

6EQUJ5: Shielded Pools for Bitcoin Tokens, Verified by the Indexers Themselves

6EQUJ5 Research
6equj5.xyz · @6EQUJ5OP

Design specification, version 0.1, October 2026. No component described in this paper is deployed on Bitcoin mainnet.

Abstract—Bitcoin's token layers inherit Bitcoin's transparency. Every BRC-20 balance is computed by public indexers from public inscriptions, so anyone can read the full history of every holder. The shielded token designs that appeared on Bitcoin in 2026 either protect non-fungible items with a project-run indexer or rely on address techniques that no indexer can verify. We present 6EQUJ5, a design for a shielded pool for fungible BRC-20 assets whose zero-knowledge proofs are verified inside brc2.0, the EVM that every BRC-20 indexer already executes alongside BRC-20 itself. A holder deposits a public balance and receives a private note. Later the holder withdraws to an unlinked address by proving, with a Groth16 proof checked through the bn254 pairing precompiles, that they own an unspent note in the pool and that their deposit belongs to a published association set. The construction adapts the audited Privacy Pools protocol to Bitcoin's inscription model and adds three Bitcoin-specific elements. Association sets are published as inscriptions. The trusted-setup ceremony is finalized with a pre-announced Bitcoin block hash. And a supply-safety argument shows that any soundness failure is bounded by the pool's public balance rather than becoming hidden inflation. We specify the protocol, its integration with brc2.0 and a threat model; analyze privacy leakage and costs at current fee levels; and outline a path to native shielded transfers.

Index Terms—Bitcoin, BRC-20, brc2.0, zero-knowledge proofs, Groth16, shielded pool, association sets, privacy.

I. INTRODUCTION

Bitcoin is a public ledger by design. Since 2023, Ordinals inscriptions have let users place arbitrary data in Bitcoin transactions [1]. BRC-20 used that capability to define fungible tokens as JSON inscriptions whose balances are not stored on Bitcoin at all. Indexers derive them by replaying every valid operation in block order [2]. The model is simple and has proved durable, but it is transparent by construction: every amount, address and transfer is visible and permanently linkable.

During 2026 a wave of Bitcoin projects adopted "shielded" language. On inspection they fall into three groups:

- Non-fungible items protected by real zero-knowledge proofs, which a project's own indexer and anyone who replays it can check. Shielded Ordinals (SHORD) is the open example [3].
- Fungible tokens with one-time receiving addresses, which hide nothing that an indexer can verify.
- Research toward shielded BTC on Bitcoin's base layer, such as the Shielded Bitcoin protocol, which defines

encrypted notes, nullifiers and a dedicated replay network [4].

None of these gives an *existing* fungible BRC-20 asset a privacy layer whose rules are enforced by the standard indexers that already compute its balances.

Our starting observation is that this gap can now be closed without new infrastructure. The BRC-20 programmable module, brc2.0, has been live since block 912,690. It is an EVM built on revm that every OPI-based BRC-20 indexer executes [5], [6]. Since block 923,369 it runs Ethereum's Prague rules. It loads Ethereum's standard precompiles, including bn254 point addition, scalar multiplication and the pairing check at addresses `0x06–0x08` [5], [7]. These are exactly the operations a Groth16 verifier needs. A shielded pool deployed as brc2.0 contracts is therefore verified by the same software that computes every BRC-20 balance, not by software its authors control. To our knowledge, no zero-knowledge verifier has yet been deployed on brc2.0; this paper specifies the first.

Contributions:

- A construction for a shielded pool of BRC-20 assets inside brc2.0. It adapts the audited Privacy Pools protocol [8], [9] to inscriptions, Bitcoin-script identities and an EVM without a native coin (Sections IV–VI).
- A supply-safety argument. Shielding never creates assets, so the worst case of a proof-system failure is a bounded, publicly visible drain of the pool, not undetectable inflation (Section IX-A).
- Bitcoin-anchored association sets. Approved-deposit sets are published as inscriptions, so anyone can rebuild and audit them from Bitcoin alone (Section VIII).
- A Bitcoin-anchored trusted setup. It is a sealed ceremony whose final randomness is the hash of a Bitcoin block announced in advance (Section VII).
- A leakage and cost analysis of the Bitcoin edges, including the exit-fee funding problem (Sections VI-D, X, XI). We also outline a path to native shielded transfers (Section V-G).

On the name: On 15 August 1977 the Big Ear radio telescope recorded a strong narrowband signal near the 1420 MHz hydrogen line. It appeared on the printout as the sequence 6EQUJ5, which an astronomer circled and annotated "Wow!" [10]. It was never received again. We use the name for a system whose contents sit on a public medium yet cannot

be read without the right key. In the project's narrative, entering the pool is called `OP_DARK`. `OP_DARK` is not a Bitcoin opcode and requires no change to Bitcoin. It names a sequence of standard `brc2.0` inscriptions (Section VI).

Status: This paper is a design specification and research agenda. Values marked *estimate* are our calculations, not measurements. Code listings are illustrative and simplified. Facts about third-party systems were checked against their public source code and documentation in October 2026 and may change.

II. BACKGROUND

A. Inscriptions and BRC-20

An Ordinals inscription places content in a Taproot script-path witness, using a commit and reveal transaction pair. The content is assigned to a satoshi and moves with it [1]. BRC-20 defines `deploy`, `mint` and `transfer` operations as JSON inscriptions [2]. A transfer takes two steps. The holder first inscribes a `transfer` inscription to their own address, which earmarks an amount. Sending that inscription to a recipient then moves the amount. Six-character tickers protect deployments against front-running with a *predeploy*: a double SHA-256 commitment to the ticker, a salt and the deployer's script. The deploy must reference it at least three blocks later [11].

B. The brc2.0 Programmable Module

`brc2.0` is a deterministic EVM executed by indexers. Contracts are deployed and called with inscriptions in the `brc20-prog` namespace, which are sent to an output `OP_RETURN "BRC20PROG"` [5], [12]. A holder brings a BRC-20 balance into the module by sending a normal transfer inscription to that output. The indexer then credits an ERC-20-style balance held by a per-ticker contract that the module's `BRC20_Controller` creates. Balances leave the module through a `brc20-module withdraw` inscription [5]. Five properties of the module shape our design [5], [13]:

- *Identity.* An account address is `keccak256(pkscript)[12:]`, the hash of the Bitcoin script that authorized the inscription. There are no EVM private keys, and BIP-322 signatures replace `ecrecover` [14].
- *Gas.* Each call receives 12,000 gas per byte of its inscription. There is no block gas limit, and calldata can be up to 1 MB. Computation is effectively paid for in Bitcoin block space.
- *No native coin.* `msg.value` is always zero, and the chain id is `0x4252433230` ("BRC20").
- *Finality.* The engine keeps ten blocks of reorganization history. Every step waits for confirmation, so interactions take minutes.
- *Precompiles.* The module loads Ethereum's standard set for the active fork, including `bn254` and, since Prague, `BLS12-381`. An automated market maker on `brc2.0` already verifies `BLS12-381` pairings in production [13].

C. Zero-Knowledge Proofs

A zk-SNARK lets a prover convince a verifier that a statement about secret inputs holds without revealing them. Groth16 [15] produces constant-size proofs: three group elements, or eight 32-byte words in EVM calldata. Verification is one pairing-product check plus one scalar multiplication per public input. On `bn254` this costs roughly 207,000 gas plus about 7,000 gas per public input [7], [16]. Groth16 needs a circuit-specific trusted setup, which is secure if at least one participant destroys their secret randomness [17]. Circuits hash with Poseidon, which is designed to be cheap inside arithmetic circuits [18]. We use `circom` and `snarkjs` [19].

D. Privacy Pools and Association Sets

Buterin *et al.* proposed privacy pools [8]. In them, a withdrawal proves two things: that it spends some earlier deposit, and that the deposit belongs to an *association set*, a published subset of deposits. Users can then separate themselves from deposits they do not wish to be associated with, without revealing which deposit is theirs. The implementation of [9] is licensed Apache-2.0 and has been audited for its contracts, circuits and upgrade path [20]. Each deposit receives a *label*. An association set provider (ASP) publishes the Merkle root of approved labels. A depositor who is not approved can always *ragequit*: withdraw their full deposit publicly, to the address it came from.

III. DESIGN GOALS AND THREAT MODEL

A. Goals

- G1 *Indexer-verified.* Every pool rule is enforced by `brc2.0` code executed by the standard indexers. No project-run indexer, sequencer or server sits in the trust path.
- G2 *Unlinkability.* Given a withdrawal, an observer cannot tell which approved deposit funded it, beyond the public leakage in Table II.
- G3 *Non-custodial.* Only the holder of a note's secrets can move its value. Relayers cannot redirect funds, and no one can confiscate deposits (subject to Section IX-C).
- G4 *Supply safety.* The pool can never create assets. The total supply of a pooled asset stays fixed by its deploy inscription.
- G5 *Compliance-compatible.* Association sets and ragequit let honest users keep their funds separate from known illicit deposits, and nobody can be trapped in the pool.
- G6 *Reproducible.* Notes are recoverable from a seed. Circuits, keys and contracts are reproducible, and their hashes are recorded on Bitcoin.

B. Adversaries

We consider seven adversaries:

- a passive observer who sees every inscription and all indexer state;
- a front-runner who watches the mempool and can replace transactions by fee;
- a malicious relayer;

- a malicious or careless ASP;
- a malicious depositor who attempts double withdrawals or forged proofs;
- a setup adversary who controls some ceremony participants;
- a faulty indexer that diverges from the others.

Out of scope are network-level deanonymization [21], compromised user devices, and Bitcoin reorganizations deeper than ten blocks.

IV. SYSTEM OVERVIEW

Fig. 1 shows the architecture. Bitcoin orders and stores every operation, and the brc2.0 engine, run independently by each indexer, executes the pool contracts. Four off-chain roles complete the system, and none of them holds user funds.

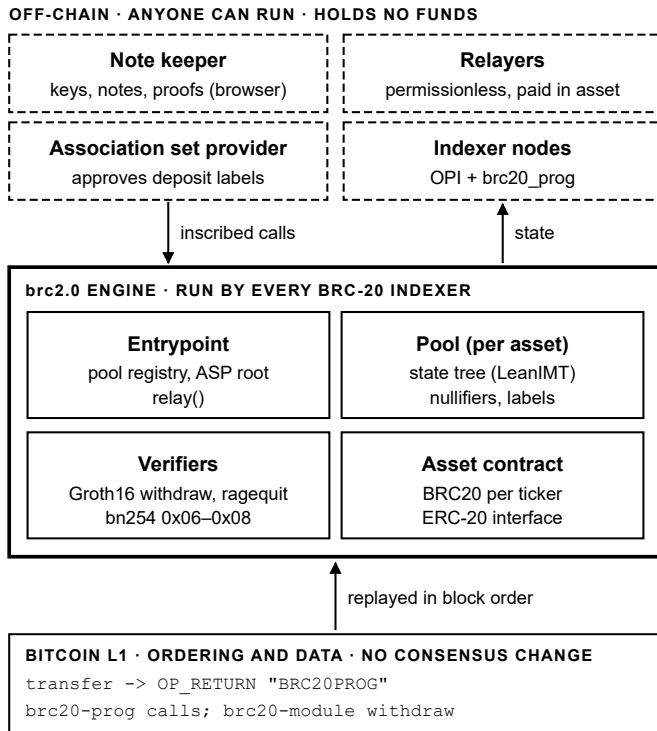


Fig. 1. Architecture. All pool rules run inside brc2.0, which every BRC-20 indexer executes independently from Bitcoin data. Off-chain roles (dashed) hold no funds and can be run by anyone.

The pool contracts are:

- an Entrypoint, which registers pools, stores the current association-set root and routes relayed withdrawals;
- one Pool per asset, holding the state tree of commitments, the set of spent nullifiers and the depositor of each label;
- two Groth16 verifiers, for withdrawal and ragequit;
- Poseidon libraries.

The off-chain roles are:

- the note keeper, a browser module that derives keys, stores notes and generates proofs;
- relayers, independent parties who submit withdrawals so the user's address never appears;

- the ASP;
- indexer nodes, which anyone can run to read state.

A holding moves through four steps (Fig. 2):

1. Enter: the public BRC-20 balance moves into the module.
2. Shield: a deposit creates a private note.
3. Withdraw: a zero-knowledge proof, submitted by a relayer, sends value to a fresh address.
4. Exit: the fresh address withdraws from the module and holds an ordinary public balance again.

Ragequit is the escape path for deposits the ASP has not approved.

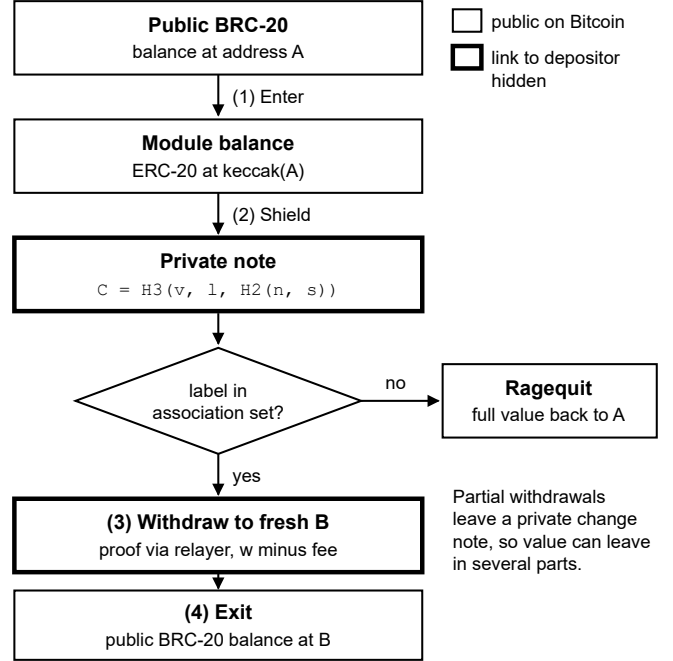


Fig. 2. Life of a holding. Steps (1) and (4) are ordinary public BRC-20 operations. Inside the pool, which deposit funds which withdrawal is hidden. Unapproved deposits can always leave by ragequit.

V. CRYPTOGRAPHIC CONSTRUCTION

Notation: p is the order of the bn254 scalar field $\mathbb{F}_p$. H_k is Poseidon with k inputs [18]. K is Keccak-256, and $[x]_p$ denotes $x \bmod p$. $\parallel$ denotes ABI concatenation. The construction follows [9] unless stated otherwise, so its audited circuits can be reused unchanged.

A. Scope, Labels and Notes

Each pool has a scope that binds proofs to one pool, one chain and one asset:

$$\text{SCOPE} = [K(\text{pool} \parallel \text{chainId} \parallel \text{asset})]_p \quad (1)$$

where $\text{chainId} = \text{"BRC20"}$. The i -th deposit is assigned the label $\ell = [K(\text{SCOPE} \parallel i)]_p$. A *note* is a tuple (v, ℓ, n, s) , where:

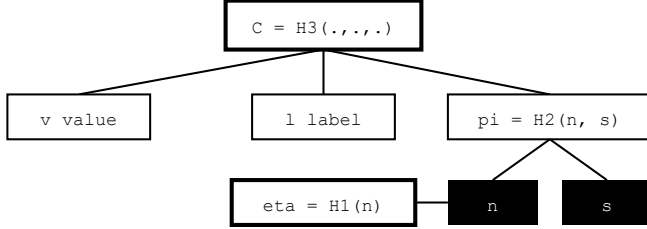
- $v < 2^{128}$ is the value;
- ℓ is the label inherited from the original deposit;
- the nullifier n and the secret s are random field elements known only to the owner.

From these the owner derives the precommitment π , the commitment C and the nullifier hash η :

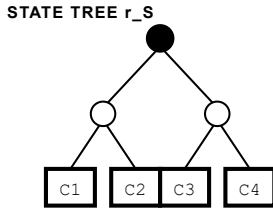
$$\pi = H_2(n, s), \quad C = H_3(v, \ell, \pi), \quad \eta = H_1(n). \quad (2)$$

When depositing, the user submits only π . The contract computes C itself, from the value it actually received and the label it assigned, so a depositor cannot misstate a note's value or label (Fig. 3).

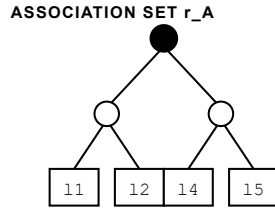
A NOTE



filled = secret, never published; bold = stored (C) or revealed when the note is spent (eta)



every commitment, append-only; last 64 roots kept



approved labels only (13 absent); root on-chain, data inscribed

Fig. 3. Top: structure of a note (l denotes the label ℓ , pi the precommitment π , eta the nullifier hash η). Bottom: the two Merkle trees in which a withdrawal proves membership.

B. Trees

Commitments are appended to a lean incremental Merkle tree (LeanIMT) [22] with Poseidon nodes and a maximum depth of 32. The pool keeps a history of the last 64 roots, so a proof built against a slightly older root stays valid while new deposits arrive. The ASP maintains a second tree over approved labels and publishes its root through the Entrypoint. A withdrawal must use the latest published root.

C. The Withdrawal Statement

A withdrawal of value w spends a note (v, ℓ, n, s) and creates a change note $(v - w, \ell, n', s')$. Its public inputs are:

- w ;
- the state root r_S and its depth;
- the association root r_A and its depth;
- a context value ctx .

Its outputs are η and C' . The circuit proves that:

1. $C = H_3(v, \ell, H_2(n, s))$ is a leaf of the tree with root r_S ;
2. ℓ is a leaf of the tree with root r_A ;
3. $w < 2^{128}$ and $v - w < 2^{128}$, so no underflow or field wraparound occurs;
4. $n' \neq n$, $\eta = H_1(n)$, and $C' = H_3(v - w, \ell, H_2(n', s'))$;
5. ctx is bound into the proof.

The pool contract accepts the proof only if four checks pass:

1. η has never been seen;
2. r_S is in the root history, and r_A equals the current association root;
3. ctx satisfies (3), where withdrawal = (processor, recipient, feeRecipient, feeBPS);
4. the proof verifies.

$$ctx = [K(\text{withdrawal} \parallel \text{SCOPE})]_p. \quad (3)$$

The contract then records η , inserts C' , and transfers w less the relay fee to the recipient. Equation (3) is what makes relaying safe. The recipient and fee terms are inside the proven statement, so a relayer, or anyone who copies the call from the mempool, can only deliver the funds exactly as the owner specified.

D. Ragequit

Ragequit proves knowledge of an opening (v, ℓ, n, s) of a commitment C in the state tree, and outputs η . The contract also requires the caller to be the address that made the deposit with label ℓ . It pays the full value v back to that address publicly. Ragequit requires no association-set membership. The ASP can therefore decline to approve a deposit, but it can never freeze it.

E. Circuit Reuse

The withdrawal circuit uses only standard components: Poseidon, two LeanIMT inclusion proofs, 128-bit range checks and an equality test. Reusing the upstream circuits byte-for-byte has three benefits:

- the existing circuit audit [20] still applies;
- the keys from the upstream ceremony can be inherited and extended (Section VII);
- the off-chain prover is independent of the chain that runs the verifier.

Listing 1 shows the commitment component.

```
pragma circom 2.1.9;
include "circomlib/circuits/poseidon.circom";

// Note commitment and nullifier hash.
// Structure follows Privacy Pools [9].
template CommitmentHasher() {
    signal input value; // range-checked, < 2^128
    signal input label; // [K(SCOPE, i)]_p
    signal input nullifier; // secret
    signal input secret; // secret
    signal output commitment;
    signal output nullifierHash;

    signal pre <== Poseidon(2)([nullifier, secret]);
    commitment <== Poseidon(3)([value, label, pre]);
    nullifierHash <== Poseidon(1)([nullifier]);
}
```

Listing 1. The note commitment in circom. The withdrawal circuit instantiates it for the spent note and for the change note.

F. Contract Checks

Listing 2 shows the withdrawal path of the pool contract, simplified from [9]. On `brc2.0, msg.sender` is the hash of the

Bitcoin script that inscribed the call. The transferred asset is the per-ticker ERC-20 balance that the module created on deposit.

```
function withdraw(
    Withdrawal calldata w,
    WithdrawProof calldata p
) external {
    // relayed calls arrive via the Entrypoint
    require(msg.sender == w.processoosor);
    // recipient and fee are bound into the proof
    require(p.context() == uint256(keccak256(
        abi.encode(w, SCOPE))) % SNARK_SCALAR_FIELD);
    require(!_isKnownRoot(p.stateRoot())); // last 64
    require(p.ASPRoot() == ENTRYPOINT.latestRoot());
    require(!spent[p.existingNullifierHash()]);
    require(VERIFIER.verifyProof( // bn254
        p.pA, p.pB, p.pC, p.pubSignals));

    spent[p.existingNullifierHash()] = true;
    _insert(p.newCommitmentHash()); // change note
    ASSET.safeTransfer(w.processoosor,
        p.withdrawnValue());
}
```

Listing 2. Withdrawal checks in Solidity, simplified from [9]. The only brc2.0-specific change is the asset: the module's ERC-20 for the pooled ticker.

G. Toward Native Shielded Transfers

The construction above hides which deposit funds which withdrawal. It does not yet support transfers between private notes without leaving the pool. A second phase adds a 2-in/2-out transfer relation in the style of Zerocash and Zcash [23], [24]:

- two input notes are spent and two output notes are created, with $\Sigma v_{in} = \Sigma v_{out}$ and every value range-checked;
- nullifiers are bound to each note's position in the tree, following [4];
- output notes are encrypted to their recipients and emitted as contract events, and recipients find them by scanning with a viewing key.

Two problems remain open. The first is how association-set membership should compose when notes with different labels are merged. A conservative rule requires every input label to be in the current set and gives outputs a derived label that the ASP can evaluate. The second is in-circuit encryption checks, which could raise the constraint count from an estimated 25,000–35,000 to 50,000–150,000 (*estimate*). A new circuit also needs its own ceremony. We leave this phase to a later version.

VI. INTEGRATION WITH BITCOIN

A. Operations as Inscriptions

Every pool interaction is an ordinary Bitcoin transaction that carries an inscription. Calls are sent to `OP_RETURN "BRC20PROG"`, which takes a second transaction after the reveal. Listing 3 shows the three forms a holder uses. The ticker is a placeholder: the pool is asset-agnostic and works for any BRC-20 balance the module can hold.

```
// Step (1), Enter: BRC-20 transfer inscription,
// then sent to OP_RETURN "BRC20PROG"
{"p":"brc-20","op":"transfer",
 "tick":"xxxxxx","amt":"10000"}

// Steps (2)-(3): shield, withdraw (relayer) and
// ragequit are brc2.0 call inscriptions
{"p":"brc20-prog","op":"call",
 "c":"0x<entrypoint or pool>",
 "b":"<compressed calldata, base64>"}

// Step (4), Exit: module withdraw at address B
{"p":"brc20-module","op":"withdraw",
 "tick":"xxxxxx","amt":"9950",
 "module":"BRC20PROG"}
```

Listing 3. Inscription forms of the protocol [2], [5]. Calldata may be hex (a) or compressed base64 (b); proof bytes are random and barely compress.

B. A Private Round Trip

Fig. 4 traces one round trip. The depositor's Bitcoin script is visible in steps (1) and (2), because `msg.sender` is derived from it. The withdrawal is therefore inscribed by a relayer. The relayer pays the Bitcoin fee for the call and is reimbursed with a fee, in the pooled asset, that the owner fixed inside the proof by (3). Relayers are permissionless: anyone can run one, the client chooses among them, and none can alter the destination or the amount. The note keeper reads pool state through any indexer's `eth_call` interface and builds the proof locally, so notes and secrets never leave the device.

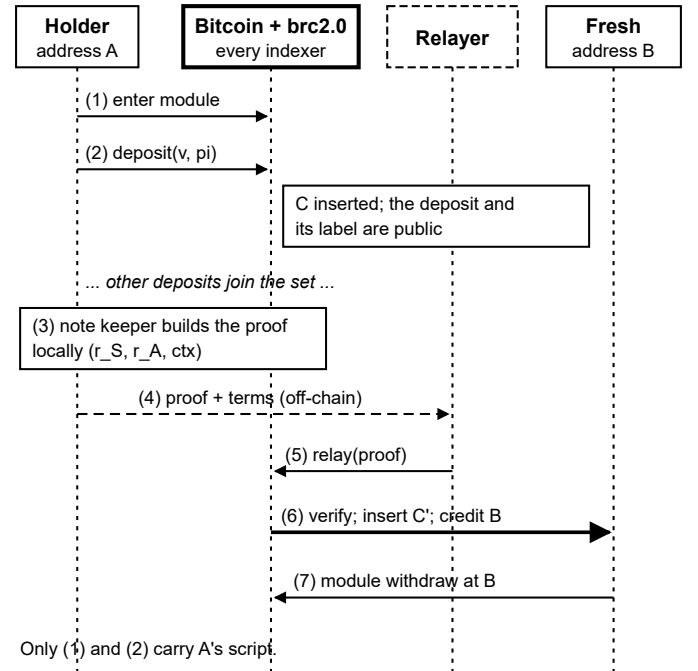


Fig. 4. A private round trip. Step (5) is inscribed by a relayer, and every indexer executes step (6) identically. Address B never interacts with A on-chain.

C. Required Adaptations

Running Privacy Pools inside brc2.0 requires four changes, none of which touch the circuits:

1. *ERC-20 pools only.* There is no native coin, so only the ERC-20 pool path is used, holding the module's per-ticker asset contract.
2. *Deployment.* Deterministic deployers used on Ethereum are replaced by plain deploy inscriptions, and contract addresses and bytecode hashes are published.
3. *Scope.* Scope derives from chain id "BRC20" by (1), so proofs cannot be replayed against the same code on another chain.
4. *Finality.* The engine keeps only ten blocks of history. Clients therefore treat deposits as final after six confirmations and compare state across at least two independent indexers.

D. The Exit-Fee Funding Problem

The largest practical leak sits at the exit, not inside the pool. Fresh address B must inscribe its own module withdrawal (step 7), which needs roughly 1,000–3,000 sats of BTC for fees at current rates (*estimate*). If B is funded from A, or from the same exchange account, the two are linked again. We recommend three mitigations:

- fund B through a Lightning-to-on-chain swap;
- keep value inside brc2.0 and use it there;
- wait, and exit in standard amounts.

We are also studying non-custodial fee sponsorship, in which a third party contributes the fee input to a transaction that B co-signs. This is not part of version 0.1.

VII. A TRUSTED SETUP ANCHORED TO BITCOIN

Groth16's security depends on a setup ceremony, and that is the first trust question any shielded system must answer. Our approach has three parts:

1. *Inherit.* We reuse the upstream circuits and their published phase-2 keys unchanged. We verify them with `snarkjs zkey verify` against the circuit and the Perpetual Powers of Tau phase-1 file [19], [25].
2. *Extend.* We open a public contribution window. Each contributor runs `snarkjs zkey contribute` on the latest key, and the transcript hashes are published.
3. *Close with Bitcoin.* The final contribution is a random beacon [17], derived from the hash of a Bitcoin block whose height H_b is fixed before the window opens. No participant can predict it.

Before the window opens, the ceremony rules are sealed. A double SHA-256 commitment to the circuit hashes, the starting key, H_b and the iteration count is inscribed on Bitcoin. This is the same commit-and-reveal pattern as a predeploy (Section II-A). After the beacon, the hashes of the final keys are inscribed. The verification key is also embedded in the bytecode of the verifier contract, which is itself deployed by inscription. Anyone can therefore check that what runs on-chain is what the ceremony produced. The setup is sound if at least one contributor, in the inherited chain or in the extension,

was honest [17]. Fig. 5 and Listing 4 summarize the procedure.

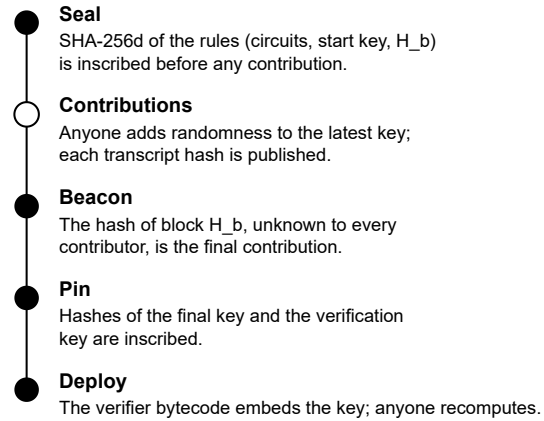


Fig. 5. The ceremony. Filled markers are records on Bitcoin. The final randomness comes from a block that no participant could predict.

```
# 0. verify inherited key (circuit, phase 1)
snarkjs zkey verify withdraw.ribs \
  ppot_final.ptau withdraw_upstream.zkey
# 1. each contributor adds randomness
snarkjs zkey contribute withdraw_k.zkey \
  withdraw_k1.zkey --name="contributor k+1"
# 2. beacon: SHA-256d of the block hash at H_b
B=$(bitcoin-cli getblockhash $H_b)
BEACON=$(echo -n "$B" | xxd -r -p | sha256sum \
  | xxd -r -p | sha256sum | cut -c1-64)
snarkjs zkey beacon withdraw_k1.zkey \
  withdraw_final.zkey $BEACON 10 \
  -n="6EQUJ5 final beacon"
# 3. export and hash the keys, then inscribe
snarkjs zkey export verificationkey \
  withdraw_final.zkey vk.json
sha256sum withdraw_final.zkey vk.json
```

Listing 4. Ceremony steps with standard snarkjs commands [19]. The beacon height is fixed in the sealed rules before any contribution is accepted.

VIII. COMPLIANCE BY DESIGN

Privacy and accountability are not opposites. The association-set model shows that a user can prove their funds come from a set of acceptable deposits without revealing which one [8]. 6EQUJ5 applies the model with five rules:

- *Bitcoin-anchored association sets.* Upstream implementations publish set data off-chain. 6EQUJ5 inscribes the approved-label list on Bitcoin in compressed batches, and each root update that references a batch is itself a brc2.0 call. Anyone can rebuild every historical set from Bitcoin data.
- *No trap.* Ragequit (Section V-D) guarantees that a deposit the ASP declines can always leave, publicly, to its origin. The ASP can withhold approval but can never take or freeze funds.
- *No protocol fee on privacy.* Protocol-level deposit and withdrawal fees are zero. The only fee is the relay fee chosen by the user, capped on-chain and paid to independent relayers.

- *Caps while young.* Per-deposit and total caps apply in the first phase. They are raised only as the code survives public review and bounties.
- *Selective disclosure.* After a note is spent, its owner can reveal the opening (v, ℓ, n, s) to a chosen party, proving which deposit funded a given withdrawal. This is safe because the nullifier is already recorded. Secrets of unspent notes are never shared, and version 2 adds read-only viewing keys (Section V-G).

The system is not designed or offered for concealing the proceeds of crime. Who performs the ASP role is a governance decision that will be made with professional advice before any mainnet deployment.

IX. SECURITY ANALYSIS

A. Supply Safety

In protocols where assets are born shielded, a soundness failure (a broken circuit or a compromised setup) can create value that no public counter would reveal. 6EQUJ5 avoids this structurally. Pooled assets are ordinary BRC-20 balances held by the pool in the module's per-ticker contract. The ticker's maximum supply is fixed by its deploy inscription, and the pool cannot mint. A forged proof could at worst withdraw more than its owner deposited. That loss is capped at the pool's balance and is visible, because the pool's balance and its deposit and withdrawal totals are public. Total supply is never affected, and deposit caps bound the worst case further.

B. Threats and Mitigations

Table I summarizes the threats considered and how the design answers each.

TABLE I
THREATS AND MITIGATIONS

Threat	Mitigation
Double withdrawal (reuse a note)	Nullifier hash recorded on first spend; repeat spends revert
Front-running or redirection of a relayed call	Recipient and fee bound in <i>ctx</i> by (3); a copier can only deliver as specified
Proof malleability (re-randomized proof)	Same nullifier, so the copy is rejected
Value forgery (inflated note value)	Contract computes <i>C</i> from the value received; 128-bit range checks
Malicious relayer (theft, censorship)	Cannot steal by (3); users switch relayers or relay themselves
Malicious ASP (censorship, narrow sets)	Ragequit; sets public on Bitcoin; root updates visible
Setup compromise (forged proofs)	1-of- <i>N</i> honest ceremony with Bitcoin beacon; loss bounded (Section IX-A)
Circuit or contract bug	Audited circuits unchanged; minimal diff; negative tests; static analysis [26]; bounty; caps
Indexer divergence (version mismatch)	Height-gated forks; clients cross-check two indexers; published block event hashes [6]
Reorganization beyond engine history	Six-confirmation finality; 64-root history tolerates normal churn
Upgrade-key abuse	Timelocked multisig; published delays; renounce upgrades after burn-in

C. Governance Keys

The upstream Entrypoint is upgradeable. Its owner role can upgrade the implementation and wind down pools, and a separate role posts association-set roots [9]. An upgrade key over a pool is effectively a key over its funds. 6EQUJ5 therefore places the owner role in a multisig behind a public timelock, announces every action before it executes, and plans to renounce upgradeability after a burn-in period. The root-posting role cannot move funds. At worst it can delay approvals, and ragequit still works.

D. Assurance on a Limited Budget

A full audit of a new zero-knowledge system is expensive, so we order the assurance work to cover the riskiest parts first:

1. reuse the audited circuits unchanged, so the existing audit [20] still applies;
2. keep the contract diff small and list every changed line;
3. run static analysis and the known zero-knowledge bug classes against anything touched [26];
4. ship negative tests that must fail: inflation, a reused nullifier, a stale root, an out-of-range value;
5. run a public bug bounty on Signet with falsifiable targets;
6. commission an independent review before deposit caps are raised.

X. PRIVACY ANALYSIS

6EQUJ5 hides one thing: *which deposit funds which withdrawal*. Table II states what remains public. Publishing the limits matters as much as the guarantee, because overstated privacy claims put users at risk.

TABLE II
LEAKAGE OF VERSION 0.1

Public	Hidden
Depositor's script, amount and time at Enter and Shield	Which deposit funded a withdrawal
Each deposit's label and its approval status	Remaining balance of a partly withdrawn note
Withdrawal amount, recipient, relayer, fee and time	Whether two withdrawals share a deposit
Exit by the recipient; the pool's total balance	Notes, secrets and keys

Anonymity set: For a withdrawal of w at time t , the candidates are the approved deposits made before t that still hold at least w . Early on this set is small, and privacy grows with use.

Deanonymization vectors: The main vectors, each with its mitigation, are:

- amount correlation: withdraw standard amounts;
- timing: let other deposits arrive first;
- exit-fee funding (Section VI-D);
- address reuse: use a fresh recipient each time;
- network observation: use Tor [21];
- narrow association sets: an ASP should approve broadly, because a set with one member hides nothing.

Privacy inside the pool rests on the zero-knowledge property of Groth16 and on Poseidon commitments being hiding when the secret s is uniformly random. We claim no more than this.

XI. PERFORMANCE AND COST

Computation: A Groth16 verification with the withdrawal circuit's eight public signals costs about $207,000 + 8 \times 7,160 \approx 265,000$ gas [16]. Poseidon hashing in Solidity costs about 13,000–74,000 gas per hash, depending on arity [27]. A withdrawal therefore totals roughly 0.4–1.0 million gas (*estimate*). brc2.0's per-byte rule allots a 1 KB withdrawal inscription about 12 million gas, so execution fits with a wide margin.

Block space: Bitcoin block space is the binding constraint. Each call needs a commit, a reveal and a send to the OP_RETURN. Table III gives estimates at the October 2026 fee levels of 1–4 sat/vB.

TABLE III
COST ESTIMATES AT 1–4 SAT/vB

Operation	Size (vB)	Cost (sats)
Small call (approve, deposit)	400–500	0.4k–2k
Proof call (withdraw, ragequit)	700–900	0.7k–3.6k
Private round trip (5 inscriptions)	3,500–5,000	4k–20k
Full contract deployment	10,000–20,000	10k–80k

These figures are estimates, not measurements, and they scale linearly with fee rates. A round trip costs a few dollars at current fees, so the pool suits meaningful holdings, and standard deposit sizes help anonymity too. Each step waits for confirmation, so the pool is for holding privately, not for instant payments.

Proving: Proofs are generated in the browser with snarkjs. Comparable Poseidon and Merkle circuits prove in about one second on current flagship phones [28], and the withdrawal circuit is of similar size (*estimate*).

XII. RELATED WORK

Zerocoin [29], Zerocash [23] and Zcash [24] introduced notes, nullifiers and shielded transfers on dedicated chains, with privacy enforced by consensus. Privacy Pools [8], [9] brought association sets to Ethereum, where consensus verifies the proofs. 6EQUJ5 ports that design to a setting without consensus support, where verification falls to the indexers that already define BRC-20 state.

Shielded Bitcoin [4] specifies private BTC transfers on Bitcoin's base layer, with compact envelopes, position-bound nullifiers, and state rebuilt by deterministic replay. It hides amounts and parties, which 6EQUJ5 version 0.1 does not, but it requires its own replay network and wallet. Shielded CSV [30] moves validation entirely to clients. Shielded Ordinals [3] runs Groth16-verified, NFT-shaped notes with its own indexer.

On Bitcoin itself, CoinJoin [31] breaks input-output links probabilistically, and Silent Payments [32] prevent address reuse for receivers. Neither applies to BRC-20 balances, which

indexers attribute by inscription and script. Table IV compares these systems.

TABLE IV
COMPARISON WITH RELATED SYSTEMS, OCTOBER 2026

System (asset)	Proofs verified by	Hidden	Status
Zcash (ZEC)	Zcash consensus	Amounts, parties, links	Live
Privacy Pools (ETH, ERC-20)	Ethereum consensus	Deposit–withdrawal link	Live
Shielded Bitcoin (BTC, via peg)	Protocol replay network	Amounts, parties, links	Paper
Shielded CSV (BTC-like)	Client-side validation	Transaction graph	Research
Shielded Ordinals (NFT notes)	Project indexer, replays	Owner links	Live
CoinJoin, Silent Payments (BTC)	Bitcoin	Input links, address reuse	Live
6EQUJ5 (BRC-20, this work)	Every brc2.0 indexer	Deposit–withdrawal link	Design

XIII. ROADMAP AND STATUS

Each phase has a gate that must be met before the next begins (Table V). We publish gates rather than dates. At the time of writing, phase 0 is complete and nothing is deployed on mainnet.

TABLE V
ROADMAP

Phase	Deliverable	Gate to proceed
0. Specification	This paper	Published for review
1. Verifier on Signet	Groth16 verifier as a brc2.0 contract; gas and vbytes measured	Valid proofs accepted, invalid ones rejected, by independent indexers
2. Pool on Signet	Contract port, open-source relayer, note keeper, public bounty	Round trip measured; no open critical reports
3. Ceremony	Sealed rules, contributions, Bitcoin beacon, inscribed hashes	Transcript verifies end to end
4. Mainnet, capped	Pool with deposit caps and timelocked governance	First withdrawal via a third-party relayer
5. Assurance	Independent review of the diff; caps raised gradually	Review published
6. Research	Shielded transfers, viewing keys, fee sponsorship	Version 2 of this paper

XIV. LIMITATIONS

- *Public edges:* Deposits and exits are public. Version 0.1 hides links, not amounts.
- *A small crowd at first:* Privacy grows with use, so early users get less.
- *Dependence on brc2.0:* The module's maintainers set fork heights and releases. Indexers that do not execute brc2.0 do not see pool state, and it is not yet confirmed that every major wallet and marketplace indexer executes brc2.0 calls itself.
- *Trusted setup:* Groth16 requires one. Transparent proof systems are currently too large for Bitcoin block space.

- *Cost and latency*: Both follow Bitcoin fees and block times.
- *Unproven on brc2.0*: No zero-knowledge verifier has run on brc2.0 before; phase 1 exists to test this.
- *Regulation*: The legal treatment of privacy software varies by jurisdiction and is changing.

XV. CONCLUSION

BRC-20 made Bitcoin tokens possible by letting indexers compute state from public inscriptions. brc2.0 made those indexers programmable, and its pairing precompiles make them able to check zero-knowledge proofs. 6EQUJ5 builds on that. It is a shielded pool whose rules are enforced by the same software that already defines every BRC-20 balance. It uses audited circuits, states its limits openly, and has a failure mode that can never inflate supply. It is a first step toward private tokens on Bitcoin, with no new chain, no bridge, no custodian and no change to Bitcoin.

DISCLAIMER

This document is a technical design for research and discussion. It is not financial, investment, legal or tax advice, and it is not an offer or solicitation to buy or sell any asset. The software described is experimental, unaudited in its adapted form, and not deployed on mainnet. It may never be deployed, and it may contain errors. Third-party systems are described as documented in October 2026. Privacy software must be used lawfully. Names of third-party projects are used for identification only and imply no affiliation or endorsement.

REFERENCES

- [1] C. Rodarmor, "Ordinal theory handbook." [Online]. Available: <https://docs.ordinals.com>
- [2] domo and Layer 1 Foundation, "BRC-20 documentation." [Online]. Available: <https://layer1.gitbook.io/layer1-foundation/protocols/brc-20>
- [3] "Shielded Ordinals (SHORD) protocol v1 and status notes," MIT License. [Online]. Available: <https://github.com/F11ntstowned/shielded-ordinals>
- [4] C. Shikhelman, M. Komarov, and A. Moskvina, "Shielded Bitcoin: Private transfers on the Bitcoin L1," [alloc] init, Sep. 2026. [Online]. Available: <https://www.allocinit.xyz/uploads/shielded-bitcoin.pdf>
- [5] Best in Slot, "brc20-programmable-module (brc2.0): README and source code." [Online]. Available: <https://github.com/bestinslot-xyz/brc20-programmable-module>
- [6] Best in Slot, "Open Protocol Indexer (OPI)." [Online]. Available: <https://github.com/bestinslot-xyz/OPI>
- [7] C. Reitwiessner and V. Buterin, "EIP-196 and EIP-197: Precompiled contracts for alt_bn128," and A. Salazar Cardozo and Z. Williamson, "EIP-1108: Reduce alt_bn128 precompile gas costs." [Online]. Available: <https://eips.ethereum.org>
- [8] V. Buterin, J. Illium, M. Nadler, F. Schär, and A. Soleimani, "Blockchain privacy and regulatory compliance: Towards a practical equilibrium," *Blockchain: Research and Applications*, 2024.
- [9] 0xbow, "privacy-pools-core" (Apache-2.0) and "Privacy Pools documentation." [Online]. Available: <https://github.com/0xbow-io/privacy-pools-core>
- [10] J. R. Ehman, "The Big Ear Wow! signal: What we know and don't know about it after 20 years," Big Ear Radio Observatory, 1997. [Online]. Available: <http://www.bigear.org>
- [11] Best in Slot, "BRC-20 proposal 001: 6-byte namespace." [Online]. Available: <https://github.com/bestinslot-xyz/brc20-proposals>
- [12] Best in Slot, "BRC2.0 programmable module proposal," Layer 1 Foundation Forum. [Online]. Available: <https://1f.discourse.group/t/brc2-0-programmable-module-proposal/766>
- [13] Best in Slot, "brc20-docs: Architecture and protocol notes." [Online]. Available: <https://github.com/bestinslot-xyz/brc20-docs>
- [14] K. Alm, "BIP-322: Generic signed message format." [Online]. Available: <https://github.com/bitcoin/bips>
- [15] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proc. EUROCRYPT*, LNCS vol. 9666, 2016, pp. 305–326.
- [16] Orbiter Research, "Groth16 verification gas costs." [Online]. Available: https://hackmd.io/@Orbiter-Research/S1nat__m0
- [17] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-SNARK parameters in the random beacon model," IACR Cryptology ePrint Archive, Rep. 2017/1050, 2017.
- [18] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *Proc. 30th USENIX Security Symp.*, 2021.
- [19] iden3, "circom" and "snarkjs." [Online]. Available: <https://github.com/iden3>
- [20] Oxorio and Auditware, "Privacy Pools security audits: contracts, circuits, and Entrypoint upgrade," Mar. 2025. Included in [9].
- [21] G. Fanti, S. B. Venkatakrishnan, S. Bakshi, B. Denby, S. Bhargava, A. Miller, and P. Viswanath, "Dandelion++: Lightweight cryptocurrency networking with formal anonymity guarantees," in *Proc. ACM SIGMETRICS*, 2018.
- [22] Privacy and Scaling Explorations, "LeanIMT (zk-kit)." [Online]. Available: <https://github.com/privacy-scaling-explorations/zk-kit>
- [23] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from Bitcoin," in *Proc. IEEE Symp. Security and Privacy*, 2014, pp. 459–474.
- [24] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification." [Online]. Available: <https://zips.z.cash/protocol/protocol.pdf>
- [25] Privacy and Scaling Explorations, "Perpetual Powers of Tau" and "Retrospective: Trusted setups and the p0tion project." [Online]. Available: <https://pse.dev>
- [26] Trail of Bits, "circomspect," and 0xPARC, "zk-bug-tracker." [Online]. Available: <https://github.com/trailofbits/circomspect>
- [27] C. Hudson, "poseidon-solidity." [Online]. Available: <https://github.com/chancehudson/poseidon-solidity>
- [28] moppro, "Client-side proving performance benchmarks." [Online]. Available: <https://zk moppro.org/docs/performance>
- [29] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from Bitcoin," in *Proc. IEEE Symp. Security and Privacy*, 2013, pp. 397–411.
- [30] J. Nick, L. Eagen, and R. Linus, "Shielded CSV: Private and efficient client-side validation," IACR Cryptology ePrint Archive, Rep. 2025/068, 2025.
- [31] G. Maxwell, "CoinJoin: Bitcoin privacy for the real world," Bitcoin Forum, Aug. 2013.
- [32] josibake and R. Somsen, "BIP-352: Silent payments." [Online]. Available: <https://github.com/bitcoin/bips>