

Violet’s Constrained Locks, or: How I Learned to Stop Worrying and Love Scriptless Chains

Strahn Cryptography
avi@strahn.xyz

Confidential draft, October 21, 2026

Abstract

Zcash’s shielded pools have no scripts, so the multisignatures, timelocks and hash locks behind payment channels and atomic swaps on Bitcoin cannot be written there. We introduce Violet’s Constrained Locks (VCLs). Two parties jointly control a shielded address created by a FROST key generation, and agree offline on the transactions that may spend from it. A lock’s conditions come from composition rather than on-chain code. Following Skyhook, transaction identifiers can be predicted before signing and checked from a few revealed fields. Transaction format v6, introduced in NU6.3 along with the Ironwood pool, moves the anchor out of the identifier, which extends prediction to spends of notes that are not yet on chain. Zero-knowledge proofs constrain the outputs of the predicted transaction. The choice of threshold signing scheme sets what releases the lock: plain FROST; preFROST, a threshold adaptor signature; or hashFROST, which ties the adaptor to a hash preimage. We define VCLs and their security and prove the algebraic core in Lean 4 with Mathlib. We give reductions for most of the rest; for the unforgeability of preFROST and hashFROST we give only a proof plan. An appendix lists the status of every claim. As applications we give hash- and point-time-locked contracts and cross-chain swaps. A Lightning-style channel and streaming payments built from VCLs, and a blind variant of hashFROST, are developed in companion papers. Under current consensus, the lock time of a shielded-only transaction is not enforced. We use a transparent state output as an interim measure and propose a consensus change.

Contents

1	Introduction	2
2	Background	4
2.1	Notation and the Pasta curves	4
2.2	Ironwood	4
2.3	FROST and re-randomization	7
2.4	Adaptor signatures	7
2.5	Halo 2 and recursion	8
3	Threshold signing schemes for locks	8
3.1	The interface	8
3.2	preFROST	9
3.3	hashFROST	10
4	Sinsemilla over Vesta	11

5	Violet’s Constrained Locks	12
5.1	Setup	12
5.2	Circuits	12
5.3	The state output	13
5.4	Funding	14
5.5	Lock transactions and the checks	14
5.6	Timelocks	15
5.7	Confirmation and completion	15
6	Security of VCLs	15
6.1	Model	15
6.2	Identifier and digest	16
6.3	What the constrainees is guaranteed	16
6.4	What the constrainer is held to	17
6.5	Privacy	18
7	Applications	18
7.1	Hash- and point-time-locked contracts	18
7.2	Payment channels	19
8	Deployment notes	19
9	Related work	21
10	Conclusion	21
A	Lean formalization	24
B	Claim status	25

1 Introduction

Payment channels, atomic swaps and conditional payments on Bitcoin rest on three script features: a 2-of-2 multisignature, a timelock that keeps a transaction invalid until a given height, and a hash lock that releases funds to whoever reveals a preimage. Zcash’s shielded pools offer none of them as script. A shielded note is spent with a single spend-authorization signature, no condition can be attached to it, and its recipient and value are hidden from everyone but the parties to it.

This paper rebuilds the three features for Zcash’s Ironwood pool, which replaced Orchard and follows its protocol [HBHW], using the v6 transaction format introduced with it (Section 2.2). The conditions of a lock are not written in on-chain code; they come from composing pieces that already exist or that we define here.

- *Joint control without script.* Spend authorization in Orchard is a re-randomized Schnorr signature (RedPallas). A 2-of-2 FROST key generation [KG20, CKGW24] therefore yields a shielded address whose spend authority is shared, and re-randomized FROST [ZIPb] produces ordinary spend signatures for it.
- *Agreeing now on transactions for later.* Since ZIP 244 [NHG] the transaction identifier commits only to a transaction’s effecting data, and for a transaction with no transparent inputs the

signature digest equals the identifier. Skyhook [Vis26] turned this into *transaction prediction*. The identifier can be computed before anyone signs, and a verifier that knows a few fields and receives opaque digests for the rest can recompute it, and so learn the transaction’s effect. Skyhook uses this across chains; we use it between the two parties of a lock. Transactions up to version 5 commit to the note-commitment tree root (the anchor), so a spend can only be predicted once the note it spends is in the tree. Transaction format v6, introduced in NU6.3 along with Ironwood, moves the anchor into the authorizing data, so a transaction that spends a note which is not yet on chain can be fully signed in advance.

- *Conditions as proofs and as signing schemes.* What a lock pays, and to whom, is checked through zero-knowledge proofs about the predicted transaction’s note commitments. When it can be spent is set by the transaction header: the expiry height, and the lock time. What releases it is set by the signing scheme: plain FROST, a threshold adaptor signature (preFROST) that completes once a discrete logarithm is revealed, or one (hashFROST) that completes once a hash preimage is revealed.

We call the resulting primitive *Violet’s Constrained Locks* (VCLs). A lock is instantiated as $\text{VCL}(\text{scheme}, \text{params}, \text{DKG_scheme}, \text{DKG_params})$, and changing the signing scheme changes what the lock does: $\text{VCL}(\text{hashFROST}, \dots)$ is a preimage lock, $\text{VCL}(\text{preFROST}, \dots)$ a witness lock.

Relation to Skyhook. Skyhook [Vis26] introduced transaction prediction and open verification for Zcash, and used them for trustless cross-chain swaps from a fully scriptless pool, with a contract on another chain checking inclusion through a light client. VCLs keep the prediction and the open check, but apply them on one chain, between the two parties of a jointly controlled address. The new elements are:

- pre-signed spends of notes that are not yet mined;
- release conditions set by the threshold signing scheme;
- timelocks from the transaction header;
- predicates over hidden outputs, enforced by recursive proofs.

A payment channel built from such locks, **ELSA**, is developed in a companion paper [ELS26], and streaming payments that neither party has to watch in another [STR26].

Contributions.

- Syntax and security notions for VCLs (Sections 5 and 6): payment binding, mutual exclusion of a lock’s transactions, exclusivity of the locked funds, timelock windows, and a game-theoretic guarantee about the funding party.
- Two FROST variants (Section 3). preFROST is a threshold adaptor signature, with the re-randomization Zcash needs. hashFROST ties the adaptor point to a hash preimage through a proof. A blind variant, following the commit-first order of Fuchsbaauer and Wolf [FW24], is the subject of a companion paper [BHF26].
- Sinsemilla over the Vesta curve (Section 4), so that circuits on the Pallas side of the Pasta cycle can hash Ironwood note commitments natively.

- Applications (Section 7): hash- and point-time-locked contracts and cross-chain swaps, worked through end to end.
- Machine-checked algebra. The algebraic core of the signing schemes and several structural lemmas are proved in Lean 4 with Mathlib [dMU21, The20], with no `sorry` and an axiom audit (Appendix A). Appendix B lists, for every claim, whether it is machine-checked, proved on paper, sketched, assumed or conjectured.
- Deployment findings (Section 8). Under current consensus the lock time of a transaction with no transparent inputs is not enforced; we read this off the Zebra and zcashd source [Zca, Eleb], use a transparent “state output” as an interim measure, and propose a consensus change.

What we do not claim. The unforgeability of `preFROST` and plain `hashFROST` (P2) is given only as a proof plan. A black-box reduction to FROST’s theorem does not work, and a proof must redo FROST’s unforgeability argument with the adaptor point. Knowledge soundness of Halo 2 and of recursive proof composition is assumed, not proved. Ironwood and the v6 format are modelled as described in Section 2.2; if the deployed pool or format differs, the affected statements must be rechecked. A lock’s counterparty can recognize the lock’s transactions on chain, through the transparent state output, so we make no claim of unlinkability against it.

Organization. Section 2 recalls Ironwood and the v6 format, FROST and the proof systems. Section 3 defines `preFROST` and `hashFROST`, and Section 4 Vesta-Sinsemilla. Section 5 defines VCLs and Section 6 proves their properties. Section 7 gives the applications, Section 8 the deployment notes, and Section 9 related work.

2 Background

2.1 Notation and the Pasta curves

Groups are written additively and $[a]P$ is scalar multiplication. Pallas is the curve $y^2 = x^3 + 5$ over $\mathbb{F}_p$, with a group of prime order q ; Vesta is the same equation over $\mathbb{F}_q$, with a group of prime order p . The two form a cycle: each curve’s scalar field is the other’s base field. Both p and q are 255-bit primes slightly above 2^{254} . Ironwood keys and notes live on Pallas. A circuit over $\mathbb{F}_p$ can therefore do Pallas arithmetic natively, and its Halo 2 proofs commit to polynomials on Vesta; a circuit over $\mathbb{F}_q$ does Vesta arithmetic natively and commits on Pallas [Elea].

For a hash function we write H and give its domain separation in words. All probabilities are over the random coins of the experiment. A quantity is negligible if it vanishes faster than any inverse polynomial in the security parameter λ .

2.2 Ironwood

Assumption 2.1 (Ironwood and the v6 format). *Network upgrade NU6.3 introduced the following two components.*

1. The pool. *Ironwood, a shielded pool, follows the Orchard protocol of [HBHW]. Its keys, addresses, notes, note commitments, nullifiers, note encryption, Action statement and signatures are as in Orchard.*

2. The transaction format. *Ironwood bundles are carried by transactions of format v6. A v6 transaction commits the anchor of each shielded bundle, Ironwood's anchorIronwood included, in the authorizing-data digest instead of the transaction identifier. Transactions of versions 4 and 5 commit it in the identifier.*

The two parts are separate because the anchor change belongs to the v6 format, not to the pool; v6 treats Sapling bundles the same way.

We recall the parts of the Orchard protocol that the paper uses, which Ironwood shares; [HBHW] has the details.

Keys and addresses. A spending key determines a spend-authorizing key $\text{ask} \in \mathbb{Z}_q$, a nullifier-deriving key $\text{nk} \in \mathbb{F}_p$ and a commitment randomness $\text{rivk} \in \mathbb{Z}_q$. The spend-validating key is $\text{ak} = [\text{ask}]\mathcal{G}$ for a fixed generator $\mathcal{G}$, and ask is negated if needed so that ak has $\tilde{y} = 0$. The full viewing key is $(\text{ak}, \text{nk}, \text{rivk})$. The incoming viewing key is $\text{ivk} = \text{Commit}_{\text{rivk}}^{\text{ivk}}(\text{Extract}_{\mathbb{P}}(\text{ak}), \text{nk})$, a Sinsemilla short commitment, where $\text{Extract}_{\mathbb{P}}$ returns the x -coordinate of a Pallas point. The diversifier key dk and the outgoing viewing key ovk are the two halves of $\text{PRF}_{\text{rivk}}^{\text{expand}}([0x82] \parallel \text{ak} \parallel \text{nk})$. A diversified address is (d, pk_d) with $d = \text{FF1}_{\text{dk}}(j)$ for an index j , $g_d = \text{DiversifyHash}(d)$ and $\text{pk}_d = [\text{ivk}]g_d$. Addresses derived from one ivk cannot be linked without ivk ; an adversary who computes discrete logarithms can link them, since $\text{pk}_d = [\text{ivk}]g_d$.

Notes and note commitments. A note is $(d, \text{pk}_d, v, \rho, \psi, \text{rcm})$ with value $v < 2^{64}$. The sender picks rseed , and rcm , ψ and the ephemeral secret esk are derived from rseed and ρ with $\text{PRF}^{\text{expand}}$. The note commitment is the Sinsemilla commitment

$$\text{cm} = \text{NoteCommit}_{\text{rcm}}(\text{repr}(g_d), \text{repr}(\text{pk}_d), v, \rho, \psi), \quad \text{cm}_x = \text{Extract}_{\mathbb{P}}(\text{cm}).$$

It is binding under the discrete-logarithm assumption on Pallas and perfectly hiding.

Nullifiers. The nullifier of a note with commitment cm is $\text{nf} = \text{Extract}_{\mathbb{P}}([(\text{PRF}_{\text{nk}}^{\text{nf}}(\rho) + \psi) \bmod q] \mathcal{K} + \text{cm})$ for a fixed point $\mathcal{K}$. In each action, the new note's ρ equals the nullifier of the note spent in the same action: $\rho^{\text{new}} = \text{nf}^{\text{old}}$.

Actions and signatures. An action spends one note and creates one. Its public part is $(\text{cv}^{\text{net}}, \text{nf}, \text{rk}, \text{cm}_x, \text{epk}, C^{\text{enc}}, C^{\text{out}})$. A Halo 2 proof for the bundle shows, for each action: that the spent note's commitment is in the tree with the bundle's anchor; that nf is its nullifier; that $\text{rk} = \text{ak} + [\alpha]\mathcal{G}$ for the spender's ak and some α ; that the spent note's address satisfies $\text{pk}_d = [\text{ivk}]g_d$, with ivk computed from ak , nk and rivk given as witnesses; that cm_x is the new note's commitment with $\rho^{\text{new}} = \text{nf}$; and that cv^{net} commits to the difference of the two values with randomness rcv . Each action carries a spend-authorization signature (RedPallas, a Schnorr signature with challenge $H(R \parallel \text{rk} \parallel m)$) under rk on the signature digest m . The bundle carries a binding signature under $\sum \text{rcv}$. Spending a note therefore needs its full opening, the spender's ak , nk , rivk , a signature under rk , and every rcv of the bundle.

Note encryption. With $\text{epk} = [\text{esk}]g_d$ and key $K^{\text{enc}} = \text{KDF}([\text{esk}]\text{pk}_d, \text{epk})$, the 564-byte plaintext $(\text{leadByte}, d, v, \text{rseed}, \text{memo})$ is encrypted with ChaCha20-Poly1305 into C^{enc} . Its first 52 bytes encrypt $(\text{leadByte}, d, v, \text{rseed})$ and the next 512 bytes the memo. The recipient computes the same key as $\text{KDF}([\text{ivk}]\text{epk}, \text{epk})$. Consensus never checks C^{enc} : an output whose ciphertext is not a valid encryption is still a valid output, and whoever knows its opening can spend it.

```

txid_digest_v6
|-- header_digest
|-- transparent_digest
|   |-- prevouts_digest, sequence_digest, outputs_digest
|-- sapling_digest_v6
|   |-- sapling_spends_digest_v6
|   |   |-- sapling_spends_compact_digest
|   |   |-- sapling_spends_noncompact_digest_v6 (cv, rk)
|   |-- sapling_outputs_digest (compact, memos, noncompact)
|   |-- valueBalance
|-- orchard_digest_v6 (compact, memos, noncompact, flagsOrchard, valueBalanceOrchard)
|-- ironwood_digest_v6
|   |-- ironwood_actions_compact_digest_v6
|   |-- ironwood_actions_memos_digest_v6
|   |-- ironwood_actions_noncompact_digest_v6
|   |-- flagsIronwood
|   |-- valueBalanceIronwood

```

Figure 1: The v6 transaction-identifier tree. No pool’s digest contains an anchor: v6 commits every bundle’s anchor in the authorizing-data digest.

The v6 transaction digest. Figure 1 shows the transaction-identifier tree for v6 transactions as specified for NU6.3; it follows ZIP 244 [NHG]. Every node is a BLAKE2b-256 hash with its own personalization. For a pool with no actions, and for an empty transparent part, the node is a fixed hash of the empty input. The Ironwood node hashes three digests over all actions, plus the flags and the value balance:

- the *compact* digest, over each action’s $(nf, cm_x, epk, C^{\text{enc}}[0..52])$ in order;
- the *memos* digest, over each action’s $C^{\text{enc}}[52..564]$;
- the *non-compact* digest, over each action’s $(cv^{\text{net}}, rk, C^{\text{enc}}[564..], C^{\text{out}})$.

The header digest covers the version, the version group, the consensus branch identifier, `lock_time` and `expiry_height`.

Transaction prediction and open verification. Skyhook [Vis26] abstracts this structure as a segregated-witness ledger with three properties:

- the identifier is computed over data only, excluding every witness;
- it can be predicted from agreed parameters before any witness exists;
- it is collision-resistant.

A verifier that knows some leaves of the tree, and receives an opaque digest for every other maximal subtree, recomputes the identifier and learns that any transaction with it has the revealed effect. Skyhook calls this open verification. For a transaction that uses only Ironwood, the transparent, Sapling and Orchard nodes collapse to constants, and only the header and the Ironwood node vary. We use the prediction, the open check and this digest classification as Skyhook defines them.

Signature digest. Signatures are not part of the identifier. For a transaction with no transparent inputs the signature digest that spend-authorization and binding signatures sign equals the identifier, $\text{sighash} = \text{txid}$. With transparent inputs, the shielded signatures sign a digest in which the transparent node is replaced by one that also commits to the spent outputs’ amounts and scripts (ZIP 244’s S.2); every other node is shared with the identifier.

Consensus facts used. We rely on four rules, checked against the specification and the Zebra and zcashd source [ZIPa, Zca, Eleb].

1. A transaction with a non-zero `expiry_height` E is valid only in blocks of height at most E .
2. The header commits to the consensus branch identifier, and a transaction is valid only in the network-upgrade epoch of that identifier. A pre-signed transaction therefore becomes invalid at the next upgrade.
3. `lock_time` is enforced only if some transparent input has a sequence number other than `0xffffffff`. A transaction with no transparent inputs can be mined regardless of its `lock_time` (Section 8).
4. A nullifier, and a transparent output, can be spent at most once.

Node relay policy also rejects transparent outputs below a dust threshold, which is 54 satoshis for pay-to-script-hash outputs.

2.3 FROST and re-randomization

In FROST [KG20, CKGW24] with signer set S , signer i holds a share x_i of the group secret, with $X_i = [x_i]\mathcal{G}$ and group key $X = \sum_{i \in S} \lambda_{i,S} X_i$ for the Lagrange coefficients $\lambda_{i,S}$. Key generation is a distributed key generation with proofs of possession. Signing takes two rounds:

1. each signer publishes nonce commitments $(D_i, E_i) = ([d_i]\mathcal{G}, [e_i]\mathcal{G})$;
2. given the list B of commitments, each signer computes the binding factors $\mu_j = \text{H}_{\text{bind}}(j, X, m, B)$, the nonce $R = \sum_j (D_j + [\mu_j]E_j)$ and the challenge $c = \text{H}(R \parallel X \parallel m)$, and returns $z_i = d_i + \mu_i e_i + \lambda_{i,S} c x_i$.

The signature is $(R, \sum_i z_i)$. The binding factors tie every signer’s nonce to the message and to all commitments, which is what defeats the ROS-style attacks on concurrent sessions [BLL⁺21] that break simpler two-nonce schemes. Unforgeability has been proved under one-more discrete-logarithm assumptions in the random-oracle model [BCK⁺22, CKM23].

Spending in Zcash needs signatures under $\text{rk} = X + [\alpha]\mathcal{G}$. Re-randomized FROST [ZIPb] binds a randomizer α into the session, computes the challenge over rk , and outputs $(R, \sum_i z_i + c\alpha)$.

2.4 Adaptor signatures

An adaptor (pre-)signature for a Schnorr signature is $\hat{\sigma} = (\hat{R}, \hat{z})$ with $[\hat{z}]\mathcal{G} = (\hat{R} - T) + [c]X$ and $c = \text{H}(\hat{R} \parallel X \parallel m)$, for an adaptor point $T = [t]\mathcal{G}$. Whoever knows t completes it to the signature $(\hat{R}, \hat{z} + t)$, and anyone holding both learns $t = s - \hat{z}$ [Poe17]. The usual properties are pre-signature correctness, adaptability, witness extractability and unforgeability with pre-signing queries; we use the definitions of Aumayr et al. [AEE⁺21].

2.5 Halo 2 and recursion

Halo 2 proves PLONKish circuits with inner-product-argument polynomial commitments, without a trusted setup [BGH19, Elea]. A proof can be verified inside a circuit on the other side of the cycle, with the expensive part of the verifier deferred by accumulation; the security of this composition is analysed in [BCMS20, BCL⁺21].

Assumption 2.2 (Proof systems). *Halo 2, as instantiated for the circuits of this paper and for Ironwood’s Action circuit, is a non-interactive zero-knowledge argument of knowledge in the random-oracle model. The same holds for the recursive composition used in Section 5.2.*

3 Threshold signing schemes for locks

A VCL uses its signing scheme only through a small interface (Section 3.1), so a lock’s security proof is written once and each scheme supplies one lemma saying that it meets the interface. This section defines the two FROST variants and states what each provides.

Throughout, $\mathcal{G}$ generates the Pallas group of order q , the signer set has $|S| \geq k$, signer i holds x_i with $X_i = [x_i]\mathcal{G}$, and $X = \sum_{i \in S} \lambda_{i,S} X_i$. We write μ_j for FROST binding factors, α_{rk} for Zcash’s spend-authorization randomizer, and $rk = X + [\alpha_{rk}]\mathcal{G}$. Challenges are computed as in RedPallas, $c = H(R \parallel rk \parallel m)$, where m is the signature digest of the transaction being authorized.

3.1 The interface

Definition 3.1 (Lock signing scheme). A lock signing scheme Σ has a key generation **KeyGen** (a FROST distributed key generation in every scheme here) and a two-round signing protocol between a coordinator and the signers. Its *lock type* is one of:

- *none*: signing outputs a signature σ with $\text{Vf}(rk, m, \sigma) = 1$;
- *witness*: signing takes an adaptor point T and outputs a pre-signature $\hat{\sigma}$; $\text{PreVerify}(rk, m, T, \hat{\sigma})$ checks it, $\text{Adapt}(\hat{\sigma}, t)$ completes it with $t = \log_{\mathcal{G}} T$, and $\text{Extract}(\hat{\sigma}, \sigma)$ recovers t ;
- *preimage*: as witness, but T comes with a hash h and a proof that T is derived from a preimage of h ; **Adapt** takes the preimage, and in reveal mode **Extract** returns it.

The interface properties are:

- (U) *Unforgeability*. The adversary controls the coordinator and up to $k - 1$ signers and runs concurrent sessions. It cannot output a valid signature on a message on which no honest signer closed a session. If the honest signers closed only pre-signing sessions on a message, each with an adaptor point sampled by the experiment, it still cannot output a valid signature on that message. A pre-signature by itself proves nothing, since for an adaptor point of one’s own choosing anyone can produce one: pick $\hat{R}, \hat{z}$ and solve the pre-verification equation for T .
- (A) *Adaptability*. A pre-signature that pre-verifies completes, with the witness, to a valid signature.
- (X) *Extractability*. A pre-signature and a valid signature on the same message with the same nonce yield the witness, and in reveal mode the preimage.

A blind scheme also hides m from the signers. It provides blindness and plaintext binding, and its unforgeability becomes one signature per closed session. Blind hashFROST, which follows the

Scheme	Lock type	Properties	Status
FROST, re-randomized [ZIPb]	none	U	cited [BCK ⁺ 22, CKM23]
preFROST	witness	U, A, X	A, X machine-checked; U proof plan only (P2)
hashFROST, plain	preimage	U, A, X	A, X machine-checked; U proof plan only (P2)

Table 1: The schemes and the interface properties they provide.

commit-first order of Fuchsbauer and Wolf [FW24], is the subject of a companion paper [BHF26]. None of the schemes used here is blind.

Table 1 lists which scheme provides which property and with what status; Appendix B gives the full claim list.

3.2 preFROST

Construction 3.2 (preFROST). Signing takes a message m , an adaptor point T and a randomizer α_{rk} , all fixed before any nonce is used.

1. Each signer $i \in S$ samples (d_i, e_i) and publishes $(D_i, E_i) = ([d_i]\mathcal{G}, [e_i]\mathcal{G})$; B is the list of commitments.
2. Each signer computes $\mu_j = \text{H}_{\text{bind}}(j, X, m, B, T, \alpha_{\text{rk}})$ for $j \in S$, $R = \sum_j (D_j + [\mu_j]E_j)$, the adapted nonce $\hat{R} = R + T$ and $c = \text{H}(\hat{R} \parallel \text{rk} \parallel m)$, and returns $z_i = d_i + \mu_i e_i + \lambda_{i,Sc} x_i$.
3. The coordinator checks $[z_i]\mathcal{G} = D_i + [\mu_i]E_i + [\lambda_{i,Sc}]X_i$ for every i , which identifies a misbehaving signer, and outputs $\hat{\sigma} = (\hat{R}, \hat{z})$ with $\hat{z} = \sum_i z_i + c\alpha_{\text{rk}}$.

$\text{PreVerify}(\text{rk}, m, T, (\hat{R}, \hat{z}))$ accepts iff $[\hat{z}]\mathcal{G} = (\hat{R} - T) + [c]\text{rk}$ with $c = \text{H}(\hat{R} \parallel \text{rk} \parallel m)$. $\text{Adapt}((\hat{R}, \hat{z}), t) = (\hat{R}, \hat{z} + t)$, and $\text{Extract}((\hat{R}, \hat{z}), (\hat{R}, s)) = s - \hat{z}$.

Lemma 3.3 (P1: algebra of preFROST; machine-checked). *For every value of the hash functions:*

1. *honest signing yields a pre-signature that pre-verifies;*
2. *if $\hat{\sigma}$ pre-verifies and $T = [t]\mathcal{G}$, then $\text{Adapt}(\hat{\sigma}, t)$ is a valid Schnorr signature under rk with the same nonce and challenge;*
3. *if $\hat{\sigma} = (\hat{R}, \hat{z})$ pre-verifies and $(\hat{R}, s)$ is a valid signature with the same challenge, then $[s - \hat{z}]\mathcal{G} = T$;*
4. *adding $c\alpha_{\text{rk}}$ turns a pre-signature under X into one under $X + [\alpha_{\text{rk}}]\mathcal{G}$.*

Proof. Each item is a linear identity in a $\mathbb{Z}_q$ -module. They are `prefrost_correct`, `adapt`, `extract` and `rerandomize` in the Lean development (Appendix A). For instance, item 2 is $[\hat{z} + t]\mathcal{G} = (\hat{R} - T) + [c]\text{rk} + T = \hat{R} + [c]\text{rk}$. $\square$

Theorem 3.4 (P2: unforgeability of preFROST; proof plan only). *Under the assumptions of FROST’s unforgeability proofs [BCK⁺22, CKM23], preFROST satisfies (U). This holds against an adversary that controls the coordinator and up to $k - 1$ signers, runs concurrent sessions, and chooses T and α_{rk} for each session before the nonces are revealed.*

Why there is no black-box reduction. One might forward each preFROST session to a FROST signing oracle. But that oracle computes its challenge at the nonce R , whereas preFROST needs the challenge at $\hat{R} = R + T$, and the reduction cannot program the FROST challenger’s random oracle. The obvious repair is to fold T into a corrupt signer’s nonce commitment. It fails too: different pairs (B, T) fold to the same commitment list, so the reduction cannot answer binding-factor queries on (B, T) as a random function and remain consistent with the FROST oracle.

Proof plan. A proof must redo a FROST unforgeability proof, for instance the one-more-discrete-logarithm reduction of [BCK⁺22], with the adaptor point carried through. The reduction controls H and H_{bind} , simulates the honest signers as in that proof, and treats T and α_{rk} as extra binding inputs. Binding them means that, once the commitments are fixed, the adversary controls $\hat{R}$ no more than it controls the message in FROST. Two cases then remain for a forgery on m^* :

- no honest session closed on m^* : as in FROST’s proof;
- m^* was only pre-signed, under an adaptor point T^* sampled by the experiment. With that session’s nonce, Lemma P1(3) yields $\log_{\mathcal{G}} T^*$, a separate discrete-logarithm reduction; with any other nonce, the first case applies.

We have not carried this out.

Remark 3.5 (Extraction needs the same nonce). Schnorr signatures are not unique: the signers together can sign m again with fresh nonces, and nothing can be extracted from that signature. A party that relies on extraction must refuse every later signing session on the same m . Since ZIP 244 this matters, because signatures are outside the identifier: a re-signed transaction has the same identifier and is just as valid.

3.3 hashFROST

hashFROST fixes how the adaptor point is produced: T is derived from a preimage of a hash h , and a proof convinces every signer of the link before it signs.

Locks. H_{lock} is the hash lock (SHA-256 for interoperability). In *reveal mode* the preimage is a scalar and completing the signature reveals it; in *hide mode* $t = H_2(\text{pre})$ and completing reveals only t . Pallas’ q is slightly above 2^{254} , so a uniformly random 32-byte preimage is a canonical scalar only about a quarter of the time. We trim: $t_{\text{reveal}}(\text{pre}) = \text{pre} \bmod 2^{254}$, which is always below q . The lock relation is

$$R_{\text{link}} = \{((h, T, \text{mode}), \text{pre}) : H_{\text{lock}}(\text{pre}) = h \wedge [t_{\text{mode}}(\text{pre})]\mathcal{G} = T\}.$$

Lock samples pre (a 254-bit value suffices in reveal mode), computes h , t , T and a proof π_{link} ; VerifyLock checks the proof.

Lemma 3.6 (H4: the trim; machine-checked). $2^{254} < q$, so $\text{pre} \bmod 2^{254}$ is a canonical Pallas scalar for every pre . Every $\text{pre} < 2^{256}$ equals $b \cdot 2^{254} + (\text{pre} \bmod 2^{254})$ for some $b < 4$. Whoever extracts t therefore recovers pre with at most four evaluations of H_{lock} , and the result is unique unless H_{lock} has a collision among the four candidates.

Proof. The arithmetic is `two_pow_254_lt_pallasQ`, `trim_lt` and `trim_recover` (Appendix A). Uniqueness is collision resistance of H_{lock} . $\square$

The bytes-to-integer convention must be fixed once: the trimmed bits are the two most significant bits of the integer that is multiplied by $\mathcal{G}$, under the same byte order used to read the preimage as a scalar.

Plain hashFROST. This is preFROST with T from Lock. The coordinator sends $(m, h, T, \pi_{\text{link}})$ in the first move. Each signer checks VerifyLock and the provenance of h (if h must match a lock elsewhere, the signer takes it from there, not from the coordinator). The binding factors become $\mu_j = H_{\text{bind}}(j, X, m, B, h, T, \alpha_{\text{rk}})$. Lemma P1 applies unchanged, and so does the sketch of Theorem P2.

What a hash lock buys. The party that ran Lock knows pre and can complete whenever it likes. The lock gives atomicity, not delay: completing in reveal mode publishes pre , so anything else locked to h becomes claimable. A lock that should not complete before an event needs pre to be held by someone other than the completing party.

4 Sinsemilla over Vesta

The constraint circuit of a VCL (Section 5.2) recursively verifies proofs whose polynomial commitments are Vesta points. It therefore lives on the other side of the cycle, over $\mathbb{F}_q$, where Vesta arithmetic is native and Pallas arithmetic is not. It must also bind its statement to the transaction by hashing the Ironwood note commitments cm_x . Orchard’s Sinsemilla works on Pallas points, so we define the same hash over Vesta.

Definition 4.1 (Vesta-Sinsemilla). Let $k = 10$ and $c = 253$, as in Orchard. Fix generators $Q^\vee(D) = \text{GroupHash}^\vee(\text{"vcl:VSinsemillaQ"}, D)$ and $S^\vee(j) = \text{GroupHash}^\vee(\text{"vcl:VSinsemillaS"}, j)$ for $0 \leq j < 2^k$, where $\text{GroupHash}^\vee$ hashes to Vesta with the simplified SWU map on an isogenous curve, as for Pallas. For a domain D and a message M of at most kc bits, pad M with zeros to $n = \lceil |M|/k \rceil$ chunks $m_1, \dots, m_n$ of k bits, set $\text{Acc}_0 = Q^\vee(D)$ and $\text{Acc}_i = (\text{Acc}_{i-1} \oplus S^\vee(m_i)) \oplus \text{Acc}_{i-1}$, where $\oplus$ is incomplete addition, which fails on exceptional inputs. Then $\text{SinsemillaHash}^\vee(D, M) = \text{Extract}_\vee(\text{Acc}_n)$, the x -coordinate in $\mathbb{F}_q$, or $\perp$ if some addition failed.

The circuit hashes the list $(\text{cm}_{x1}, \dots, \text{cm}_{xN})$ of a transaction’s output commitments by a length-prefixed chain:

$$\begin{aligned} h_0 &= \text{SinsemillaHash}^\vee(D_{\text{len}}, \text{I2LEBSP}_{64}(N)), \\ h_j &= \text{SinsemillaHash}^\vee(D_{\text{chain}}, h_{j-1} \parallel \text{cm}_{xj}) \quad (1 \leq j \leq N). \end{aligned}$$

Each step hashes $2 \cdot 255 = 510$ bits, below the limit $kc = 2530$; a single call could take at most nine commitments. Each cm_{xj} is an element of $\mathbb{F}_p$. It enters the $\mathbb{F}_q$ circuit as an integer below $p < q$, which must be range-checked to be canonical.

Theorem 4.2 (S1: collision resistance; proved, except the exceptional-case step, which is cited). *If the generators $Q^\vee(D)$ and $S^\vee(j)$ have no known discrete-logarithm relation (for instance, $\text{GroupHash}^\vee$ modelled as a random oracle and discrete logarithms hard in Vesta), then finding $M \neq M'$ with $\text{SinsemillaHash}^\vee(D, M) = \text{SinsemillaHash}^\vee(D, M') \neq \perp$ is hard. So is finding two distinct lists with the same chain value.*

Proof. Without exceptional additions, $\text{Acc}_n = [2^n]Q^\vee(D) + \sum_{i=1}^n [2^{n-i}]S^\vee(m_i)$. Write this as $[2^n]Q^\vee + \sum_j [a_j]S^\vee(j)$ with $a_j = \sum_{i:m_i=j} 2^{n-i}$. Since n is read off the first coefficient, and each position i contributes the bit $n - i$ to exactly one a_j , the integer coefficient vector determines the message: the map from messages to coefficient vectors is injective.

Equal x -coordinates mean $\text{Acc}_n = \pm \text{Acc}'_n$. Hence $[2^n \mp 2^{n'}]Q^\vee + \sum_j [a_j \mp a'_j]S^\vee(j) = 0$. With the minus sign, injectivity makes the integer vector nonzero. With the plus sign, the first coefficient

is already nonzero. Every entry has absolute value at most $2^{c+1} = 2^{254}$, and Vesta’s order p exceeds 2^{254} . The vector is therefore nonzero modulo p , and it is a nontrivial discrete-logarithm relation among the generators. Exceptional additions are handled by Orchard’s analysis [HBHW], which shows that they too yield such relations. We expect it to carry over with Vesta’s order in place of Pallas’, but have not rechecked it line by line; this is the one cited step. The chain is collision-resistant by the usual Merkle–Damgård argument, since the length is fixed by h_0 and each step’s input has a fixed length. $\square$

5 Violet’s Constrained Locks

A VCL has two parties. The *constrainer* A builds the lock’s transactions and the *constraine* B checks and co-signs them. Funds are locked in a shielded address that A and B control together. Before the funds arrive, the two agree on the transactions that may spend them, and on what those transactions pay, to whom, and when. An instance is written $\text{VCL}(\Sigma, \text{params}, \text{DKG}, \text{params}_{\text{DKG}})$, where Σ is a lock signing scheme (Definition 3.1).

5.1 Setup

1. *Spend authority.* A and B run the 2-of-2 key generation DKG (FROST’s, in all our instances), obtaining shares x_A, x_B of a spend-authorizing key and the key ak . If ak has $\tilde{y} = 1$, both negate their shares, as Orchard requires.
2. *Viewing key.* They fix nk and rivk jointly, so that both hold the full viewing key $(\text{ak}, \text{nk}, \text{rivk})$. Either can derive ivk , dk and ovk , and hence every diversified address of the lock. The lock address is one of them.
3. *State output keys.* They exchange secp256k1 public keys Q_A, Q_B for a pay-to-script-hash 2-of-2 multisignature script st (Section 5.3).
4. *Circuits.* They fix the Action-Lock circuit and agree on the constraint circuit (Section 5.2).

The parameters params include the consensus branch identifier, the timelock heights, and the predicate the constraint circuit enforces.

5.2 Circuits

Action-Lock. This circuit is the same for every lock. It is a Halo 2 circuit over $\mathbb{F}_p$, with commitments on Vesta, for the relation

$$R_{\text{AL}} = \{((\text{cm}_x, g_d, \text{pk}_d, v), (\rho, \psi, \text{rcm})) : \\ \text{cm}_x = \text{Extract}_{\mathbb{P}}(\text{NoteCommit}_{\text{rcm}}(\text{repr}(g_d), \text{repr}(\text{pk}_d), v, \rho, \psi))\}.$$

A proof says that cm_x commits to a note of value v for the address (g_d, pk_d) . Skyhook’s optional privacy layer [Vis26] is the precedent: in Groth16 over BLS12-381, it proves that a hidden path hashes to the identifier and that its output commitment opens to a committed address and value. Action-Lock proves only the opening. The identifier is recomputed in the clear by the checking party, and the circuit lives on the Pasta cycle, where Sinsemilla is native rather than emulated. In an Ironwood action, ρ is the nullifier of the note spent by the same action, which the checking party sees in the compact data. We therefore recommend the variant with ρ in the statement, together with the check $\rho = \text{nf}$. This costs no privacy, and it rules out proofs about a commitment that no valid action could create.

Constraint circuit. B supplies the proving and verifying keys of a circuit that encodes what the lock must satisfy, and A accepts the verifying key after recomputing it from the agreed circuit description. Since Halo 2’s verifying key is a deterministic function of the circuit, this check is exact. The circuit is over $\mathbb{F}_q$, with commitments on Pallas, and relates:

- a public value h , the Vesta-Sinsemilla chain hash (Section 4) of the transaction’s output commitments $\mathbf{cm}_{x1}, \dots, \mathbf{cm}_{xN}$, together with the accumulator of the recursive verification;
- private inputs: the commitments, and one Action-Lock proof for each of them, with its public values $(g_d, \mathbf{pk}_d, v)$.

It verifies every Action-Lock proof recursively and checks the chain hash. It then checks a predicate on the private values: for example, that the outputs paying the lock address sum to an exact amount, or lie in a range. It may also check that the first 52 bytes of each ciphertext encrypt that note’s opening to the stated address, which we call *recoverability*. Whether a lock requires recoverability is a choice: a lock that burns funds must not.

Practical notes. Recursive verification needs an algebraic Fiat–Shamir transcript for the Action-Lock proofs, and $\mathbb{F}_p$ arithmetic that is either emulated or deferred to the other side of the cycle. The final multi-scalar multiplication of the inner product argument is accumulated, and the party checking the proof also checks the accumulator (Assumption 2.2). A recoverability check inside this circuit is non-native, because it uses Pallas operations. When the checking party holds the lock’s viewing key, it can instead trial-decrypt the ciphertexts itself (Section 5.7).

5.3 The state output

Under current consensus a transaction’s `lock_time` binds only if some transparent input has a non-final sequence number (Section 2.2). Every VCL therefore carries a transparent *state output*: an output of at least 54 satoshis, the relay-policy dust threshold, locked by `st`, the 2-of-2 script of Q_A and Q_B . The funding transaction creates it, and every transaction of the lock spends it, which gives three properties.

- *Timelocks.* A lock transaction that needs a lock time gives the state input a non-final sequence number.
- *Mutual exclusion.* All of the lock’s transactions spend the same outpoint, so at most one of them can be mined (Section 6).
- *Joint control.* The output needs both parties’ signatures, so neither party can invalidate the lock’s transactions by spending it alone.

During negotiation, a party rejects every proposed transaction of the lock that does not spend the state output.

The FROST key cannot sign transparent inputs, which use ECDSA on secp256k1; hence the separate keys. A threshold-ECDSA key [Lin17, DKLas18] would replace the visible 2-of-2 script with a single key, at the cost of a second, heavier key generation. The state output is public. It links the funding transaction to the transaction that later spends it, and spending it reveals a 2-of-2 script. We accept this leak in the instantiation, and Section 8 proposes the consensus change that removes the need for it.

5.4 Funding

A builds a funding transaction F that spends notes of its own and creates, in Ironwood, the notes to be locked, each paid to an address of the lock, together with the state output. For every action, A proves Action-Lock for the action's output. A then proves the constraint circuit for the list of output commitments.

Only A is supposed to pay into the lock. B cannot check the nullifiers of A 's inputs, which would need A 's nk , and A can spend those inputs elsewhere before F is mined. So B treats F as a promise: its guarantee begins only once the notes have arrived (Section 5.7), and if they never arrive, the lock is discarded unused.

5.5 Lock transactions and the checks

A *lock transaction* spends the lock's notes and the state output. Because the Ironwood anchor is outside the identifier, its identifier, and the digest its signatures sign, are fixed before F is mined. A builds each lock transaction as a partially created transaction, with Action-Lock proofs for its outputs and a constraint proof. B runs **Check** on the data A sends. This is Skyhook's open check [Vis26], run by the counterparty rather than by a contract on another chain, with the outputs' openings replaced by proofs:

1. *Identifier*. B recomputes txid from:
 - the header fields;
 - the transparent part;
 - the empty-pool constants for Sapling and Orchard;
 - the Ironwood node, built from every action's compact data ($\text{nf}, \text{cm}_x, \text{epk}, C^{\text{enc}}[0..52]$) together with the memos and non-compact digests as A provides them, the flags and the value balance.
2. *Header*. The version and version group are v6 , and the branch identifier is the current epoch's. lock_time and expiry_height have the values the lock requires (Section 5.6).
3. *Transparent part*. The only transparent input is the state output, with a non-final sequence number if a lock time is required; the transparent outputs are the agreed ones.
4. *Outputs*. The constraint proof verifies against the chain hash of the outputs' cm_x , and, in the recommended variant, each output's ρ equals its action's nf .
5. *Signature digest*. B recomputes the digest it is asked to sign from the same data. It differs from txid only in the transparent node, which also commits to the state output's amount and script. B also checks that this digest belongs to the transaction whose identifier it recomputed.

If every check passes, A and B run Σ 's signing protocol on the signature digest, and each keeps the result. They also exchange their ECDSA signatures for the state input.

B uses the memos and non-compact digests as A provides them. A wrong value cannot make an invalid transaction acceptable: no valid transaction has that identifier unless BLAKE2b collides, so the lock transaction simply cannot be mined. A memo stays hidden from B until the transaction is published. It stays hidden only if B cannot guess it: B can compute the note key, encrypt a guess and compare it with the memos digest. Another action whose memo ciphertext B does not know also hides it.

5.6 Timelocks

The header provides the lock’s two programmable functions:

- `enforceExpiry`: `expiry_height` E , so the transaction is valid only at heights $\leq E$;
- `checkLockTimeVerify`: `lock_time` L , so the transaction is valid only at heights $> L$, given the non-final state input.

The name follows Bitcoin’s `OP_CHECKLOCKTIMEVERIFY` [Tod14]; the mechanism is an absolute timelock on a pre-signed transaction. Every lock transaction is also valid only in the current network-upgrade epoch. Zcash has no relative timelocks [FBDk15]: a non-final sequence number only switches `lock_time` on.

5.7 Confirmation and completion

Confirmation. Once F is mined, B trial-decrypts the outputs paid to the lock with the lock’s `ivk`. It checks that their commitments are the cm_x it verified, and that the state output exists. B ’s guarantee starts here. A balance check would not be enough: A could fund the lock through another transaction with the right total but notes that fail the lock’s other conditions. Matching the commitments also matters for any lock transaction signed in advance, which spends specific notes.

Completion. To be mined, a lock transaction needs more than signatures:

- an anchor that contains the spent notes, chosen at broadcast time;
- Action proofs, which need the spent notes’ openings (available to both parties through the shared viewing key), the outputs’ openings, and every `rcv`;
- a binding signature under $\sum \text{rcv}$;
- the ECDSA signatures for the state input.

The party that built the transaction knows the outputs’ openings and the `rcv`, and no one else does unless it is told. So a lock transaction can be completed only by its builder, or by a party the builder has handed this *completion data* to. B should check the completion data against the identifier when it receives it. Applications choose who completes what (Section 7).

6 Security of VCLs

6.1 Model

Ledger. The ledger is a sequence of blocks with heights. A transaction is valid in a block of height η if it satisfies the consensus rules of Section 2.2:

- its proofs and signatures verify;
- none of its nullifiers or transparent outpoints has been spent before;
- its branch identifier is the epoch’s;
- $\eta \leq E$ for a non-zero `expiry_height` E ;
- $\eta > L$ for a non-zero `lock_time` L , if some transparent input is non-final.

We assume that a transaction an honest party broadcasts while it is valid is in the ledger within Δ blocks, if it stays valid that long. This synchrony assumption is used only for timelock arguments.

Assumptions.

- Assumption 2.1 (Ironwood and the v6 format).
- Assumption 2.2 (knowledge soundness and zero knowledge of Halo 2 and of the recursion).
- Collision resistance of the BLAKE2b-256 nodes of the identifier tree.
- Binding of NoteCommit, from the discrete logarithm on Pallas.
- Collision resistance of Vesta-Sinsemilla (Theorem 4.2).
- Unforgeability of the lock signing scheme (property U) and of ECDSA for the state input.

Adversaries. For B 's guarantees the adversary controls A , and for A 's it controls B . An honest B signs with the lock's key only digests that passed Check. This rule is part of the protocol, and it makes B 's share a gate.

6.2 Identifier and digest

Lemma 6.1 (V1: the identifier binds the effecting data; structure machine-checked). *Every node of the identifier tree hashes a fixed-length encoding of its children, or a concatenation of fixed-length records. The map from a transaction's effecting data to its tree is therefore injective. Hence two transactions with different effecting data and the same identifier give a BLAKE2b collision at some node.*

Proof. The usual induction on the tree: at the root the encodings are equal, so either the children are equal or the root hash collides. Recurse into equal children's digests, which either collide or have equal preimages, down to the leaves. A list of fixed-length records determines the records. The tree argument is `txid_inj` in the Lean development, with the hash functions `abstract`. $\square$

Lemma 6.2 (V2: the signature digest binds the same data; machine-checked). *For fixed effecting data, the signature digest and the identifier share the header, Sapling, Orchard and Ironwood nodes. They differ only in the transparent node, which in the signature digest additionally commits to the spent outputs' amounts and scripts. Hence a collision-free signature digest determines the Ironwood node, and so the outputs. A party that recomputes both digests from the same data and signs the first has signed the transaction that has the second as its identifier.*

Proof. By the definitions, with collision resistance as in V1: `sighash_determines_txid` in Lean shows that equal signature digests give a collision or equal identifiers. $\square$

6.3 What the constrainees is guaranteed

Theorem 6.3 (V3: payment binding; proved modulo the assumptions). *Suppose an honest B accepted a lock transaction with identifier τ and signature digest m , and the constraint proof's predicate held. Except with negligible probability, every transaction in the ledger that carries a valid spend-authorization signature under the lock's key on m has identifier τ . Each output commitment cm_x that B verified then commits to a note with the verified (g_d, pk_d, v) .*

Proof. A transaction in the ledger with a valid signature on m has m as its signature digest, since the signature verifies on that digest. By V2 and collision resistance, its Ironwood node is the one B hashed, and so are its header and transparent nodes. Hence its identifier is τ , and its outputs are

the commitments B saw. Knowledge soundness lets us extract an opening of each cm_x from the Action-Lock proofs inside the constraint proof (Assumption 2.2). Ironwood’s Action proof shows the transaction’s actual output note opens the same cm_x . Binding of `NoteCommit` makes the two openings equal, except with negligible probability. $\square$

Theorem 6.4 (V4: the predicate holds; proved modulo the assumptions). *In the situation of V3, the predicate checked by the constraint circuit holds for the opened notes of the mined transaction.*

Proof. The constraint proof’s public value is the chain hash of the outputs’ cm_x . By Theorem 4.2 it determines the list. Knowledge soundness of the recursive composition gives witnesses for which the predicate holds, and by V3 these are the openings of the mined outputs. $\square$

Lemma 6.5 (V5: mutual exclusion; machine-checked). *At most one transaction that spends the lock’s state output is in the ledger. In particular, at most one of the lock’s transactions is mined.*

Proof. A transparent outpoint can be spent at most once, and every lock transaction spends the state output. This is `mutual_exclusion` in Lean, for an abstract ledger in which no outpoint is spent twice. $\square$

Theorem 6.6 (V6: exclusivity; proved modulo the cited results). *Except with negligible probability, every transaction in the ledger that spends a note of the lock is one that B accepted in Check.*

Proof. Spending a note of the lock needs a signature under $\text{rk} = \text{ak} + [\alpha]\mathcal{G}$ for some α , by the Action statement. An honest B contributes its share only to digests that passed `Check`. By property U of the signing scheme, which covers re-randomized keys, a valid signature on any other digest is a forgery. The state output adds the same guarantee for ECDSA. $\square$

Theorem 6.7 (V8: timelock windows; machine-checked). *A lock transaction whose state input is non-final, with `lock_time` L and `expiry_height` E , is valid exactly at heights η with $L < \eta \leq E$ within its branch epoch; a zero field imposes no bound. Two lock transactions with $E_1 \leq L_2$ are never valid at the same height. With synchrony, a party has a window of $E_1 - \eta_0$ blocks, from the height η_0 at which it can first broadcast, to get the first mined, and the second becomes valid only after that window closes.*

Proof. Interval arithmetic on heights: `window_iff` and `windows_disjoint` in Lean. $\square$

6.4 What the constrainer is held to

The protocol leaves some things to incentives rather than cryptography. B does not check C^{enc} unless the lock asks for recoverability, and it cannot check that A ’s inputs are unspent. B does not need to: its guarantee begins at confirmation, and it releases only notes that match what it checked.

Definition 6.8 (Funding game). A chooses one of three actions:

- *honest*: fund the lock with the checked transaction;
- *deviate*: send the lock notes of total value $V' \geq 0$ that do not match the checked commitments;
- *abort*: never fund, for example by spending its inputs elsewhere.

B plays the protocol strategy: it delivers its side of the deal, worth w to A , if and only if confirmation succeeds, and it never signs a spend of notes that failed confirmation. A ’s payoffs are:

- honest: $w - V + r$, where V is the value locked and r what the lock's agreed transactions return to A ;
- abort: 0;
- deviate: $-V'$, since the notes are frozen in the 2-of-2 address.

A is individually rational if it enters only locks with $w - V + r \geq 0$.

Theorem 6.9 (V7: no profitable deviation for the constrainer; machine-checked in the model). *In the funding game, if A is individually rational, then against B 's protocol strategy no action of A yields more than the honest one.*

Proof. $w - V + r \geq 0 \geq -V'$. This is `no_profitable_deviation` in Lean. $\square$

The game is deliberately simple. It shows that the parts of the protocol left to incentives, namely funding as promised and paying only through the checked transaction, cost a rational A nothing to follow. It does not model griefing, where A accepts a loss to harm B , and in a 2-of-2 lock such a loss falls on A .

6.5 Privacy

B sees the identifier, every action's compact data, and the header. With the lock's viewing key it can also decrypt every note paid to the lock. Against B we claim only what zero knowledge gives:

- the proofs reveal nothing about the openings of outputs that B cannot decrypt, such as A 's change, beyond the predicate;
- a memo stays hidden from B until the transaction is published, provided it is unpredictable to B (Section 5.5).

Against third parties, amounts and addresses stay shielded. The state output, however, is public: it links the funding transaction to the spending transaction, and reveals a 2-of-2 script when spent. In Ironwood an output's ρ is the nullifier of the note spent in the same action, so hiding ρ from B in the Action-Lock proof gains nothing once B has seen the compact data.

7 Applications

7.1 Hash- and point-time-locked contracts

A locks a value v for B . B may claim it by revealing a preimage of h , or the discrete logarithm of T , until a deadline; after that, A may take it back. The hash-locked version is `VCL(hashFROST, ...)` with plain `hashFROST`, and the point-locked version is `VCL(preFROST, ...)`. Whoever knows the secret runs `Lock`; in a payment that is B , who hands $(h, T, \pi_{\text{link}})$ to A .

Transactions. Both spend the lock's notes and its state output, and both are fully negotiated before A funds the lock.

- *Claim C:* pays v to B , with `expiry_height` E_C . A and B produce a pre-signature on its digest for the adaptor point T , and B keeps it. A also gives B the claim's completion data, namely its effecting data, the output openings and every `rcv`, together with A 's ECDSA signature for the state input; B checks them against the identifier. Without this, only A could finish the claim (Section 5.7).

- *Refund R*: pays to A , with `lock_time` $L_R \geq E_C$ and a non-final state input. A holds a full signature.

Because both are signed before funding, neither party can hold the other’s funds hostage.

Theorem 7.1 (A1; sketched). *Assume synchrony with delay Δ , and let η_0 be the height at which the lock is confirmed.*

1. (Claim.) *If B knows the secret and broadcasts the completed claim by height $E_C - \Delta$, the claim is mined.*
2. (Refund.) *If the claim is not mined by E_C , A can get the refund mined after L_R .*
3. (Atomicity.) *If the claim is mined, A learns the secret: t from the signature, and in reveal mode the preimage. This needs A never to have signed the claim’s digest in another session.*
4. (Exclusivity.) *No transaction other than C or R spends the locked notes, and at most one of them is mined.*

Proof sketch. 1. The claim is valid up to E_C (V8), and B can complete it alone with the completion data, the adaptation of Lemma P1(2), and the ECDSA signatures.

2. The refund is valid from $L_R + 1 \geq E_C + 1$ (V8) and is already signed.

3. The published signature reuses the pre-signature’s nonce, since no other session on the digest exists (Remark 3.5), so Lemma P1(3) and Lemma H4 apply.

4. V5 and V6.

The claims are those of [AEE⁺21] for scriptless contracts. We have not written a proof that covers the network model in full. $\square$

Cross-chain swaps. With H_{lock} the SHA-256 function used by Bitcoin’s hash locks, a reveal-mode hashFROST lock pairs with an HTLC on another chain. The party that chose the preimage either samples it below 2^{254} , or proves the trimmed relation of Lemma H4 so that a 256-bit preimage can be used. The usual ordering of timeouts across the two chains applies.

7.2 Payment channels

A Lightning-style channel can be built from VCLs. It has two-level commitments, revocation through plain hashFROST pre-signatures, and a resolution deadline in place of relative timelocks, which Zcash lacks. This channel, ELSA, is developed in a companion paper [ELS26]. That paper uses payment binding (V3), mutual exclusion (V5), exclusivity (V6) and the timelock windows (V8) as black boxes.

8 Deployment notes

Lock times on shielded-only transactions. Zcash inherits Bitcoin’s finality rule: a transaction whose lock time has not passed is still final if every transparent input has sequence number `0xffffffff`. Bitcoin Core states the exemption in a comment on `IsFinalTx`. zcashd’s `IsFinalTx` runs the same loop over the transparent inputs [Eleb]. In Zebra [Zca], `Transaction::lock_time` returns no lock time unless some transparent input has a non-final sequence number, and

`lock_time_has_passed` accepts a missing lock time without comparing heights; the transaction verifier calls it for every transaction. A transaction with no transparent inputs has no sequence number to make non-final, so it can be mined regardless of its `lock_time`. Bitcoin never meets this case, because every non-coinbase transaction has an input; Zcash meets it for every shielded-only transaction. (Lean: `lock_ignored_if_final` states the rule in the model.)

This is a consensus rule, not an implementation bug: a node that enforced lock times on shielded-only transactions on its own would split from the others. We propose a consensus change for v6 transactions:

- a non-zero `lock_time` binds whether or not the transaction has transparent inputs;
- optionally, relative lock times in the style of BIP 68 [FBDk15], which ELSA-style channels would use to renew their dispute window.

Until such a change, VCLs use the transparent state output of Section 5.3.

Dust. Relay policy rejects transparent outputs other than `OP_RETURN` outputs below the dust threshold. Zebra computes it, following `zcashd`, as $3\lfloor 100(\text{size} + 148)/1000 \rfloor$ zatoshis, which is 54 zatoshis for pay-to-public-key-hash and pay-to-script-hash outputs. A zero-value state output would not relay; it must hold at least 54 zatoshis. Policy also rejects bare multisignature outputs, hence pay-to-script-hash.

Network upgrades. Every pre-signed transaction is valid only in its branch identifier’s epoch. A lock’s lifetime ends at the next upgrade, and long-lived protocols must re-sign their latest state before it. Revoked states need not be re-signed, since their transactions die with the epoch too.

Expiry. Wallets set a short expiry by default. Lock transactions must set `expiry_height` explicitly, either to 0 or to the end of the lock’s window, and the checking party verifies it.

Transaction prediction. Predicting a spend before the spent note is mined depends on the anchor being outside the identifier, which the v6 format provides (Assumption 2.1). In v4 and v5 transactions, which commit the anchor in the identifier, the same protocols work only after the funding transaction is mined and an anchor containing its notes exists, which forfeits the offline property.

Exchanging transactions. Lock transactions are exchanged as partially created Zcash transactions (PCZTs). The checking party needs every action’s compact data, the memos and non-compact digests, the header, the transparent part, and the state output’s amount and script. The full non-compact data are needed only by whoever completes the transaction.

Privacy. The state output is visible to everyone. It links a lock’s funding transaction to the transaction that spends it, and spending it reveals a 2-of-2 script. A threshold-ECDSA key would remove the script, but not the link. Removing the state output altogether needs the consensus change above.

9 Related work

Transaction prediction. Skyhook [Vis26] is the closest prior work. It introduced transaction prediction and open verification for Zcash, and used them for trustless cross-chain swaps from a fully scriptless pool, with a light client on a contract chain checking inclusion. Its optional privacy layer hides the recipient and the value behind a Groth16 proof. VCLs apply the same prediction and open check on one chain, between the two parties of a jointly controlled address. What they add:

- pre-signed spends of notes not yet mined, which need the v6 format’s anchor outside the identifier;
- release conditions set by the threshold signing scheme;
- timelocks from the header;
- predicates over hidden outputs, through recursive proofs.

Channels built from VCLs, such as ELSA [ELS26], have no counterpart in Skyhook.

Scriptless contracts. Adaptor signatures let a signature carry a condition without a script [Poe17], and generalized channels build Lightning-style channels from adaptor signatures and minimal scripts [AEE⁺21]. These designs still use the chain’s multisignature and timelock scripts. For chains without scripts, PayMo [TMSS20] and Universal Atomic Swaps [TMMS22] replace timelocks with verifiable timed signatures and timed commitments. VCLs take a third route. Joint control comes from threshold Schnorr signing, which a shielded pool already verifies. Timing comes from the header fields the chain enforces. Conditions on hidden outputs are checked through proofs about a transaction whose identifier is fixed in advance.

Channels for Zcash. Bolt [GM17] gives anonymous payment channels for Zcash-like currencies. Our companion paper [ELS26] builds a Lightning-style channel from VCLs and compares it with Bolt and with Decker and Wattenhofer’s absolute-timelock channels [DW15]. Another [STR26] builds one-way streaming payments, in the line of Spilman’s channels, that need no revocation.

Threshold Schnorr signatures. FROST [KG20, CKGW24] and its analyses [BCK⁺22, CKM23] underlie all our schemes, and ZIP 312 [ZIPb] specifies its re-randomized use for Zcash spends. Blind Schnorr signatures fall to the ROS attack [BLL⁺21] unless the user commits before seeing the nonce, as in Fuchsbauer and Wolf [FW24]; a blind hashFROST in that style is the subject of [BHF26].

Recursion. Halo [BGH19] and the accumulation-scheme analyses [BCMS20, BCL⁺21] underlie the constraint circuit’s recursive verification.

10 Conclusion

A shielded pool without scripts can still host locks. Threshold Schnorr signing supplies the multisignature. Predictable v6 transaction identifiers let the parties sign before funding. Zero-knowledge proofs constrain the hidden outputs, the header supplies the timelocks, and the choice of signing scheme sets what releases the lock. VCLs package these pieces. Hash- and point-time-locked contracts follow directly. A Lightning-style channel [ELS26], streaming payments [STR26] and a blind signing scheme [BHF26] build on them in companion papers.

Four things remain open here:

- a full proof of the unforgeability of preFROST and plain hashFROST (P2), and of the contract properties (A1);
- a consensus change that makes lock times bind for shielded-only transactions, which would remove the transparent state output and its privacy cost;
- an implementation of the Action-Lock and recursive constraint circuits, with their costs;
- machine-checking the counting part of Vesta-Sinsemilla’s collision resistance (S1).

References

- [AEE⁺21] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostakova, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized channels from limited blockchain scripts and adaptor signatures. In *Advances in Cryptology – ASIACRYPT 2021*, LNCS. Springer, 2021.
- [BCK⁺22] Mihir Bellare, Elizabeth Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. Better than advertised security for non-interactive threshold signatures. In *Advances in Cryptology – CRYPTO 2022*, LNCS. Springer, 2022.
- [BCL⁺21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In *Advances in Cryptology – CRYPTO 2021*, LNCS. Springer, 2021.
- [BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data from accumulation schemes. In *Theory of Cryptography (TCC 2020)*, LNCS. Springer, 2020.
- [BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019.
- [BHF26] Blind hashFROST: Concurrently secure threshold adaptor signatures with hash locks. Companion paper, in preparation, 2026.
- [BLL⁺21] Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. In *Advances in Cryptology – EUROCRYPT 2021*, LNCS. Springer, 2021.
- [CKGW24] Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A. Wood. The flexible round-optimized Schnorr threshold (FROST) protocol for two-round Schnorr signatures. RFC 9591, 2024.
- [CKM23] Elizabeth Crites, Chelsea Komlo, and Mary Maller. Fully adaptive Schnorr threshold signatures. In *Advances in Cryptology – CRYPTO 2023*, LNCS. Springer, 2023.
- [DKLas18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *IEEE Symposium on Security and Privacy (S&P 2018)*, 2018.
- [dMU21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction (CADE-28)*, LNCS. Springer, 2021.

- [DW15] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with Bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems (SSS 2015)*, LNCS. Springer, 2015.
- [Elea] Electric Coin Company. The halo2 book. <https://zcash.github.io/halo2/>.
- [Eleb] Electric Coin Company. zcashd. <https://github.com/zcash/zcash>. Source code, master branch, read on 2026-10-20.
- [ELS26] ELSA: Efficient layer-2 Schnorr-based apparition. Companion paper, 2026.
- [FBDk15] Mark Friedenbach, BtcDrak, Nicolas Dorier, and kinoshitajona. BIP 68: Relative lock-time using consensus-enforced sequence numbers. <https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki>, 2015.
- [FW24] Georg Fuchsbauer and Mathias Wolf. Concurrently secure blind Schnorr signatures. In *Advances in Cryptology – EUROCRYPT 2024*, LNCS. Springer, 2024.
- [GM17] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *ACM Conference on Computer and Communications Security (CCS 2017)*, 2017.
- [HBHW] Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. <https://zips.z.cash/protocol/protocol.pdf>. Version as of 2026.
- [KG20] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In *Selected Areas in Cryptography (SAC 2020)*, volume 12804 of *LNCS*, pages 34–65. Springer, 2020.
- [Lin17] Yehuda Lindell. Fast secure two-party ECDSA signing. In *Advances in Cryptology – CRYPTO 2017*, LNCS. Springer, 2017.
- [NHG] Kris Nuttycombe, Daira-Emma Hopwood, and Jack Grigg. ZIP 244: Transaction identifier non-malleability. <https://zips.z.cash/zip-0244>.
- [Poe17] Andrew Poelstra. Scriptless scripts. Presentation slides, 2017.
- [STR26] Private streaming payments from Violet’s constrained locks. Companion paper, 2026.
- [The20] The mathlib Community. The Lean mathematical library. In *Certified Programs and Proofs (CPP 2020)*, 2020.
- [TMMS22] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. Universal atomic swaps: Secure exchange of coins across all blockchains. In *IEEE Symposium on Security and Privacy (S&P 2022)*, 2022.
- [TMSS20] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmidt, and Dominique Schröder. PayMo: Payment channels for Monero. Cryptology ePrint Archive, Paper 2020/1441, 2020.
- [Tod14] Peter Todd. BIP 65: OP_CHECKLOCKTIMEVERIFY. <https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki>, 2014.

- [Vis26] Abhinav Vishnu. Skyhook: Trustless cross-chain value transfer with a fully scriptless pool. Cryptology ePrint Archive, Paper 2026/2210, 2026.
- [Zca] Zcash Foundation. Zebra: a Zcash node in Rust. <https://github.com/ZcashFoundation/zebra>. Source code, main branch, read on 2026-10-20.
- [ZIPa] ZIP 203: Transaction expiry. <https://zips.z.cash/zip-0203>.
- [ZIPb] ZIP 312: FROST for spend authorization multisignatures. <https://zips.z.cash/zip-0312>.

A Lean formalization

The development is in `lean/` (namespace `Vc1`), and also holds the machine-checked results of the companion papers [ELS26, BHF26, STR26]. It uses Lean 4 v4.35.0 and Mathlib v4.35.0, with all dependency revisions locked in `lake-manifest.json`. It contains no `sorry`. `Audit.lean`, built by default, runs `#print axioms` on every result below. Each depends only on Lean’s standard axioms `propext`, `Classical.choice` and `Quot.sound`, and the build script `ci.sh` fails otherwise.

Paper	Lean declaration	File
Lemma P1	<code>prefrost_correct</code> , <code>adapt</code> , <code>extract</code> , <code>rerandomize</code>	<code>Schnorr</code>
H1 of [BHF26]	<code>blind_preverify</code> , <code>blind_consistent</code>	<code>Schnorr</code>
Lemma H4	<code>two_pow_254_lt_pallasQ</code> , <code>trim_lt</code> , <code>trim_recover</code>	<code>Trim</code>
Lemma V1	<code>txid_inj</code>	<code>Txid</code>
Lemma V2	<code>sighash_determines_txid</code>	<code>Txid</code>
Lemma V5	<code>mutual_exclusion</code>	<code>Ledger</code>
Theorem V7	<code>no_profitable_deviation</code>	<code>Game</code>
Theorem V8	<code>window_iff</code> , <code>windows_disjoint</code>	<code>Ledger</code>
Section 8	<code>lock_ignored_if_final</code>	<code>Ledger</code>
E1 of [ELS26], timing	<code>penalty_in_time</code> , <code>clean_in_time</code> , <code>commitment_by_deadline</code> , <code>wr_cr_disjoint</code> , <code>commit_he_disjoint</code>	<code>Elsa</code>
SP1–SP3 of [STR26]	<code>linearStream_mono</code> , <code>linearStream_le</code> , <code>linearStream_start</code> , <code>linearStream_end</code> , <code>claim_pays</code> , <code>total_claimed</code> , <code>claims_then_final</code> , <code>within_schedule</code> , <code>claim_window</code> , <code>claim_final_disjoint</code> , <code>final_in_time</code>	<code>Stream</code>

Modelling choices.

1. *Group*. An additive commutative group that is a module over $\mathbb{Z}/q$, with a fixed element g . The identities hold for every q ; primality is not used.
2. *Hashes*. Challenges are arbitrary scalars, so the algebraic lemmas hold for every value of the hash functions. In `Txid`, each node hash is an arbitrary function of the tuple of its children, so node encodings are injective by construction, and a list of fixed-length records is a `List`. Collision resistance is not formalized: the lemmas say “a collision, or the conclusion”.
3. *Ledger*. A list of transactions, each reduced to the finite set of outpoints and nullifiers it spends, with no element spent twice.

4. *Timelocks.* Heights are natural numbers, and time-based lock times are not modelled. A value of 0 means the field is unset, and the lock time binds only with a non-final transparent input, as in Section 2.2. The ELSA timing lemmas are arithmetic over these heights, with the network delay Δ and the offline gap K related to the dispute window by $K + \Delta + 1 = W$.
5. *Game.* Payoffs are rationals, and B 's strategy is fixed to the protocol's.

Build. The toolchain is the swiss environment `leviackerman/lean@2`. The command is `swiss run -env leviackerman/lean@2 -private - bash ci.sh`, which runs `lake exe cache get` and `lake build` in a persistent directory on the server and checks the audit's output.

B Claim status

Each claim is marked as one of:

- *machine-checked*: proved in Lean, in the model of Appendix A;
- *proved*: proved on paper here, possibly modulo named assumptions;
- *sketched*: the argument is given but not written in full;
- *cited*: taken from the literature without rechecking;
- *target*: stated, not proved.

Claim	Statement	Status
P1	preFROST algebra: honest signing, adaptation, extraction, re-randomization	machine-checked
P2	preFROST and plain hashFROST unforgeability	proof plan only (Theorem 3.4); there is no black-box reduction to FROST, so FROST's proof must be redone with the adaptor point
H4	the 2-bit trim	machine-checked; uniqueness of the recovered preimage assumes collision resistance of H_{lock}
S1	collision resistance of Vesta-Sinsemilla	proved, assuming discrete logarithms in Vesta; the exceptional-case step is cited from Orchard
V1	the identifier binds the effecting data	machine-checked (structure); collision resistance of BLAKE2b assumed
V2	the signature digest determines the identifier	machine-checked (structure)
V3	payment binding	proved, assuming Assumption 2.2, binding of NoteCommit and collision resistance
V4	the constraint predicate holds	proved, assuming knowledge soundness of the recursion and S1
V5	mutual exclusion	machine-checked, for an abstract ledger

Claim	Statement	Status
V6	exclusivity	proved, assuming unforgeability of the signing scheme (cited for FROST) and of ECDSA
V7	no profitable deviation for the constrainer	machine-checked, in the model of Definition 6.8
V8	timelock windows	machine-checked, for the consensus rules as modelled
A1	claim, refund, atomicity and exclusivity of HTLC and PTLC	sketched
—	lock times bind only with a non-final transparent input	read from the Zebra and zcashd source; modelled in Lean (<code>lock_ignored_if_final</code>)
—	Ironwood and the v6 format	assumed (Assumption 2.1), as described to us; no specification is cited yet

Assumptions used.

- Assumption 2.1 (Ironwood and the v6 format).
- Assumption 2.2: Halo 2 and its recursion are zero-knowledge arguments of knowledge.
- Collision resistance of BLAKE2b-256 and SHA-256.
- Discrete logarithms are hard in Pallas, which gives the binding of `NoteCommit`, and in Vesta (S1).
- Unforgeability of re-randomized FROST and of ECDSA.
- Synchrony with delay Δ , for the timelock arguments.