can check the sighash mode and reject non-compliant transactions [2, Section 7.1]. However, for standalone use—where we want the on-chain script alone to guarantee transaction integrity—this mitigation is insufficient. The script must enforce the sighash flag directly, or the construction must be redesigned so that the sighash flag is inherently controlled.

3 Our Scheme

We now present our modifications to BinoHash. The central change is replacing the signature size puzzle (Section 2.6.1) with a *hash-to-sig* puzzle whose security relies only on the pre-image resistance of cryptographic hash functions—not on any elliptic curve hardness assumption. This directly addresses the quantum vulnerability identified in Section 2.8.1.

3.1 Hash-to-Sig Puzzle

3.1.1 Core Observation

The replacement puzzle is based on the following observation, first noted by Linus [8]:

Observation. *The output of a cryptographic hash function can, with some probability, be a valid DER-encoded ECDSA signature.*

A DER-encoded ECDSA signature has a rigid structure: `30 [len] 02 [r-len] [r] 02 [s-len] [s] [sighash]`, where the length fields must be internally consistent and both r and s must be positive integers with no unnecessary leading zeros (i.e., MSB < 128). The trailing sighash byte can be any value at the consensus level, since `SCRIPT_VERIFY_STRICTENC` is a relay policy rule, not a consensus rule [2, Section 2.4.2]. A random byte string satisfying all of these constraints simultaneously is a rare event—but not impossibly rare.

This gives us a new type of PoW puzzle: *grind for an input whose hash output happens to be a valid DER signature.* The work required is determined by the probability that a random output of the hash function satisfies the DER constraints.

[5] As noted in Section 1.1.1, once the discrete logarithm for the smallest valid r is known, the puzzle *could* be redesigned around it. But all existing transactions created using $r_{\min}$ would remain vulnerable.

3.1.2 SHA-256 Variant

Linus [8] first described this approach using SHA-256. A random 32-byte string is a valid DER-encoded ECDSA signature with probability approximately $2^{-45.4}$ at the consensus level (where the sighash byte is unconstrained). The dominant constraints are structural: the first byte must be the compound tag `0x30`, the second byte must encode the correct total length, the integer tags `0x02` must appear at the correct positions, and the r-length and s-length fields must be consistent with each other and with the total signature length. These structural constraints account for the vast majority of the difficulty. The positivity constraints on r and s (MSB < 128) contribute only $\sim$2 additional bits.

Verification in Script is straightforward: compute `OP_SHA256` on the input, then use `OP_CHECKSIG` (or `OP_CHECKMULTISIG`) to verify the result as a signature. If the hash output is not valid DER, the signature check fails.

3.1.3 RIPEMD-160 Variant ($\sim 2^{46}$ work)

As noted in BINOHASH [2, Appendix D], the same approach works with RIPEMD-160, producing 20-byte outputs. A random 20-byte string is a valid DER-encoded ECDSA signature with probability approximately $2^{-46.4}$. The probability analysis is similar to the SHA-256 variant—both are dominated by the same structural constraints—with the 20-byte output having slightly fewer valid (r-len, s-len) combinations than the 32-byte output.

Verification in Script uses `OP_RIPEMD160` instead of `OP_SHA256`. The two variants present a minor tradeoff: SHA-256 has a slightly higher DER probability ($\sim 2^{-45.4}$ vs. $\sim 2^{-46.4}$), reducing off-chain work by roughly $2\times$, but the larger puzzle target slightly weakens the security analysis. The per-candidate GPU cost is essentially identical, as both are single-block hash operations dominated by the EC recovery step. A hybrid approach—using RIPEMD-160 for pinning and SHA-256 for the digest rounds, or vice versa—is also possible. For simplicity, we adopt RIPEMD-160 throughout.

3.1.4 Quantum Safety

Unlike the signature size puzzle, the hash-to-sig puzzle does not depend on any elliptic curve hardness assumption. The work required is determined entirely by the pre-image resistance of the hash function: the spender must find an input whose hash output satisfies the DER structural constraints. A quantum adversary gains only Grover's quadratic speedup [7]:

- RIPEMD-160: 2^{46} classically $\rightarrow \sim 2^{23}$ with Grover (on the puzzle alone).
- SHA-256: 2^{45} classically $\rightarrow \sim 2^{22.5}$ with Grover.

3.1.5 ECDSA Key Recovery

A standard property of ECDSA—used extensively in both BINOHASH and our scheme—is that given a valid signature (r, s) and the signed message hash z, one can *recover* the public key that produced the signature, without knowing the private key. This is known as ECDSA public key recovery.

The recovery works as follows. The signature component r is the x-coordinate of a curve point $R = k \cdot G$ (where k is the ephemeral nonce). Given r, we can reconstruct the candidate point R (up to a small number of possibilities, typically 2, corresponding to the two y-coordinates for a given x). From the ECDSA verification equation:

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

we can solve for the public key $Q = d \cdot G$:

$$Q = r^{-1}(s \cdot R - z \cdot G)$$

This computation involves only public values—r, s, z, and the curve parameters—and can be performed by anyone. No private key or secret information is required.

Crucially, key recovery is *not* a security assumption—it is a public computation that anyone can perform, including a quantum adversary.

3.2 Using the Hash-to-Sig Puzzle in Script

We now describe how the hash-to-sig puzzle and ECDSA key recovery are combined to create a quantum-safe PoW check that is fully verifiable in Bitcoin Script.

Step 1: Hardcoded signature with a fixed sighash flag. The locking script contains a hardcoded signature sig_{nonce} with a predetermined sighash flag—in our case, `SIGHASH_ALL`. Because the signature is embedded directly in the script (rather than provided by the spender), the sighash flag is fixed at script creation time. This immediately addresses the sighash flag uncertainty identified in Section 2.8.2: the spender cannot choose a different flag.

Step 2: Derive and verify the transaction-bound key. Given sig_{nonce} and the spending transaction, there is a deterministic function—ECDSA key recovery (Section 3.1.5)—that produces the corresponding public key:

$$key_{\text{nonce}} = \text{Recover}(sig_{\text{nonce}}, z)$$

where z is the sighash of the transaction. The spender computes key_{nonce} off-chain and provides it in the witness. The script verifies the binding via `OP_CHECKSIGVERIFY`: if key_{nonce} does not match (sig_{nonce}, z), verification fails. Since z depends on the transaction content, key_{nonce} is uniquely determined by the transaction—changing any part of the transaction changes z, and therefore changes key_{nonce}.

Step 3: Hash-to-signature puzzle check. The script computes $sig_{\text{puzzle}} =$ `OP_RIPEMD160`(key_{nonce}). For the puzzle to be solved, this 20-byte output must be a valid DER-encoded ECDSA signature—an event with probability $\sim 2^{-46}$. To verify this in Script, the spender provides a second public key key_{puzzle} (derived via ECDSA key recovery from $(sig_{\text{puzzle}}, z')$, where z' is the relevant sighash). The script verifies $(sig_{\text{puzzle}}, key_{\text{puzzle}})$ via `OP_CHECKSIGVERIFY`: if sig_{puzzle} is not valid DER, this check fails.

Result. If both verifications pass, the puzzle is solved. The chain of trust is:

1. sig_{nonce} is hardcoded $\rightarrow$ its sighash flag is fixed (`SIGHASH_ALL`).
2. $key_{\text{nonce}} = \text{Recover}(sig_{\text{nonce}}, z) \rightarrow$ bound to this specific transaction.
3. $sig_{\text{puzzle}} = \text{RIPEMD-160}(key_{\text{nonce}}) \rightarrow$ inherits the transaction binding.
4. key_{puzzle} verifies that sig_{puzzle} is valid DER $\rightarrow$ puzzle check complete.

Fixed puzzle difficulty. An important distinction between the hash-to-sig puzzle and the signature size puzzle is that the hash-to-sig puzzle has a *fixed* difficulty: it is determined entirely by the hash function and the target primitive (DER-encoded ECDSA signature). For RIPEMD-160, this is $\sim 2^{46}$; for SHA-256, $\sim 2^{45}$. There is no knob to tune.

By contrast, the signature size puzzle allows flexible difficulty: the script creator can choose any size threshold, adjusting the PoW target continuously. This makes script planning more constrained in our case: the entropy generated by the binomial subset selection (i.e., $\binom{n}{t}$) must be matched to the fixed 2^{46} or 2^{45} puzzle target. A mismatch—where $\binom{n}{t}$ is significantly smaller or larger than the puzzle target—results in either excessive transaction grinding (if too small) or wasted entropy (if too large). We analyze the impact on off-chain work in Section 4.

3.3 Full Scheme

We now describe the complete construction, replacing the signature size puzzle in BINOHASH with the hash-to-sig puzzle throughout.

3.3.1 Transaction Pinning

Transaction pinning ensures that modifying any part of the spending transaction requires solving the hash-to-sig puzzle anew ($\sim 2^{46}$ work). In BINOHASH, this requires four signature size puzzles at a cost of 13 opcodes. In our scheme, a single application of the hash-to-sig puzzle suffices, since the hardcoded `SIGHASH_ALL` flag commits to all transaction fields.

The pinning script is a direct instantiation of the puzzle described in Section 3.2:

```
// Witness: <key_puzzle> <key_nonce>       (key_nonce on top)

<sig_nonce>                              // hardcoded, SIGHASH_ALL (data
   push)
OP_OVER                           // [1] copy key_nonce
OP_CHECKSIGVERIFY                 // [2] verify (sig_nonce, key_nonce)
OP_RIPEMD160                      // [3] key_nonce -> sig_puzzle
OP_SWAP                           // [4] get key_puzzle on top
OP_CHECKSIGVERIFY                 // [5] verify (sig_puzzle, key_puzzle)
```

Matching this to the steps in Section 3.2:

- **Step 1** (fixed sighash): `<sig_nonce>` is hardcoded with `SIGHASH_ALL`—the flag cannot be changed by the spender.
- **Step 2** (derive and verify key): `OP_OVER` copies `key_nonce` (so it survives for use in Step 3), then `OP_CHECKSIGVERIFY` verifies that `key_nonce` matches `(sig_nonce, sighash)`. This binds `key_nonce` to this specific transaction.
- **Step 3** (hash-to-sig check): `OP_RIPEMD160` hashes `key_nonce` to produce `sig_puzzle`. Then `OP_SWAP` brings `key_puzzle` on top, and the second `OP_CHECKSIGVERIFY` verifies that `sig_puzzle` is a valid DER signature.

Total cost: **5 opcodes**—a saving of 8 compared to BINOHASH's 13-opcode pinning. The pinning is self-contained: after both checks, nothing remains on the stack.

3.3.2 Digest Rounds

The hardcoded sig_{nonce} is verified inside `CHECKMULTISIG` alongside the selected dummy signatures. FindAndDelete removes the selected dummies from the scriptCode before computing the sighash, so each subset produces a different sighash. This gives us $\binom{n}{t}$ distinct sighash candidates from a single fixed transaction. The purpose of the selection stage is to translate the

digest provided in the witness into the specific subset that solves the puzzle, after verifying the Lamport signature on it.

Each digest round follows the same high-level structure as BINOHASH (Section 2.7) and proceeds in three stages:

Stage 1: Subset selection and Lamport verification. This stage is identical to BINOHASH. The spender provides t indices in the witness, each selecting one dummy signature from the pool of n. For each selected index, the script verifies the corresponding HORS preimage against its hash commitment—this constitutes the Lamport signature over the digest. The selected dummy signatures are left on the stack for use in Stage 3. The detailed script for this stage is given in Appendix A.

Stage 2: Puzzle signature derivation. The spender provides key_{nonce} in the witness. The script duplicates it and computes $sig_{\text{puzzle}} = \texttt{OP_RIPEMD160}(key_{\text{nonce}})$. One copy of key_{nonce} is consumed by the hash; the other is preserved for use as a public key in Stage 3. The RIPEMD-160 output sig_{puzzle} is left on top of the stack as the puzzle signature.

Stage 3: Verification. First, the script verifies that sig_{puzzle} is a valid DER signature by rolling key_{puzzle} from the witness and executing `OP_CHECKSIGVERIFY` on the pair $(sig_{\text{puzzle}}, key_{\text{puzzle}})$. This check must happen *before* `CHECKMULTISIG`—not inside it—because placing sig_{puzzle} inside `CHECKMULTISIG` would create a circular dependency: the sighash determines key_{nonce}, which determines sig_{puzzle}, which would in turn affect the sighash.

After sig_{puzzle} is consumed by `CHECKSIGVERIFY`, the remaining key_{nonce} copy is on the stack above the dummy signatures. The script then executes a $(t+1)$-of-$(t+1)$ `CHECKMULTISIG`, verifying the t selected dummy signatures and sig_{nonce} (hardcoded, with `SIGHASH_ALL`—resolving the sighash flag vulnerability of Section 2.8.2) against their corresponding public keys. key_{nonce} serves as the public key for sig_{nonce}.

The critical point is that sig_{nonce} goes through `CHECKMULTISIG`, where FindAndDelete modifies the scriptCode before computing sighashes. The sighash that sig_{nonce} is verified against therefore depends on *which* dummy signatures were selected. This means $key_{\text{nonce}} = \text{Recover}(sig_{\text{nonce}}, z)$ is different for each subset, and consequently $sig_{\text{puzzle}} = \text{RIPEMD-160}(key_{\text{nonce}})$ is different as well. The subset that happens to produce a valid DER output becomes the digest for this round.

The script for Stages 1–3:

```
// Witness (bottom to top):
//   <key_puzzle> <key_nonce>
//   <pub_t> ... <pub_1>
//   <preimage_t> ... <preimage_1>
//   <index_t> ... <index_1>

// Locking script:
<H(pre_n)> ... <H(pre_1)>      // n HORS commitments
<sig_n> ... <sig_1>             // n dummy sigs (SIGHASH_SINGLE)
OP_0                            // CHECKMULTISIG dummy
<sig_nonce>                       // hardcoded (SIGHASH_ALL)

// Stage 1: selection + Lamport verification
...                             // (see Appendix A)
// After Stage 1: selected dummy sigs + sig_nonce + OP_0 on stack

// Stage 2: derive puzzle sig
{pos} OP_ROLL                // roll key_nonce from witness
OP_DUP                       // copy (one for RIPEMD, one for pubkey)
OP_RIPEMD160                 // key_nonce -> sig_puzzle
```

```
// Stage 3a: verify sig_puzzle is valid DER
{pos} OP_ROLL              // roll key_puzzle from witness
OP_CHECKSIGVERIFY          // verify (sig_puzzle, key_puzzle)

// Stage 3b: CHECKMULTISIG (t+1)-of-(t+1)
OP_{t+1}                   // M = t+1 signatures
OP_2 OP_ROLL               // move key_nonce copy into pubkey zone
{pos} OP_ROLL (x t)        // roll t dummy pubkeys from witness
OP_{t+1}                   // N = t+1 public keys
OP_CHECKMULTISIG           // verify t dummies + sig_nonce
```

Here key_{puzzle} is derived by the spender via ECDSA key recovery from (sig_{puzzle}, z)—the same mechanism used throughout the scheme.

The per-round overhead compared to BINOHASH is +4 opcodes for the puzzle and `CHECKMULTISIG` setup. Combined with the 8-opcode saving from pinning (Section 3.3.1), the net overhead for a two-round scheme is **exactly zero**.

Two rounds. The full scheme uses two independent rounds, each with its own pool of n dummy signatures, HORS commitments, and hardcoded sig_{nonce}. The full digest is the concatenation of the two rounds' selected subsets. The second round's script is identical in structure to the first; only the embedded data differs.

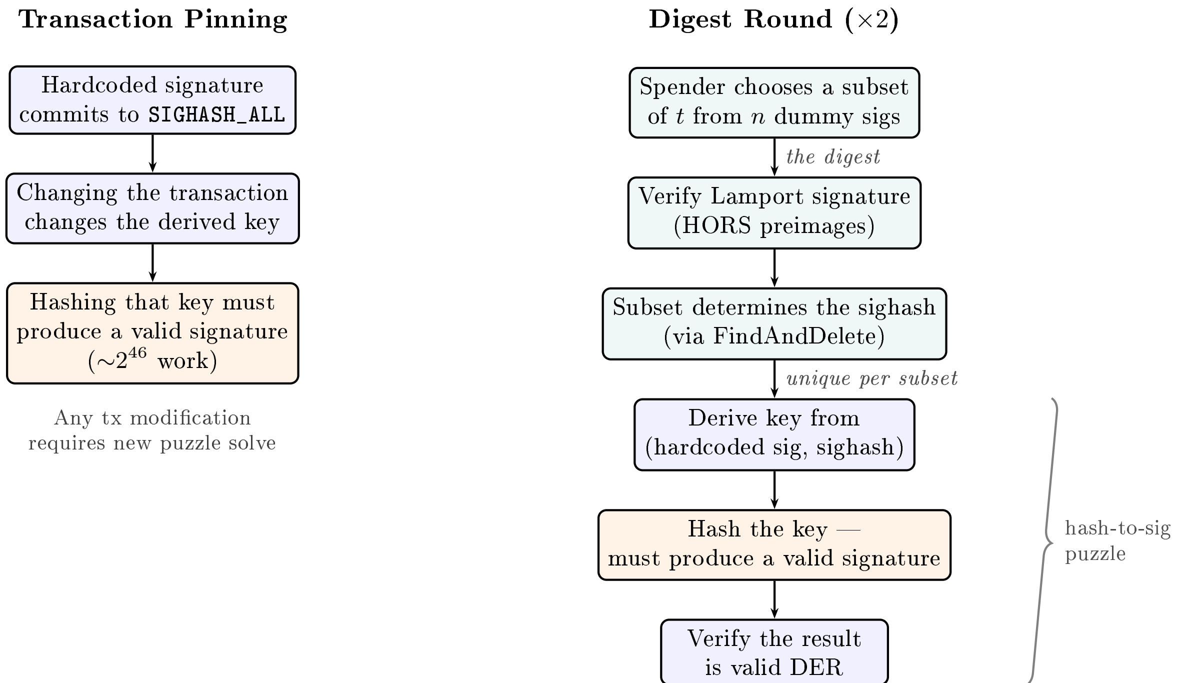

Figure 1: The QSB scheme. **Left:** transaction pinning ensures each transaction attempt costs $\sim 2^{46}$ work. **Right:** each digest round selects a subset (the digest), verifies its Lamport signature, then checks the hash-to-sig puzzle—the subset that produces a valid RIPEMD-160 output is the solution. Two independent rounds are executed.

Bonus Keys. An optimization that reduces off-chain cost by adding extra selections that participate in FindAndDelete but skip HORS verification. Bonus keys cost only 3 opcodes per selection (vs. 9 for signed), and increase the effective subset count to match the 2^{46} puzzle target. We analyze bonus keys and their security implications in Section 4.5.

3.4 Spending Procedure (Off-Chain)

To spend a QSB-protected UTXO, the spender performs the following off-chain procedure:

1. **Construct the transaction.** Set up the desired inputs, outputs, and other transaction fields.

2. **Solve the pinning puzzle.** Vary any free transaction parameter that is committed to by `SIGHASH_ALL`—such as `nLocktime`, `nSequence`, output amounts, change address, or `OP_RETURN` data in an extra output—until RIPEMD-160(key_{nonce}) is a valid DER signature, where key_{nonce} is derived via ECDSA key recovery from the hardcoded sig_{nonce} and the current sighash. Any parameter that changes the sighash is equally valid; the choice is an implementation detail (see [2, Section 5.2] for a discussion). This requires $\sim 2^{46}$ attempts. The transaction is now fixed.

3. **Search for the digest (per round).** Iterate over all $\binom{n}{t}$ dummy signature subsets. For each subset, FindAndDelete produces a different scriptCode and therefore a different sighash. Recover key_{nonce} from (sig_{nonce}, sighash) and compute RIPEMD-160(key_{nonce}). If the output is a valid DER signature, this subset is the digest for the round.

4. **Repeat for both rounds.** If both rounds find a valid subset, the full digest is determined. Reveal the corresponding HORS preimages (Lamport signature) in the witness. If either round fails—none of the $\binom{n}{t}$ subsets produces a valid DER output—return to step 1 with a different transaction and try again.

3.5 Operational Architecture

A key operational advantage of QSB is the clean separation between the computationally intensive puzzle search and the security-sensitive signing operations. The search involves only public computations—ECDSA key recovery, RIPEMD-160 hashing, and sighash evaluation—none of which require private keys or secret data. This allows the search to be safely outsourced to untrusted hardware (e.g., rented GPU clusters), while all secrets remain on the spender's personal device.

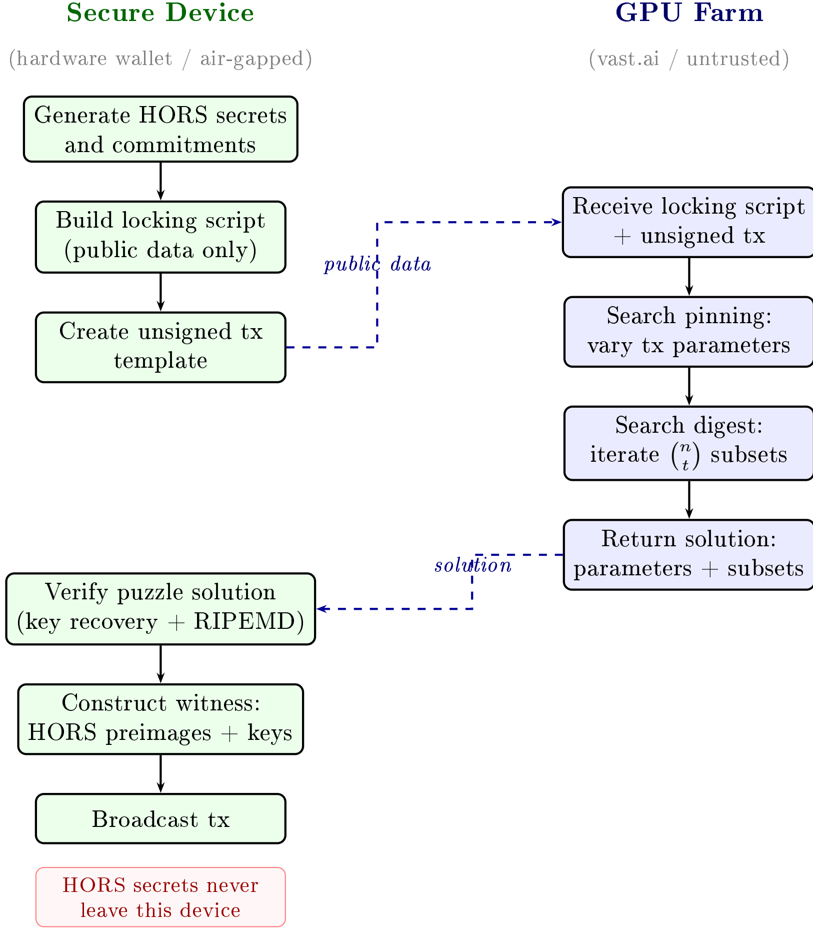

Figure 2: Operational architecture. The GPU farm performs only public computations (key recovery, hashing). All secrets—HORS preimages—remain on the secure device. The spender verifies the solution locally before constructing the witness and broadcasting.

What the GPU farm receives. The locking script (containing HORS commitments, dummy signatures, and hardcoded sig_{nonce} values) and an unsigned transaction template. All of this is public data.

What the GPU farm computes. For each candidate (transaction parameter variant or subset): compute the sighash, recover key_{nonce} via ECDSA key recovery, compute RIPEMD-160, and check DER validity. These are all public operations—no private key or secret is involved.

What the GPU farm returns. The winning transaction parameters and subset indices. The spender's device independently verifies the solution (a single key recovery + RIPEMD-160 per round), then constructs the full witness by revealing the HORS preimages for the winning subsets.

Polyglot digest optimization. BINOHASH [2, Appendix D] proposes a "polyglot digest" optimization that merges dummy signatures and HORS commitments into a single element: grind a preimage P_i such that RIPEMD-160(P_i) is a valid DER signature. Then P_i serves as the HORS preimage, while its hash serves as the dummy signature. This eliminates the need for separate dummy signature pushes, saving ∼10 bytes and 2 opcodes per selection—enabling higher t values (e.g., $t = 10$) or smaller n within the same budget. The paper acknowledges two costs: (1) a one-time off-chain grinding cost of $\sim 2^{53.5}$ RIPEMD-160 hashes to prepare the polyglot elements

(pure hashing, no sighash computation), and (2) the polyglot preimages are HORS secrets, so the grinding must be performed on trusted hardware. In our operational architecture, we require that all secrets remain on the spender's secure device. If the polyglot grinding exceeds that device's capacity and must be outsourced to a GPU farm, the farm would learn the HORS signing keys. We therefore do not adopt this optimization, though it remains a promising direction if the trusted-hardware constraint can be relaxed.

Transaction broadcast. QSB transactions are consensus-valid but non-standard: bare scripts exceeding the default relay policy limits are not propagated by standard Bitcoin Core nodes. As with BINOHASH [2, Section 6], broadcasting requires submitting the transaction directly to a mining pool that accepts non-standard transactions, such as via a private mempool service (e.g., Marathon's Slipstream).

4 Security and Cost Analysis

We analyze the security of our scheme for the baseline configuration: $n = 150$, $t = 8$ signed selections per round, two rounds. The puzzle target is fixed at $\sim 2^{46}$ (RIPEMD-160; precisely $2^{46.4}$). Throughout this section, all security levels are rounded down to the nearest integer.

4.1 Baseline: $t = 8$, $t = 8$

Property	Value
Non-push opcodes	197 / 201 (4 spare)
Script size	$\sim$9,650 / 10,000 bytes
Subsets per round	$\binom{150}{8} \approx 2^{42.3}$
Puzzle target per round	$\sim 2^{46}$ (RIPEMD-160)
Digest size	$2 \times 42.3 = 84.5$ bits
Tx grinds	$\left(2^{46}/2^{42.3}\right)^2 \approx 2^{7.5}$ ($\approx$180 attempts)
Honest spender work	$\sim 2^{53.5}$
Second pre-image resistance	$\sim 2^{138}$
Collision resistance	$\sim 2^{88}$

4.2 Opcode and Script Size Budget

Opcodes. The pinning stage costs 5 non-push opcodes. Each digest round costs:

$$11t + 8$$

where t is the number of *signed* selections per round (with no bonus keys). The breakdown is: $9t$ for the selection loop (9 opcodes per selection) and $2t + 8$ for the puzzle derivation, `CHECKSIGVERIFY`, and `CHECKMULTISIG` setup. For two rounds with t_1 and t_2 signed selections, the total is:

$$5 + (11t_1 + 8) + (11t_2 + 8) = 11(t_1 + t_2) + 21 \leq 201$$

When bonus keys are used, the per-round formula generalizes to $9t_{\text{signed}} + 3t_{\text{bonus}} + 2t_{\text{total}} + 8$ (see Section 4.5).

Script size. The pinning stage occupies approximately 15 bytes (a 9-byte signature push plus 5 opcode bytes). Each digest round occupies approximately:

$$31n + 13t + 20 \text{ bytes}$$

The dominant term is $31n$: each of the n pool entries contributes a 20-byte HORS commitment (plus 1-byte push opcode) and a 9-byte dummy signature (plus 1-byte push opcode). The remaining terms account for the selection logic, puzzle verification, and `CHECKMULTISIG` setup. For two rounds, the total is approximately $62n + \text{overhead} \leq 10{,}000$.

At $n = 150$, each additional pool entry costs ∼62 bytes across both rounds, while adding ∼0.08 bits of digest entropy per entry.

4.3 Security and Grinding Analysis

Second pre-image resistance. An attacker who observes a valid spending transaction must find a different transaction that produces the same digest—the same signed indices in both rounds. Each transaction attempt costs 2^{46} (the pinning puzzle). For each pinned transaction, the probability that a specific t-subset solves the round puzzle is 2^{-46}. Since both rounds must independently match, the probability per pinned attempt is 2^{-92}. The total cost is therefore $2^{46} \times 2^{92} = 2^{138}$.

Collision resistance. A malicious spender who controls the transaction creation process aims to find two transactions with the same digest. The digest space is $2^{84.5}$. By the birthday bound, $\sim 2^{42.3}$ samples suffice to find a collision. Each sample costs 2^{46} (pinning), giving a total collision cost of $2^{46+42.3} \approx 2^{88}$.

Honest spender: subset mismatch. The subsets per round ($\binom{150}{8} \approx 2^{42.3}$) fall short of the 2^{46} puzzle target by a factor of $2^{3.7}$. This means each round has only a $2^{-3.7}$ probability of containing a valid subset for a given pinned transaction. Since both rounds must succeed, the spender needs on average $\sim 2^{7.5} \approx 180$ pinned transaction attempts. Each attempt requires iterating over all $\binom{150}{8}$ subsets per round, giving a total work of $\sim 2^{53.5}$ RIPEMD-160 evaluations. This mismatch motivates the bonus key optimization (Section 4.5), which increases the effective subset count to $\binom{150}{9} \approx 2^{46.2}$ and eliminates the grinding overhead entirely.

4.4 Quantum Security

We analyze the security of the baseline configuration under quantum adversaries, distinguishing between two threat levels introduced in Section 2.3.1.

Shor's algorithm only (ECDSA broken, hash functions unaffected). This is the primary threat our scheme is designed to resist. A quantum adversary running Shor's algorithm can compute discrete logarithms and forge ECDSA signatures, but cannot speed up hash function evaluation.

Under this model, the security of our scheme is *identical* to the classical case:

Property	**Classical**	**Shor only**
Pinning cost	2^{46}	2^{46} (hash-based)
Second pre-image	2^{138}	2^{138}
Collision	2^{88}	2^{88}

This is the central result: because our puzzle relies on RIPEMD-160 pre-image resistance rather than ECDSA hardness, Shor's algorithm provides *no advantage* to the attacker. The scheme achieves full classical security even against a quantum adversary with unbounded access to Shor's algorithm.

Grover's algorithm (quadratic speedup on hash search). Grover's algorithm [7] provides a quadratic speedup for unstructured search problems. Applied to our scheme, it reduces the effective security:

Property	Classical	With Grover
Pinning cost	2^{46}	$\sim 2^{23}$
Second pre-image	2^{138}	$\sim 2^{69}$
Collision	2^{88}	$\sim 2^{65}$

For second pre-image resistance, Grover provides a square-root speedup on the full search: the classical cost of 2^{138} becomes $\sim 2^{69}$. For collision resistance, the pinning cost per sample drops from 2^{46} to $\sim 2^{23}$, while the birthday bound remains at $2^{42.3}$ samples, giving $2^{23+42.3} \approx 2^{65}$.

Practical feasibility of Grover. While these reductions appear significant on paper, Grover's algorithm faces fundamental limitations that make it far less threatening than Shor's in practice. Grover cannot be efficiently parallelized: running k quantum computers in parallel provides only a $\sqrt{k}$ speedup, not a k-fold speedup [7]. Moreover, implementing the search oracle as a quantum circuit requires all intermediate state—including the sighash computation (SHA-256d) and key recovery—to be represented in reversible logic, demanding significant qubit overhead.

More importantly, Shor's algorithm for the secp256k1 curve requires on the order of a few thousand logical qubits and is expected to be feasible well before Grover becomes practical against targets of 2^{46} or higher. This asymmetry is precisely why eliminating reliance on ECDSA hardness—while retaining hash-based security—is the right design strategy. Note that with bonus keys (Section 4.5), the recommended Config A trades some security for reduced cost, achieving 2^{118} pre-image and 2^{78} collision resistance.

4.5 Bonus Keys

Motivation. The baseline configuration ($t = 8$, $t = 8$) achieves strong security but incurs significant off-chain cost. The problem is the *subset mismatch*: each round offers $\binom{150}{8} \approx 2^{42.3}$ subsets, but the RIPEMD-160 puzzle target is fixed at $\sim 2^{46}$. The gap of $2^{3.7}$ means that for a given pinned transaction, each round has only a $2^{-3.7} \approx 8\%$ chance of containing a valid subset. Since both rounds must succeed, the spender needs $\sim 2^{7.5} \approx 180$ pinned transaction attempts, each requiring 2^{46} work.

The ideal remedy is to increase the subset count to $\binom{150}{t} \geq 2^{46}$. This requires $t \geq 9$, since $\binom{150}{9} \approx 2^{46.2}$. However, adding a fully signed selection costs 9 opcodes, and the opcode budget is already tight at 197 / 201.

The bonus key optimization. A *bonus key* is a subset selection that participates in FindAndDelete—contributing to $\binom{n}{t_{\text{total}}}$ and increasing the number of sighash candidates—but skips HORS verification. Because the HORS check (DUP + ADD + ROLL + ROLL + HASH160 + EQUALVERIFY) is omitted, a bonus selection costs only **3 opcodes** instead of 9:

```
// Bonus selection (3 ops):
{pos} OP_ROLL           // roll index from witness
{n-i} OP_MIN            // sanitize
OP_ROLL                 // roll dummy sig
```

With $t_{\text{signed}} = 8$ and $t_{\text{bonus}} = 1$, the effective subset count becomes $\binom{150}{9} \approx 2^{46.2}$, which exceeds the puzzle target. The spender now finds a valid subset in almost every pinned transaction, eliminating the grinding overhead.

The per-round opcode formula generalizes to:

$$9t_{\text{signed}} + 3t_{\text{bonus}} + 2t_{\text{total}} + 8$$

Security impact. Bonus keys reduce security because they give the attacker additional freedom:

- **Pre-image**: The attacker knows the signed indices (from revealed HORS preimages) but can freely choose bonus indices. For a round with t_{signed} signed and t_{bonus} bonus keys, the attacker gets $\binom{n-t_{\text{signed}}}{t_{\text{bonus}}}$ free bonus combinations per pinned transaction, each producing a different sighash. This reduces the pre-image cost by a factor of $\binom{n-t_{\text{signed}}}{t_{\text{bonus}}}$ per round.
- **Collision**: The attacker can assign which of the t_{total} passing indices are "signed" versus "bonus." This gives $\binom{t_{\text{total}}}{t_{\text{signed}}}$ digest choices per passing subset per round, effectively multiplying the birthday attack.

Configurations.

Config	Ops	Digest	Pre-image	Collision	Grinds
t=8, 8 (baseline)	197 (+4)	84.5b	2^{138}	2^{88}	∼180×
t=8+1b, 8	202 (−1)	84.5b	2^{131}	2^{85}	∼13×
t=8+1b, 7+2b	201 (+0)	80.4b	2^{118}	2^{78}	∼1×

Config A (t = 8+1b, 7+2b) is the recommended configuration: it fits exactly within the 201-opcode limit and eliminates grinding entirely ($\binom{150}{9} \approx 2^{46.2} \geq 2^{46}$), trading ∼20 bits of pre-image resistance (from 2^{138} to 2^{118}) for dramatically lower off-chain cost—a margin that remains ample for all foreseeable threats. Config B (t = 8+1b, 8) preserves full digest entropy but exceeds the opcode limit by 1.

4.6 Off-Chain Cost

The off-chain cost of spending a QSB transaction is dominated by the search for valid subsets across both digest rounds. We estimate costs relative to Binohash, which reports a cost of ∼$50 in GPU compute [2].

Cost structure. For each candidate subset, the spender must:

1. Apply FindAndDelete to remove the selected dummy signatures from the scriptCode.
2. Compute the sighash (SHA-256d of the ∼5 KB modified scriptCode).
3. Recover the public key via ECDSA key recovery.
4. Compute RIPEMD-160 of the recovered key and check for valid DER.

Steps 1–2 (sighash computation) dominate the per-candidate cost and are shared with Binohash. Step 3 (EC recovery, ∼10 μs on CPU) adds overhead that Binohash avoids, since its OP_SIZE puzzle requires only an ECDSA sign (∼1 μs).

Incremental sighash optimization. The sighash cost can be reduced by precomputing SHA-256 states over the fixed portions of the scriptCode. The HORS commitments (∼3,150 bytes) are identical across all subsets and can be precomputed once. Furthermore, when searching Round 2, the entire Round 1 section (∼4,850 bytes) is unaffected by FindAndDelete and forms a fixed prefix, allowing precomputation of ∼125 out of ∼152 SHA-256 blocks. For pinning, only a few bytes change per candidate (e.g., `nLocktime` or `nSequence`), allowing precomputation of all but the final SHA-256 block.

Phase	SHA blocks/candidate	EC cost	Total
Pinning	1 (of 152)	∼10 μs	∼10 μs
Round 1	∼102 (of 152)	∼10 μs	∼41 μs
Round 2	∼27 (of 152)	∼10 μs	∼18 μs

Comparison with Binohash. QSB is more expensive than BINOHASH for two reasons:

1. **Larger search space.** The RIPEMD-160 puzzle target (2^{46}) is fixed and cannot be tuned to match $\binom{n}{t}$. With Config A ($t_{\text{total}} = 9$), $\binom{150}{9} \approx 2^{46.2}$ closely matches the target, but BINOHASH can tune its OP_SIZE target to exactly match $\binom{150}{8} \approx 2^{42.3}$. This results in $\sim$16× more subsets to search per round.

2. **EC recovery overhead.** Each QSB candidate requires ECDSA key recovery ($\sim$10 μs) rather than ECDSA signing ($\sim$1 μs). However, since the sighash computation ($\sim$8–30 μs per candidate with precomputation) dominates, the per-candidate overhead is only $\sim$1.3×.

The combined overhead depends on the optimization level. At the raw work level, QSB requires $2^{47.7}$ total candidates versus BINOHASH's $2^{44.6}$—a ratio of $\sim$9×. The per-candidate cost ratio varies from 0.5× (if QSB applies incremental sighash but BINOHASH does not) to $\sim$2.5× (if both are fully optimized and EC recovery dominates), depending on implementation.

Calibrating against BINOHASH's reported $\sim$\$50 GPU cost for $2^{44.6}$ honest work [2], we estimate:

Scheme	Honest work	GPU cost
BINOHASH (t=8, 8, W_1=W_2=42)	$2^{44.6}$	$\sim$\$50
QSB Config A (t=8+1b, 7+2b)	$2^{47.7}$	$\sim$\$75–\$200

The range reflects uncertainty about the relative optimization levels. The fundamental overhead is $\sim$9× in raw work, driven primarily by the larger subset space ($\binom{150}{9}$ vs. $\binom{150}{8}$ per round). This is the cost of quantum safety under Shor's algorithm.

Measured cost breakdown. Using a GPU implementation with SHA-256 midstate precomputation and precomputed EC lookup tables, the off-chain computation consists of three phases: transaction pinning ($\sim$\$25–\$50), and two rounds of digest search ($\sim$\$25–\$50 each), for a total of $\sim$\$75–\$150 on commodity cloud GPUs at current spot pricing. The computation is embarrassingly parallel, so wall-clock time scales inversely with the number of GPUs.

SHA-256 round puzzle variant. As noted in Section 3.1.3, using `OP_SHA256` instead of `OP_RIPEMD160` for the round puzzles yields a slightly higher DER probability ($\sim 2^{-45}$ vs. $\sim 2^{-46}$). The per-candidate GPU cost is essentially identical, since the final hash is less than 2% of the total work (dominated by EC scalar multiplication and modular inversion). However, the higher DER probability means each round requires fewer candidates and the pool size n can be reduced while maintaining $P(\text{round}) \geq 1$. Table 1 compares the options.

Puzzle	n	Script	P(rnd)	Digest	Pre-image	Collision	Cost
RIPEMD, n=150	150	$\sim$9,500 B	1.18	84.1 b	2^{118}	2^{88}	\$75–\$150
SHA-256, n=150	150	$\sim$9,500 B	2.36	84.1 b	2^{115}	2^{87}	\$65–\$130
SHA-256, n=145	145	$\sim$9,200 B	1.72	83.2 b	2^{115}	2^{87}	\$50–\$100
SHA-256, n=140	140	$\sim$8,900 B	1.24	82.3 b	2^{115}	2^{86}	\$40–\$80

Table 1: Comparison of RIPEMD-160 and SHA-256 round puzzles (Config A: t_1 = 8+1b, t_2 = 7+2b). The SHA-256 variant trades $\sim$3 bits of pre-image resistance for $\sim$20–50% lower off-chain cost, and enables a smaller pool size n (saving script bytes). A hybrid configuration—RIPEMD-160 for pinning, SHA-256 for rounds—is also possible.

5 Conclusion

We have presented Quantum Safe Bitcoin (QSB), a modification of BINOHASH [2] that eliminates the scheme's reliance on Shor-breakable assumptions. The key change is replacing the signature-size puzzle—which depends on the hardness of finding small elliptic curve r-coordinates—with a hash-to-sig puzzle whose security rests entirely on the pre-image resistance of RIPEMD-160. This enables signing Bitcoin transactions in a quantum-safe manner today, without any changes to the Bitcoin protocol: QSB operates under the existing consensus rules for legacy transactions. To the best of our knowledge, this is the first scheme to achieve quantum-safe Bitcoin transactions on the live network.

The scheme offers a tunable tradeoff between security and off-chain cost. Using bonus keys, the spender can match the subset count to the fixed 2^{46} puzzle target, eliminating grinding overhead at the cost of a modest reduction in digest entropy. The resulting configuration (Config A) achieves ~118-bit pre-image resistance and ~78-bit collision resistance, with an estimated off-chain GPU cost on the order of a few hundred dollars—roughly 5–10× that of BINOHASH.

Our main contributions are:

1. **Hash-to-signature puzzle.** Replacing the OP_SIZE-based signature puzzle with a RIPEMD-160 hash-to-sig puzzle, achieving quantum safety. As a side effect, the hard-coded `SIGHASH_ALL` flag in our puzzle resolves the sighash flag uncertainty identified in BINOHASH.

2. **Bonus keys.** Introducing bonus selections that participate in FindAndDelete without HORS verification, providing a flexible tradeoff between security and off-chain cost within the tight 201-opcode budget.

5.1 Future Work

- Improved opcode utilization to increase security margins.
- Create a version that is closer to standard or perhaps even fully standard, potentially by lowering the security parameters and/or using a different method to derive the digest.[6]
- Formal security proofs of the protocol.
- Exploration of other applications of QSB beyond quantum safety—such as a more widely used tool for transaction introspection, as it resolves the sighash flag uncertainty present in BINOHASH.

Acknowledgements

We thank Robin Linus for reviewing and pointing out that the SHA-256 puzzle probability is lower than previously assumed, leading to the cost–security tradeoff presented in Table 1, as well as for useful discussions about BINOHASH. We also thank Eli Ben-Sasson and Adam Borco for comments that helped improve the presentation of this article.

References

[1] E. Heilman, "Signing a Bitcoin Transaction with Lamport Signatures (no OP_CAT)," Bitcoin Development Mailing List, 2024. https://groups.google.com/g/bitcoindev/c/mR53go5gHIk

[6] We tested using hash chains instead for the digest generation, but the opcode and script size cost exceeded the limit for the security level we were targeting. Perhaps it can be done in a better way.

[2] R. Linus, "Binohash: Transaction Introspection Without Softforks," 2026. https://robinlinus.com/binohash.pdf

[3] P. Wuille, J. Nick, and T. Ruffing, "BIP 341: Taproot: SegWit version 1 spending rules," Bitcoin Improvement Proposals, 2020. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

[4] A. Bor, "BTC PoW Locked Outputs," GitHub, 2024. https://github.com/adambor/btc-pow-locked-outputs

[5] J. Lau and E. Lombrozo, "BIP 143: Transaction Signature Verification for Version 0 Witness Program," Bitcoin Improvement Proposals, 2016. https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki

[6] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," Proceedings of the 35th Annual Symposium on Foundations of Computer Science, pp. 124–134, 1994.

[7] L. K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search," Proceedings of the 28th Annual ACM Symposium on Theory of Computing, pp. 212–219, 1996.

[8] R. Linus, "SHA-2 ECDSA," GitHub, 2024. https://github.com/RobinLinus/sha2-ecdsa

[9] L. Reyzin and N. Reyzin, "Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying," Proceedings of ACISP, pp. 144–153, 2002.

A Subset Selection Script

The following script implements a single signed selection iteration (Stage 1 of a digest round, Section 3.3.2). It is repeated t_{signed} times, with the position constants adjusted for each iteration i:

```
// Signed selection i (9 opcodes):
{2n+1-i} OP_ROLL            // roll index from witness
{n-i} OP_MIN                 // sanitize (prevent out-of-bounds)
OP_DUP                       // copy index (needed for sig lookup)
{n+1} OP_ADD                 // compute commitment position
OP_ROLL                      // roll HORS commitment to top
{2n+1+t-2i} OP_ROLL          // roll HORS preimage from witness
OP_HASH160                   // hash preimage
OP_EQUALVERIFY               // verify preimage matches commitment
OP_ROLL                      // roll dummy sig (using index on stack)
```

This is identical to the selection loop in Binohash [2, Appendix B]. The position constants account for sig_{nonce} being on the stack above the dummy signatures (hence $2n+1$ rather than $2n$).

B Full QSB Script ($t = 8$, $t = 8$, $n = 150$)

The complete locking script for the baseline configuration (197 opcodes, ∼9,650 bytes). HORS commitments and dummy signatures are abbreviated.

```
// ============================================================
// PINNING (5 opcodes)
// ============================================================
// Witness: <key_puzzle> <key_nonce>
```

```
<sig_nonce>                     // 9 bytes, hardcoded, SIGHASH_ALL
OP_OVER                         // [1] copy key_nonce
OP_CHECKSIGVERIFY               // [2] verify (sig_nonce, key_nonce)
OP_RIPEMD160                    // [3] key_nonce -> sig_puzzle
OP_SWAP                         // [4] get key_puzzle on top
OP_CHECKSIGVERIFY               // [5] verify (sig_puzzle, key_puzzle)


// ============================================================
// ROUND 1 (96 opcodes)
// ============================================================

// --- 150 HORS commitments (20 bytes each) ---
<H(pre_149)_r1>
<H(pre_148)_r1>
...
<H(pre_000)_r1>

// --- 150 dummy sigs (9 bytes each, SIGHASH_SINGLE) ---
<sig_149_r1>
<sig_148_r1>
...
<sig_000_r1>

OP_0                            // CHECKMULTISIG dummy
<sig_nonce_r1>                  // 9 bytes, hardcoded, SIGHASH_ALL

// --- 8 signed selections (9 ops each = 72 ops) ---
// Selection 0:
301 OP_ROLL                     // roll index_0 from witness
150 OP_MIN                      // sanitize
OP_DUP                          // copy index
151 OP_ADD                      // commitment position
OP_ROLL                         // roll commitment
309 OP_ROLL                     // roll preimage_0 from witness
OP_HASH160                      // hash preimage
OP_EQUALVERIFY                  // verify HORS
OP_ROLL                         // roll dummy sig
// Selections 1--7: (same structure, adjusted positions)
...

// --- Puzzle + CHECKMULTISIG (24 ops) ---
302 OP_ROLL                     // roll key_nonce from witness
OP_DUP                          // copy for RIPEMD + pubkey
OP_RIPEMD160                    // key_nonce -> sig_puzzle

302 OP_ROLL                     // roll key_puzzle from witness
OP_CHECKSIGVERIFY               // verify (sig_puzzle, key_puzzle)

OP_9                            // M = 9 (8 dummies + sig_nonce)
OP_2 OP_ROLL                    // move key_nonce into pubkey zone
302 OP_ROLL                     // roll pub_0
...                             // (8 rolls total)
302 OP_ROLL                     // roll pub_7
OP_9                            // N = 9
OP_CHECKMULTISIG                // verify all 9 sig/key pairs


// ============================================================
// ROUND 2 (96 opcodes) -- same structure as Round 1
// ============================================================
// (identical structure, independent data)
...
```

Full annotated scripts with exact position constants for multiple configurations are available at: https://github.com/avihu28/quantum-safe-bitcoin-tx