Antichain Winternitz: Guaranteed Garbled-Circuit Label Revelation on Bitcoin with Permissionless Recovery

Mukesh Tiwari* and Aaron Feickert†

Alpen Labs

Abstract

Trust-minimized bridges on Bitcoin move SNARK verification off chain by evaluating the verifier as a garbled circuit. The bridge's on-chain spending condition obliges the Garbler to reveal the labels for one input without enabling the Evaluator to derive labels for any other input. Existing designs commit to each input bit with a Lamport signature which is costlier on chain, or with adaptors where there is no guarantee that the spend actually reveals the labels. We present *Antichain Winternitz*, a parametrized hash-chain construction whose admissible codewords form a constant-sum antichain. The on-chain locking script accepts an opening witness only if it encodes a valid codeword consistent with the committed chain terminals. Every accepted opening witness yields a valid codeword while the public off-chain table, verified during setup, maps every admissible codeword to its garbled-circuit labels, so any observer can recover them. Depending on the parameter set, we can obtain up to a 52.9% saving over Lamport signatures with only added off-chain storage of 43.8 kB per message bit.

1 Introduction

Recent trust-minimized Bitcoin bridge designs, including Glock [3], Mosaic [5], BitVM3 [7], and Delbrag [10], use garbled circuits to move the verification of SNARK proofs off chain. During setup, the parties agree on the SNARK-verifier circuit and ensure that the corresponding garbled circuit is correct. Privacy-free garbling [4] ensures authenticated evaluation of that circuit on an input selected by the Garbler. The relevant Bitcoin spending condition is intended to ensure that every successful spend publishes enough information for any observer to recover one complete input label bundle. The reveal mechanism must provide one usable label per wire without enabling the Evaluator to derive an additional label for any input wire. The circuit's output can then serve as evidence in the on-chain protocol.

Once SNARK verification moves off chain, label revelation accounts for a significant portion of the remaining on-chain cost. Both the verification script and the witness data needed to reveal the input labels must be included in the spending transaction. BitVM3 identifies per-bit Lamport as an immediate instantiation [6, 7], but uses adaptor signatures in its concrete construction. Mosaic also uses adaptor signatures for label revelation [5]. In §2.7, we explain why even public, verified adaptors do not provide guaranteed permissionless extraction, since they admit accepted spends that reveal no labels. We present *Antichain Winternitz* (ACW), a parametrized hash-chain constructions that provide guaranteed permissionless extraction and label unforgeability while lowering the on-chain cost per input bit relative to per-bit Lamport. We also explore why naive Winternitz does not fulfill the requirements in §2.6.

Contributions. Constant-sum Winternitz encodings and maximum antichains in products of chains are known [2, 13]. Our contribution is the per-digit interpretation of these results for garbled-circuit label revelation, a compact Bitcoin Script verifier that uses BIP 342's consensus MINIMALIF rule as a range check [12], and a measurement of the trade-off between on-chain cost and off-chain table size.

*Corresponding author: mukesh@alpenlabs.io

†aaron@alpenlabs.io

2 Preliminaries

2.1 Privacy-Free Garbled Circuits

A Boolean circuit computes a function on bits by composing simple operations, called *gates*, such as AND, XOR, and NOT. Wires carry input bits into gates and carry each gate's output either to another gate or to the circuit output.

A privacy-free garbled circuit allows one party, called the *Garbler*, to encode a Boolean circuit so that the other party, called the *Evaluator*, can evaluate it only on an input chosen by the Garbler.

The Garbler privately assigns random bit strings, called *labels*, to each possible value of the input, intermediate, and output wires. The Boolean representation of the function is then garbled gate by gate to obtain garbled tables. Each garbled table allows the labels on a gate's input wires to be used to obtain the output label corresponding to the bit value of the output wire. The actual garbling scheme used is not relevant for the label reveal mechanism as it only cares about each labels corresponding to the values of a wire which holds generally.

For input wires $1, \ldots, \ell$, let W_i^0 and W_i^1 denote the two labels for wire i. For an input $\mathbf{v} = (v_1, \ldots, v_\ell)$, we call $\Lambda_{\mathbf{v}} := W_1^{v_1} \parallel \cdots \parallel W_\ell^{v_\ell}$ its *input label bundle*.

The output label can have some semantic value imposed by an overarching protocol. By sharing an input label bundle $\Lambda_{\mathbf{v}}$, the Garbler gives the Evaluator exactly one label per input wire, encoding the input $\mathbf{v}$. The Garbler can then be sure that the Evaluator can obtain only the output labels corresponding to the value of the function evaluated at $\mathbf{v}$. This is the *authenticity* property.

The term *privacy-free* reflects that the scheme provides this authenticity without requiring the input to remain private [4].

2.2 Bitcoin UTXOs and Tapscript

Bitcoin transactions consume existing unspent transaction outputs (UTXOs) and create new ones. Each UTXO records an amount and specifies how that amount may be spent. A transaction spending the UTXO must supply the data required by those spending rules.

Taproot allows a Bitcoin UTXO to be spent either by providing a signature, called the *key path*, or by revealing and satisfying a script committed by the UTXO, called the *script path* [11]. For our use case, the key path must not be usable. This can be achieved by choosing a well-known and standard Nothing Up My Sleeve (NUMS) point as the Taproot internal public key, meaning a curve point for which no corresponding secret key is efficiently computable.

These locking scripts are written in Bitcoin Script, a stack-based language. A script revealed in a Taproot script-path spend is called a *tapscript*. We call the tapscript the *locking script* and the stack inputs supplied to it the *witness*. Tapscript uses Bitcoin Script with some changes [12]. One of the changes relevant to us is the MINIMALIF rule, which requires a value consumed by `OP_IF` or `OP_NOTIF` to be either the empty vector or the single byte `0x01`. Tapscript execution succeeds if it terminates without error and leaves exactly one stack element that evaluates to true.

Bitcoin script evaluates opcodes using a main stack: each opcode consumes its operands from the top of the stack and pushes any result back onto it, while the witness elements form the initial stack. We use the notation $\langle x_1 \rangle \cdots \langle x_t \rangle$ for stack elements ordered from bottom to top, so $\langle x_t \rangle$ is the top element and is consumed first. Bitcoin Script also provides an *alternate stack* for temporary storage; `OP_TOALTSTACK` moves the top element of the main stack to the alternate stack, while `OP_FROMALTSTACK` moves the top element back to the main stack.

Bitcoin transaction costs are measured by weight. Non-witness bytes contribute four weight units each, while witness bytes contribute one. The corresponding virtual size is measured in *virtual bytes* (vBytes) and is the transaction weight divided by four, rounded up [8]. In a Taproot script-path spend, the locking script, and stack inputs are all serialized as witness data. Therefore, both the locking script and its witness contribute to the spending cost reported in vBytes.

2.3 Garbled Locks

Glock [3] formalized the idea of *garbled locks* as a way to make the result of an off-chain garbled-circuit evaluation enforceable on Bitcoin. This was expanded upon by subsequent works like Mosaic [5] and BitVM3 [7]. The circuit encodes a SNARK verifier whose direct execution on Bitcoin would be challenging (due to script limitations) and expensive (due to script size). During setup, the parties agree on a SNARK verifier circuit and ensure that the corresponding garbling tables are correctly generated.

The Garbler chooses an input $\mathbf{v}$ and authenticates it by revealing the input label bundle $\Lambda_{\mathbf{v}}$. The Evaluator uses this bundle and the garbled tables to obtain the output label corresponding to the result of the verification.

Bitcoin spending conditions connect these labels to funds. One spending condition requires publication of the input label bundle, while another can use the output label as a witness to authorize a reward or enforce a penalty.

2.4 Label Reveal on Bitcoin

For the label-revelation mechanism described in Section 2.3, we impose the following two requirements:

1. *Guaranteed permissionless extraction.* Any successful spend of the UTXO must allow any observer to recover exactly one complete input-label bundle using only transaction data and any public setup data. Setup data may be generated by the garbler in advance and made publicly available; this ensures that extraction is not limited to permissioned entities.

2. *Label unforgeability.* Given an accepted spending witness revealing the input label bundle ($\Lambda_{\mathbf{v}}$), together with any public setup information, no efficient observer should be able to recover any input-wire label outside the revealed bundle (i.e, $W_i^{1-v_i}$ for any input wire i). This implies that no efficient observer can recover another input label bundle $\Lambda_{\mathbf{v}'}$ for any $\mathbf{v}' \neq \mathbf{v}$.

Validity of revealed labels with respect to a particular garbled circuit depends on the setup process and is therefore out of scope for this work. However, we require that once any setup process is completed, extraction must succeed according to our definition.

2.5 Lamport Signatures

The Lamport signature scheme [6] is a hash-based one-time signature scheme. We briefly explain how the scheme works for signing a single message bit.

Let n be the security parameter and $H : \{0,1\}^n \to \{0,1\}^n$ be a preimage- and collision-resistant hash function. Let $m \in \{0,1\}$ be the message bit.

KeyGen(1^n): Sample $x_0, x_1 \xleftarrow{\$} \{0,1\}^n$ independently and uniformly, and set $y_b := H(x_b)$ for $b \in \{0,1\}$. The secret and public keys are

$$sk = (x_0, x_1), \qquad pk = (y_0, y_1).$$

Sign(sk, m): output $\sigma := x_m$.

Verify(pk, m, σ): output 1 iff $H(\sigma) = y_m$.

Figure 1. The Lamport one-time signature scheme for a single bit.

To generalize the scheme over ℓ-bit messages, use ℓ independent instances of the scheme, one for each bit.

For input wire i, the first 128 bits of each Lamport secret value x_b are used as the corresponding input label W_i^b [1]. A Lamport signature on $\mathbf{v}$ therefore reveals the complete input label bundle $\Lambda_{\mathbf{v}}$. Thus, every

[1] Bitcoin script supports specific hash functions of digest length 20 or 32 bytes; for security targets below this level, it suffices to truncate the hashes to get desired length

successful UTXO spend gives guaranteed permissionless extraction. Unforgeability of labels follows from the pre-image resistance of the the hash function and the independent sampling of the secret keys.

The corresponding Bitcoin script verification logic and detailed cost analysis are given in Appendix A.

2.6 Winternitz Signatures

The Winternitz signature scheme [9] provides an optimization of the Lamport scheme in terms of both public key and signature size. The tradeoff is in more computation needed to generate and verify signatures. Although several variants of Winternitz exist in the literature, we describe one that is well suited for use with Bitcoin Script.

Definition 1 (Hash chain and coordinates)**.** Let $n \geq 1$ and $L \geq 1$ be integers, and let $f : \{0,1\}^n \to \{0,1\}^n$ be a function. For an integer $j \geq 0$, the *j-th iterate* of f is defined by $f^{(0)}(x) := x$ and $f^{(j)}(x) := f\big(f^{(j-1)}(x)\big)$ for $j \geq 1$. For $x \in \{0,1\}^n$, the sequence

$$x,\ f^{(1)}(x),\ \ldots,\ f^{(L-1)}(x)$$

is the *hash chain* of length L with *base* x and *terminal* $f^{(L-1)}(x)$. Write $[L] := \{0, \ldots, L-1\}$. For $a \in [L]$, the value at *coordinate* a of this chain is

$$\sigma_a := f^{(L-1-a)}(x).$$

If $y := f^{(L-1)}(x)$ is the terminal, then $f^{(a)}(\sigma_a) = y$. Thus the terminal value lies at coordinate 0, while the base value lies at coordinate $L-1$. For $a > 0$, applying f to σ_a yields σ_{a-1}.

Fix a Winternitz digit width $w \geq 1$ and set the chain length to $L := 2^w$. To encode an ℓ-bit message, set

$$k_1 := \left\lceil \frac{\ell}{w} \right\rceil, \qquad k_2 := \left\lfloor \log_L\big(k_1(L-1)\big) \right\rfloor + 1, \qquad k := k_1 + k_2.$$

Pad the message to a multiple of w bits and parse it as $\mathbf{m} = (m_1, \ldots, m_{k_1}) \in [L]^{k_1}$. Define its Winternitz checksum by

$$c(\mathbf{m}) := \sum_{i=1}^{k_1} (L - 1 - m_i),$$

and let $(c_1, \ldots, c_{k_2}) \in [L]^{k_2}$ be the length-k_2 base-L representation of $c(\mathbf{m})$, padded with leading zeros when necessary. The standard Winternitz encoding is

$$P_{\mathrm{WOTS}}(m) := (m_1, \ldots, m_{k_1}, c_1, \ldots, c_{k_2}) \in [L]^k.$$

KeyGen(1^n): Sample $x_1, \ldots, x_k \xleftarrow{\$} \{0,1\}^n$ independently and uniformly, and set $y_i := f^{(L-1)}(x_i)$ — the terminal of the chain with base x_i — for $i \in \{1, \ldots, k\}$. The secret and public keys are

$$sk = (x_1, \ldots, x_k), \qquad pk = (y_1, \ldots, y_k).$$

Sign(sk, m): compute $(s_1, \ldots, s_k) := P_{\mathrm{WOTS}}(m)$ and output

$$\sigma := \big(f^{(L-1-s_1)}(x_1), \ldots, f^{(L-1-s_k)}(x_k)\big).$$

Verify(pk, m, σ): compute $(s_1, \ldots, s_k) := P_{\mathrm{WOTS}}(m)$ and output 1 iff $f^{(s_i)}(\sigma_i) = y_i$ for all $i \in \{1, \ldots, k\}$.

Figure 2. The Winternitz one-time signature scheme with k chains, L coordinates per chain, and encoding P_{WOTS}.

We require that f be preimage- and collision-resistant so that nonterminal chain values cannot be derived from the public terminals and accepted values remain bound to the chains fixed during setup.

For label revelation, one option is to publish a table mapping every valid Winternitz signature to its corresponding input label bundle. This satisfies both requirements, but the table contains one entry per message and therefore grows exponentially in ℓ. Such a table is impractical for any realistic message length, particularly when the message is a SNARK proof.

A more size-efficient alternative is to map each digit's chain values independently to the labels of the bits encoded by that digit. If the digit width is w, the public translation table has 2^w entries. The number of tables then grows linearly with number of digits needed to encode the message. Although this scheme gives guaranteed permissionless extraction, it violates label unforgeability: from a revealed coordinate, the observer can hash forward to a smaller coordinate and recover the labels associated with another digit value, even though the complete Winternitz signature remains unforgeable.

Adding a separate checksum to each digit prevents this forward derivation without requiring a table over complete messages. This is the $k = 2$ variant of the ACW construction presented in Section 3; larger values of k give more efficient instantiations, but with other tradeoffs.

2.7 Adaptor Signatures

Adaptor signatures are an elegant and economical way to tie a signature to the revelation of committed data [1]. The signing-key holder produces a pre-signature that another party can complete on a specific message using a previously committed secret. Once it is completed, the signing-key holder can use the pre-signature and the completed signature to extract that secret.

Let $\mathbb{G}$ be a group of prime order p with generator G, write $[x] = xG$, and let $H : \mathbb{G} \times \mathbb{G} \times \{0,1\}^* \to \mathbb{F}_p$. The Evaluator holds a signing key $sk \in \mathbb{F}_p$ with public key $pk = [sk]$.

Definition 2 (Schnorr adaptor signature)**.** For a message m, a secret value y, and the corresponding adaptor point $S = [y]$, the scheme consists of the following algorithms.

GenAdaptor	Sample $r' \xleftarrow{\$} \mathbb{F}_p^*$, set $R' = [r']$, $R = R' + S$, $e = H(R, pk, m)$, and output the adaptor pre-signature $\widetilde{\sigma} = (s', R', S)$, where $s' = r' + e\,sk$.
VerAdaptor	Accept $\widetilde{\sigma}$ if $[s'] = R' + e\,pk$, where $e = H(R' + S, pk, m)$.
Complete	Given y such that $S = [y]$, output the ordinary Schnorr signature $\sigma = (s' + y, R' + S)$.
Extract	Given $\widetilde{\sigma}$ and its completed signature $\sigma = (s, R)$, output $y = s - s'$.

Mosaic [5] and BitVM3 [7] use adaptor signatures for label revelation. Although adaptor signatures are attractive from a cost perspective, they do not satisfy guaranteed permissionless extraction. Because the Evaluator holds the signing key, it can spend the UTXO using an ordinary signature, bypassing adaptor completion and revealing no label. This remains possible even if the adaptor pre-signatures are public and verified.

Thus, only the Evaluator can ensure that extraction occurs; every other observer must trust the Evaluator not to bypass adaptor completion.

2.8 Order Theory

The ACW construction uses partial orders to describe which combinations of hash-chain coordinates can be obtained from one another by forward hashing.

Definition 3 (Partial order and comparability)**.** A binary relation $\preceq$ on a set P is a *partial order* if, for all $x, y, z \in P$,

$$x \preceq x, \qquad (x \preceq y \wedge y \preceq x) \Longrightarrow x = y, \qquad (x \preceq y \wedge y \preceq z) \Longrightarrow x \preceq z.$$

These properties are called reflexivity, antisymmetry, and transitivity, respectively. The pair $(P, \preceq)$ is a *partially ordered set*, or *poset*. Elements $x, y \in P$ are *comparable* if $x \preceq y$ or $y \preceq x$; otherwise they are *incomparable*.

Definition 4 (Antichain)**.** A subset of a poset is an *antichain* if no two distinct elements are comparable.

3 Construction

In this section, we construct ACW, its on-chain spending condition, and the public off-chain translation table used to recover an input label bundle from a spending witness.

Definition 5 (Coordinate poset)**.** For $k \geq 2$ hash chains, define the *coordinate space*

$$\mathcal{P}_{k,L} := [L]^k \quad \text{where} [L] = \{0, \ldots, L-1\}.$$

For $\mathbf{a} = (a_1, \ldots, a_k)$ and $\mathbf{b} = (b_1, \ldots, b_k)$ in $\mathcal{P}_{k,L}$, define

$$\mathbf{a} \preceq \mathbf{b} \quad \Longleftrightarrow \quad a_i \leq b_i \text{ for every } i \in \{1, \ldots, k\}.$$

The relation $\preceq$ is the *coordinatewise order* on $\mathcal{P}_{k,L}$.

The following result is Theorem 4 of [13], stated in our notation without proof.

Theorem 6 (Maximum constant-sum antichain)**.** *The set*

$$\mathcal{A}_{k,L} := \left\{ \mathbf{a} \in \mathcal{P}_{k,L} : \sum_{i=1}^{k} a_i = \left\lfloor \frac{k(L-1)}{2} \right\rfloor \right\}$$

is a maximum-cardinality antichain in $\mathcal{P}_{k,L}$.

We call each $\mathbf{a} \in \mathcal{A}_{k,L}$ an *admissible codeword*. For fixed k and L, using a maximum antichain maximizes the number of digit values that the construction can represent.

To describe the ACW construction, we use Setup, Reveal, and Verify rather than the signature-scheme notation KeyGen, Sign, and Verify for ACW as it is not developed as a complete signature scheme: it operates directly on admissible codewords, and the corresponding hash chain values are used by the translation table in Section 3.2 to recover input- label bundles.

Setup(1^n): Sample $x_1, \ldots, x_k \xleftarrow{\$} \{0,1\}^n$ independently and uniformly, and set $y_i := f^{(L-1)}(x_i)$ for every $i \in \{1, \ldots, k\}$. The secret and public values are

$$\mathbf{x} := (x_1, \ldots, x_k), \qquad \mathbf{y} := (y_1, \ldots, y_k).$$

Reveal$(\mathbf{x}, \mathbf{a})$: For $\mathbf{a} = (a_1, \ldots, a_k) \in \mathcal{A}_{k,L}$, output

$$\boldsymbol{\sigma}_{\mathbf{a}} := \left(f^{(L-1-a_1)}(x_1), \ldots, f^{(L-1-a_k)}(x_k) \right).$$

Verify$(\mathbf{y}, \mathbf{a}, \boldsymbol{\sigma})$: Output 1 iff $\mathbf{a} \in \mathcal{A}_{k,L}$ and

$$f^{(a_i)}(\sigma_i) = y_i \qquad \text{for every } i \in \{1, \ldots, k\}.$$

Figure 3. Revealing and verifying the hash-chain values associated with an admissible codeword $\mathbf{a} \in \mathcal{A}_{k,L}$.

For every $\mathbf{a} \in \mathcal{A}_{k,L}$, an honestly generated reveal is accepted, since applying f exactly a_i times to each revealed chain value reaches the corresponding terminal y_i.

Since $\mathcal{A}_{k,L}$ is an antichain, for any distinct $\mathbf{a}, \mathbf{b} \in \mathcal{A}_{k,L}$, there is a coordinate i such that $b_i > a_i$. Deriving the chain value at coordinate b_i from the value revealed at coordinate a_i would require moving "backward" along that chain which amounts to finding a preimage under f. Thus, a revealed ACW witness cannot be transformed into a witness for another admissible codeword.

3.1 On-chain Verification Script

Let

$$R := \left\lfloor \frac{k(L-1)}{2} \right\rfloor .$$

A spending witness provides $\sigma_1, \ldots, \sigma_k$ and the first $k-1$ coordinates $a_1, \ldots, a_{k-1}$. The verifier computes $\Sigma := \sum_{i=1}^{k-1} a_i$ and derives

$$a_k := R - \Sigma.$$

Thus the coordinates sum to R without a separate equality check.

We order the witness elements such that the script shown in Listing 1 processes $\langle \sigma_i \rangle \langle a_i \rangle$ for $i = 1, \ldots, k-1$ in sequence. After these pairs are consumed, $\langle \sigma_k \rangle$ remains. The following script fragment stores the running sum of the posted coordinates temporarily on the alternate stack:

Listing 1. Accumulating the posted coordinates and deriving a_k. Each per-chain placeholder is instantiated by Listing 2.

```
# first posted coordinate:
OP_DUP OP_TOALTSTACK                          # Sigma := a_1
<per-chain check for sigma_1, a_1>
# repeat for i = 2,...,k-1:
OP_DUP OP_FROMALTSTACK OP_ADD OP_TOALTSTACK  # Sigma := Sigma + a_i
<per-chain check for sigma_i, a_i>
# after the first k-1 checks, sigma_k is on top:
<R> OP_FROMALTSTACK OP_SUB                    # a_k := R - Sigma
<per-chain check for sigma_k, a_k>
```

For each chain, the verifier supplies a_i directly to a per-chain check. The check advances σ_i by exactly a_i hash steps and checks

$$f^{(a_i)}(\sigma_i) = y_i$$

as required by Figure 3. For example, the complete per-chain check for $L = 4$ is is shown in Listing 2.

Listing 2. Per-chain verification for $L = 4$. The input is $\langle \sigma_i \rangle \langle a_i \rangle$ and `<f>` denotes the Bitcoin Script hash opcode implementing f.

```
OP_DUP OP_2 OP_GREATERTHANOREQUAL
OP_IF OP_2 OP_SUB OP_SWAP <f> <f> OP_SWAP OP_ENDIF
OP_IF <f> OP_ENDIF                            # final count bit and range check
<y_i> OP_EQUALVERIFY
```

For general $L = 2^w$, the per-chain check treats the input coordinate a_i as the number of hash steps to apply. It maintains a remaining hash-step count, initially equal to a_i. For each $q \in \{2^{w-1}, \ldots, 2\}$, it tests whether this remaining count is at least q. If so, it subtracts q and applies f exactly q times; otherwise it does neither. The remaining count is then passed to the bare `OP_IF`, which conditionally applies f once for the final bit. If $a_i \in [L]$, the check applies exactly a_i hashes and its final `OP_IF` receives either 0 or 1. If $a_i \geq L$, the preceding steps leave a remaining count of at least 2; if $a_i < 0$, the count remains negative. In both cases, the consensus MINIMALIF rule for tapscript rejects because the argument to `OP_IF` is neither the empty vector nor the single byte `0x01` [12]. The check therefore accepts exactly when $a_i \in [L]$.

Accepted witnesses. The last coordinate derived in the script itself enforces the constant sum. The per-chain checks enforce the coordinate ranges, and the terminal comparisons enforce the verification equations. The complete locking script concatenates these checks for all k chains. Under collision resistance of f, the terminal comparisons also bind the accepted chain values to those fixed during setup. This establishes the on-chain part of guaranteed permissionless extraction: every successful spend reveals an admissible codeword and the chain values needed to recover its translation-table entry. Embedding the script and the public terminals $\mathbf{y}$ in a Taproot tapleaf commits the UTXO to this spending condition [11].

3.2 Off-chain Translation Table

Let

$$N := |\mathcal{A}_{k,L}|, \qquad b := \lfloor \log_2 N \rfloor .$$

Order the codewords lexicographically, with the first coordinate most significant,

$$\mathbf{a}^{(0)} <_{\text{lex}} \mathbf{a}^{(1)} <_{\text{lex}} \cdots <_{\text{lex}} \mathbf{a}^{(N-1)}.$$

The translation table has one entry for every codeword in $\mathcal{A}_{k,L}$. Entry $\mathbf{a}^{(j)}$ represents the b-bit encoding of $j \bmod 2^b$. If this encoding is $(v_1, \ldots, v_b) \in \{0,1\}^b$, with v_1 the most significant bit, its label bundle is

$$\Lambda_{\mathbf{a}^{(j)}} := W_1^{v_1} \parallel \cdots \parallel W_b^{v_b},$$

where W_i^0 and W_i^1 are the two garbled-circuit labels for input wire i.

The party constructing the table knows $\mathbf{x}$ and the input labels and can therefore compute both $\boldsymbol{\sigma}_{\mathbf{a}}$ and $\Lambda_{\mathbf{a}}$ for every entry. Let KDF be a hash-based key-derivation function whose output is as long as a label bundle, and define

$$K_{\mathbf{a}} := \mathsf{KDF}(\mathtt{table} \parallel \mathbf{y} \parallel \mathbf{a} \parallel \sigma_{\mathbf{a},1} \parallel \cdots \parallel \sigma_{\mathbf{a},k}), \qquad \mathsf{T}[\mathbf{a}] := \Lambda_{\mathbf{a}} \oplus K_{\mathbf{a}}.$$

The tuples are serialized canonically, with every coordinate encoded using exactly w bits. The completed table T is published before any codeword is revealed to ensure anyone can utilize this table to extract labels from the witness.

To reveal an intended b-bit value $v \in \{0, \ldots, 2^b - 1\}$, the honest party uses the canonical codeword $\mathbf{a}^{(v)}$. When N is not a power of two, the remaining codewords wrap around and provide additional encodings of existing values. Consequently, every codeword accepted by the on-chain verifier has a corresponding entry and a defined label bundle.

From an accepted spending witness, an observer derives a_k as above and obtains the complete tuple $(\mathbf{a}, \boldsymbol{\sigma}_{\mathbf{a}})$. The observer then recomputes $K_{\mathbf{a}}$ and recovers

$$\Lambda_{\mathbf{a}} = \mathsf{T}[\mathbf{a}] \oplus K_{\mathbf{a}}.$$

Thus, no additional secret or interaction is required, establishing guaranteed permissionless extraction.

The antichain property prevents a witness for one codeword from revealing any entry corresponding to a different codeword that reveals any additional input label. For distinct $\mathbf{a}, \mathbf{b} \in \mathcal{A}_{k,L}$, there exists an index i such that $b_i > a_i$, so computing the complete KDF input for entry $\mathbf{b}$ from the witness for $\mathbf{a}$ requires moving backward along that chain. Under preimage resistance of f and the pseudorandomness of KDF, the entry $\mathsf{T}[\mathbf{b}]$ therefore reveals no label not already contained in $\Lambda_{\mathbf{a}}$. This establishes label unforgeability.

4 Implementation and Benchmarks

We now provide benchmarks comparing the on-chain efficiency of ACW and Lamport signatures, using a research implementation in Rust[2]. We also provide the size of the lookup table required for ACW.

4.1 On-chain Cost and Off-chain Storage

The 20-byte Lamport baseline in Appendix A implies a cost of 17.75 vBytes/bit. ACW with parameters $(k, L) = (4, 16)$ yields 11 message bits per digit at a cost of 8.36 vBytes/bit, a 52.9% reduction relative to Lamport.

There is an inherent tradeoff between the size of on-chain and off-chain data. We have assumed labels to be 128 bits long. For $(k, L) = (4, 16)$, the complete table occupies 43.776 kB per message bit, or 44.83 MB for a 1024-bit input. Table 1 enumerates the marginal on-chain as well as off-chain cost per bit for selected ACW parameters.

[2] https://github.com/alpenlabs/acw

k	L	$\lvert\mathcal{A}_{k,L}\rvert$	bits/codeword	vBytes/bit	vs. Lamport	table size/bit
2	16	16	4	11.31	−36.3%	0.256 kB
3	16	192	7	9.82	−44.7%	3.072 kB
4	8	344	8	9.34	−47.4%	5.504 kB
4	**16**	**2736**	**11**	**8.36**	**−52.9%**	**43.776 kB**
5	16	39280	15	7.68	−56.7%	0.628 MB

Table 1. ACW's on-chain and off-chain marginal cost at a 20-byte digest for various parameter choices. The comparison uses the 17.75 vBytes/bit Lamport baseline and assumes 128 bit labels. The bits/codeword column shows the number of bits that fit in one codeword.

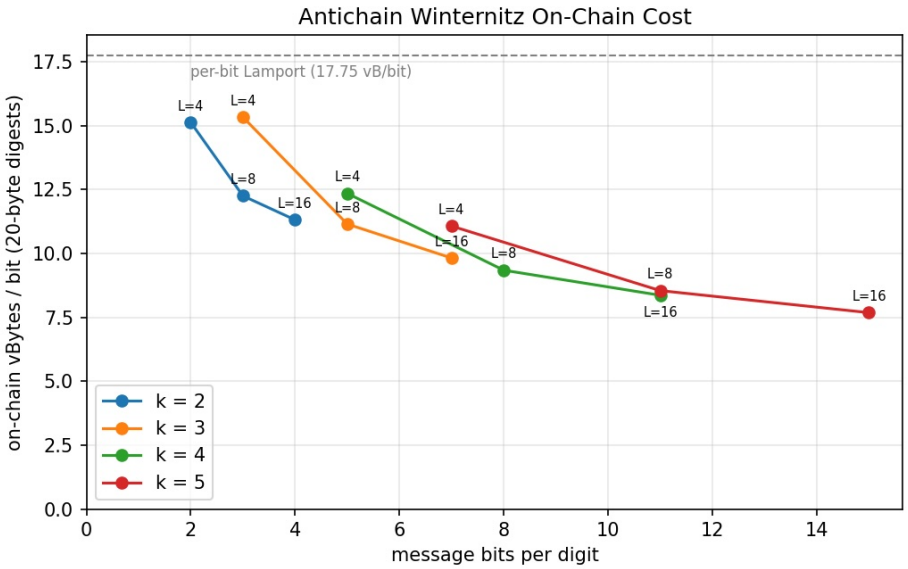

Figure 4. On-chain cost against message bits per digit for ACW instantiated using various parameter sizes while using 20-byte digest. The dashed line is the 17.75 vBytes/bit Lamport baseline. Table 1 gives the corresponding off-chain table sizes for selected parameter sets.

References

[1] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized channels from limited blockchain scripts and adaptor signatures. Cryptology ePrint Archive, Paper 2020/476, 2020.

[2] N. G. de Bruijn, C. A. van Ebbenhorst Tengbergen, and D. Kruyswijk. On the set of divisors of a number. *Nieuw Archief voor Wiskunde (2)*, 23:191–193, 1951.

[3] Liam Eagen. Glock: Garbled locks for bitcoin. Cryptology ePrint Archive, Paper 2025/1485, 2025.

[4] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *EUROCRYPT*, 2015.

[5] Nakul Khambhati, Mukesh Tiwari, Sapin Bajracharya, Manish Bista, Liam Eagen, Christian Lewe, and Aaron Feickert. Mosaic: Practical malicious security for garbled circuits on bitcoin. Cryptology ePrint Archive, Paper 2026/812, 2026.

[6] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International, 1979.

[7] Robin Linus, Lukas Aumayr, Matteo Maffei, et al. BitVM3: Efficient bitcoin bridges via garbled circuits. Cryptology ePrint Archive, Paper 2026/933, 2026.

[8] Eric Lombrozo, Johnson Lau, and Pieter Wuille. BIP-141: Segregated witness (consensus layer). Bitcoin Improvement Proposal, 2015.

[9] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology — CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1989.

[10] Jeremy Rubin. Delbrag. Draft manuscript, 2025.

[11] Pieter Wuille, Jonas Nick, and Anthony Towns. BIP-341: Taproot: Segwit version 1 spending rules. Bitcoin Improvement Proposal, 2020.

[12] Pieter Wuille, Jonas Nick, and Anthony Towns. BIP-342: Validation of taproot scripts. Bitcoin Improvement Proposal, 2020.

[13] Kaiyi Zhang, Hongrui Cui, and Yu Yu. Revisiting the constant-sum Winternitz one-time signature with applications to SPHINCS$^+$ and XMSS. Cryptology ePrint Archive, Paper 2023/850, 2023.

Appendix A Lamport Signature Locking Script

Listing 3. Single-bit Lamport: Locking Script and Witness

```
# --- Witness : pushed by the spender ---
<preimage>       # 20-byte preimage  x_b
<bit>            # 1-byte revealed bit  b_i
# --- Locking Script ---
OP_IF
  OP_RIPEMD160
  <terminal1>     # 20-byte terminal  H(x_1)
  OP_EQUALVERIFY
OP_ELSE
  OP_RIPEMD160
  <terminal0>     # 20-byte terminal  H(x_0)
  OP_EQUALVERIFY
OP_ENDIF
```

Table 2 below computes the size of the witness as well as the locking script. We analyze it for both 20-byte digest as well as 32-byte digest available in Bitcoin. This provides us with a baseline to compare our ACW scheme against.

		size of element (bytes)	
	element	20-byte digest	32-byte digest
witness	preimage push (1+ℓ)	21	33
	bit push	1	1
	witness subtotal	22	34
locking script	`OP_IF/OP_ELSE/OP_ENDIF`	3	3
	2× hash opcode	2	2
	2× terminal push (1+ℓ)	42	66
	2× `OP_EQUALVERIFY`	2	2
	locking script subtotal	49	73
total raw bytes / bit		**71**	**107**
measured vBytes / bit		17.75	26.75

Table 2. Per-bit Lamport signature script and witness size