# zkBitcoin: Zero-Knowledge Applications for Bitcoin

David Wong
zkSecurity, Σ0

Ivan Mikushin
Σ0, zkSecurity, VMware

January 2024

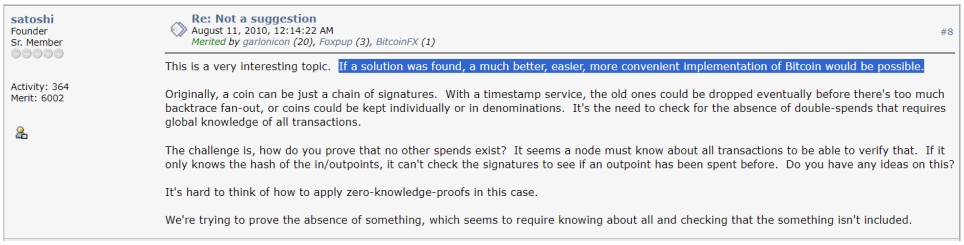
**Abstract**

We introduce a light multi-party computation protocol to verify zero-knowledge circuits on Bitcoin. This enables two use-cases: stateless zkapps (zero-knowledge applications) that lock funds on Bitcoin until users unlock them using zero-knowledge proofs, and stateful zkapps that allow users to update, deposit, and withdraw from a zkapp using zero-knowledge proofs. Since zero-knowledge proofs can't be verified directly on Bitcoin (for lack of optimized opcodes) we use a multi-party committee to verify them off-chain and compute a single on-chain signature. The protocol is akin to a minimal layer 2 on top of Bitcoin that uses Bitcoin as a data-availability layer. Specifically, the committee in charge of verifying zero-knowledge proofs does not have to be connected to the chain as hashes of circuits (verifier keys) are stored in UTXOs on-chain, and the latest state of a (stateful) application is also stored and kept on-chain.

**Keywords:** Bitcoin, UTXO, Zero-Knowledge Proofs, Multi-Party Computation, MPC, L2, FROST, Plonk, Circom, Snarkjs

# 1 Introduction

It is documented that the first time Satoshi (the inventor of Bitcoin) mentioned zero-knowledge proofs, he found the technology interesting but did not know how to apply it.



satoshi
Founder
Sr. Member

Activity: 364
Merit: 6002

Re: Not a suggestion
August 11, 2010, 12:14:22 AM
Merited by garlonicon (20), Foxpup (3), BitcoinFX (1)
#8

This is a very interesting topic. If a solution was found, a much better, easier, more convenient implementation of Bitcoin would be possible.

Originally, a coin can be just a chain of signatures. With a timestamp service, the old ones could be dropped eventually before there's too much backtrace fan-out, or coins could be kept individually or in denominations. It's the need to check for the absence of double-spends that requires global knowledge of all transactions.

The challenge is, how do you prove that no other spends exist? It seems a node must know about all transactions to be able to verify that. If it only knows the hash of the in/outpoints, it can't check the signatures to see if an outpoint has been spent before. Do you have any ideas on this?

It's hard to think of how to apply zero-knowledge-proofs in this case.

We're trying to prove the absence of something, which seems to require knowing about all and checking that the something isn't included.

Figure 1: Satoshi on zero-knowledge proofs

3 years later, the Zerocoin paper from Miers et al. was published at the 2013 IEEE Symposium on Security and Privacy, introducing a way to implement a cryptocurrency using zero-knowledge proofs. The paper was novel in that it successfully managed to hide the sender, recipient, and amount being transferred in each transaction. 3 more years and Zcash, a new cryptocurrency implementing the ideas from the Zerocoin paper and forking the codebase of Bitcoin was launched.

Today, Zcash still has no programmability (so-called smart contracts), and Bitcoin still has no support for zero-knowledge proofs. As cryptocurrencies are slow to make progress, new ones had to take advantage of all the recent advances in the field of zero-knowledge proofs. Systems like Mina and Aleo were proposed as cryptocurrencies that provide programmable constructs augmented with zero-knowledge proofs (allowing applications to have privacy), other projects like Aztec Network built on top of existing cryptocurrencies like Ethereum (as so-called "Layer 2s") to provide similar privacy-enhanced smart contracts.

While Zcash and the other technologies mentioned were a big departure in design from Bitcoin, is it sensible that today Bitcoin could still benefit from zero-knowledge proofs without considerably changing its design. Bitcoin's scripting language is too limited to verify zero-knowledge proofs, but one could consider adding new opcodes to provide such functionality (we discuss this further in the next section on related work). Unfortunately, this would require a hard-fork, and if history tells us anything, such hard-forks often end up splitting the community in two different versions of Bitcoin (the one that accepts the change, and the one that refuses the change).

In this paper we introduce a different path: an extremely light service sit-

ting on top of Bitcoin (so-called layer 2), with no knowledge of the actual canonical blockchain of Bitcoin. The service splits the ownership of a Bitcoin wallet between a multi-party committee. The committee is then in charge of unlocking or updating a zero-knowledge application by signing user-provided transactions carrying valid zero-knowledge proofs.

We released the code behind this proposal on GitHub at github.com/sigma0-xyz/zkbitcoin, and we are currently running a version of this project on the Bitcoin testnet. One can deploy and use zkapps on the Bitcoin testnet simply by using the command-line interface we published in the same repository. For example, a user can deploy a stateless zkapp using the following command:

```
$ zkbtc deploy-transaction --circom-circuit-path
↪  examples/circuit/stateless.circom --satoshi-amount 1000
```

And another user can redeem the funds by providing the correct inputs to the circuit in the following command:

```
$ zkbtc unlock-funds-request --txid "e793..."
↪  --circom-circuit-path examples/circuit/stateless.circom
↪  --proof-inputs '{"preimage":["1"]}' --recipient-address
↪  "tb1q..."
```

The rest of this paper goes like this: section 2 talks about related work, section 3 gives an overview of the protocol, section 4 gives a more detailed specification of the protocol, section 5 covers some security considerations, and section 6 concludes with future work.

## 2 Related Work

In this section we survey zero-knowledge projects related to Bitcoin.
ZeroSync is a project that attempts to provide a verifiable light client for Bitcoin. That is, its goal is to create a zero-knowledge proof that verifies the integrity of the entirety of the canonical chain of Bitcoin at some point in time (also called "state proofs"). As such, it does not offer additional functionalities on top of Bitcoin, but can help external applications to make use of Bitcoin.

BitVM is a proposal that does not use zero-knowledge proofs but is worth mentioning as it augments Bitcoin programmability with fraud proofs. That is, it allows users to lock funds using binary circuits, and unlock funds based

on the execution of such circuits. If a user provides an incorrect execution, another user can then provide a fraud proof. The upside is that, like our proposal, it broadens the boundaries of what's possible in terms of smart contracts on Bitcoin. Unlike our proposal it does not provide any privacy (not even partially, as zero-knowledge proofs allow for private inputs). The downsides are also that it is quite an inefficient protocol as fraud proofs look unrealistic in practice, spanning over a number of transactions linear in size of the circuit.

Alpen Labs is a Layer 2 that settles on Bitcoin using a zero-knowledge proof of its state transitions (so-called "zk rollups"). Its proposition is the most straight-forward one: introducing a new op code (`OP_VERIFYSTARKPROOF`) to verify zero-knowledge proofs on Bitcoin. As we discussed in the introduction, this solution is the most elegant one, but it requires a hard-fork of Bitcoin.

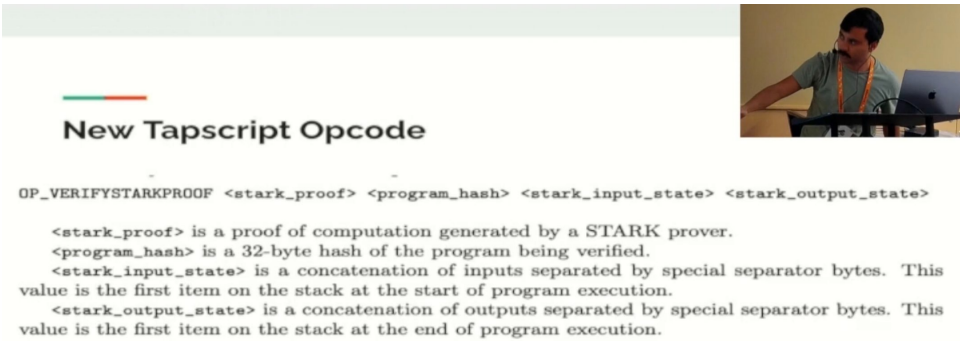



Figure 2: The `OP_VERIFYSTARKPROOF` opcode

Chainway is another "zk rollup" which settles on Bitcoin. There is currently little information about the project so it is not clear who verifies proofs and how funds go in and out of the layer 2.

zeekoe is a L2 channel protocol, akin to the Bitcoin lightning network, which augments the privacy of the participants using zero-knowledge proofs.

Zendoo is a protocol introduced by IOHK to build ZK rollups on "Bitcoin-like blockchain systems". As far as we can tell they do not actually explain how proofs could be verified on Bitcoin.

## 3 Overview of zkBitcoin

The zkBitcoin protocol works with the assumption that there exists a committee of participants that are willing to verify zero-knowledge proofs for users. This committee controls a Bitcoin address, that we'll call `0xzkBitcoin`,

using a threshold signature scheme. This way, no member of the committee knows the private key of the wallet, and UTXOs can only be spent with the agreement of a threshold of committee members.

We support two types of zero-knowledge applications (zkapps): *stateless* and *stateful* zkapps. Let's first explain how stateless zkapps work.

Stateless zkapps are simply an augmentation of the Bitcoin scripting language, that allows a user to lock funds using a zero-knowledge circuit instead of a Bitcoin script. A user can *deploy* such a stateless zkapp by sending a transaction to the Bitcoin network containing a UTXO spendable by `0xzkBitcoin`. In addition, the transaction must also have an unspendable UTXO containing the hash of the verifier key associated to the zero-knowledge proof circuit.

The transaction ID of this transaction represents the stateless zkapp. To unlock funds from this zkapp/UTXO, another user can provide the committee with a transaction that spends the UTXO. If the request is accompanied with a valid zero-knowledge proof that the user could execute the circuit authenticated by the unspendable UTXO, the committee will perform a multi-party computation to sign the transaction and return it to the user. The user can then broadcast the transaction to the Bitcoin network and unlock the funds.

It is important to note that the committee members did not need access to the Bitcoin blockchain to perform their duty: they simply had to verify that the UTXO being spent was part of a transaction matching the previous description of a deployment transaction.

Stateful zkapps are similar, except that the second UTXO containing a hash of the verifier key also contains the state of the zkapp. For the deploy transaction, this means that the second UTXO must also contain the initial state.

Users who want to use a stateful zkapp must create a transaction that spends the zkapp and produce a new zkapp as a new UTXO to `0xzkBitcoin` and a UTXO of the hashed verifier key and new state. The funds locked in the updated zkapp must match the following formula:

$$b_{\text{new}} = b_{\text{old}} + b_{\text{amount-in}} - b_{\text{amount-out}}$$

In other words, the new balance is the old balance plus anything that was deposited and minus anything that was withdrawn.

For the previously discussed mechanisms to work, a stateful zkapp must be linked to a zero-knowledge circuit that takes 4 public inputs in this order:

the new state, the previous state, the amount out and the amount in.

In addition, to ensure that a proof is strongly tied to a specific transaction, one additional public input is added to both stateless and stateful zero-knowledge circuits: the transaction ID spending the zkapp.

In the next section we give a more detailed specification of the protocol.

# 4 The zkBitcoin Protocol

In this section we fully review how the protocol works from the point of view of the users, and from the point of view of the service and the committee members. In addition, we also explain how to modify the FROST threshold signature scheme in order to support Bitcoin Taproot's Schnorr signatures. To recap, zkBitcoin offers two functionalities:

- **Stateless zkapps**: lock some funds and whoever can create a proof can spend *all* the funds.
- **Stateful zkapps**: initialize some authenticated state on-chain and update it by providing a proof.

In the next subsections, we will see how both systems work.

## 4.1 Stateless zkapps

A stateless zkapp can be deployed by anyone (e.g. Alice) with a transaction to `0xzkBitcoin` that contains only one data field:

1. The digest of a verifier key.

In more detail, the transaction should look like this:

```
Transaction {
   version: Version::TWO,
   lock_time: absolute::LockTime::ZERO,
   input: vec![/* alice's funding */],
   output: vec![
      // one of the outputs is the stateless zkapp
      TxOut {
         value: /* amount locked */,
         script_pubkey: /* p2tr script to zkBitcoin pubkey */,
      },
      // the first OP_RETURN output is the vk hash
      TxOut {
         value: /* dust value */,
         script_pubkey: /* OP_RETURN of VK hash */,
      },
      // any other outputs...
   ],
}
```

In order to spend such a transaction, someone (e.g. Bob) needs to produce:

1. The verifier key that hashes to that digest.
2. An unsigned transaction that consumes a stateless zkapp (as input), and produces a fee to the zkBitcoin fund (as output). All other inputs and outputs are free.
3. A proof that verifies with a single public input: a truncated transaction ID (so that the proof authenticates that specific transaction).

To reiterate, the public input is structured as follows:

```
PI = [truncated_txid]
```

When observing such a *valid* request, the MPC committee will sign the zkapp input and return it to Bob.

In more detail, the following transaction is produced by Bob and sent to the MPC committee:

```
Transaction {
   version: Version::TWO,
   lock_time: absolute::LockTime::ZERO,
   input: vec![
      // one of the inputs contains the stateless zkapp
      TxIn {
         previous_output: OutPoint {
               txid: /* the zkapp txid */,
               vout: /* the output id of the zkapp */,
         },
         script_sig: /* p2tr script to zkBitcoin */,
         sequence: Sequence::MAX,
         witness: Witness::new(),
      }
      // any other inputs...
   ],
   output: vec![
      // one of the outputs contains a fee to zkBitcoinFund
      TxOut {
         value: /* ZKBITCOIN_FEE */,
         script_pubkey: /* locked for zkBitcoinFund */,
      }
      // any other outputs...
   ],
}
```

### 4.2 Stateful zkapps

A stateful zkapp can be deployed with a transaction to `0xzkBitcoin` that contains the following data:

1. The digest of a verifier key.
2. 1 field element that represent the initial state of the zkapp. (If there's none the zkapp is treated as a stateless zkapp.)

Note: we are limited to 1 field element as Bitcoin nodes don't forward transactions with more than one `OP_RETURN` output. An `OP_RETURN` seems to be limited to pushing 80 bytes of data, as such we are quite limited here.

In more detail, the transaction should look like this:

```
Transaction {
   version: Version::TWO,
   lock_time: absolute::LockTime::ZERO,
   input: vec![/* alice's funding */],
   output: vec![
      // one of the outputs contain the stateful zkapp
      TxOut {
         value: /* amount locked */,
         script_pubkey: /* p2tr script to zkBitcoin */,
      },
      // an OP_RETURN output containing the vk hash
      ↪  concatenated with the state
      TxOut {
         value: /* dust value */,
         script_pubkey: /* OP_RETURN of VK hash and new state
         ↪  */,
      },
      // arbitrary spendable outputs are also allowed...
   ],
}
```

In order to spend such a transaction Bob needs to produce:

1. The verifier key that hashes to the digest of the verifier key.
2. An unsigned transaction that consumes a stateful zkapp (as input), and produces a fee to the zkBitcoin fund as well as a new stateful zkapp (as outputs). All other inputs and outputs are free.
3. A number of public inputs in this order:
    1. The new state as 1 field element.
    2. The previous state as 1 field element.
    3. A truncated SHA-256 hash of the transaction id (authenticating the transaction).
    4. An amount `amount_out` to withdraw.
    5. An amount `amount_in`to deposit.
4. A proof that verifies for the verifier key and the previous public inputs.

To reiterate, the public input is structured as follows:

```
PI = [new_state | prev_state | truncated_txid | amount_out |
↪  amount_in ]
```

Note: we place `new_state` first, because outputs in Circom are placed first (see this tweet).

Because Bob's transaction will contain the new state, Bob needs to run a proof with `truncated_txid=0` first in order to obtain the new state, then run it again with the `txid` obtained. For this reason, **it is important that the output of the circuit is not impacted by the value of** `truncated_tixd`.

When receiving such a *valid* request (e.g. proof verifies), the MPC committee signs the zkapp input of the transaction and returns it to Bob.

In more detail:

```
Transaction {
   version: Version::TWO,
   lock_time: absolute::LockTime::ZERO,
   input: vec![
      // one of the inputs contains the stateful zkapp
      TxIn {
         previous_output: OutPoint {
               txid: /* the zkapp txid */,
               vout: /* the output id of the zkapp */,
         },
         script_sig: ScriptBuf::new(),
         sequence: Sequence::MAX,
         witness: Witness::new(),
      }
      // other inputs are allowed...
   ],
   output: vec![
      // one of the outputs is a fee to the zkBitcoin fund
      TxOut {
         value: /* ZKBITCOIN_FEE */,
         script_pubkey: /* locked for zkBitcoinFund */,
      }
      // one of the outputs contain the new stateful zkapp
      TxOut {
         value: /* the zkapp value updated to reflect
         ↪  amount_out and amount_in */,
         script_pubkey: /* locked for zkBitcoin */,
      },
      // an OP_RETURN output containing the vk hash as well as
      ↪  the new state
      TxOut {
         value: /* dust value */,
         script_pubkey: /* OP_RETURN of VK hash and new state
         ↪  */,
      },
      // arbitrary spendable outputs are also allowed...
   ],
}
```

### 4.3 The Multi-Party Protocol of zkBitcoin

This section specifies the multi-party architecture of zkBitcoin. We first introduce a number of useful characters:

- Alice is the user that wants to lock her funds in a zkapp
- Bob is the user that wants to unlock funds from Alice's smart contract
- MPC members form a committee of $N$ members, of which the threshold $T < N$ needs to be online to unlock the funds by signing a transaction collaboratively
- The orchestrator is an endpoint that Bob can query to unlock the funds, and who literally "orchestrates" the signature process by talking to the MPC members (MPC members don't talk to one another, and Bob does not need to talk to them either).

Now that the characters have been introduced, here is the flow, which starts with Bob (as Alice does not need to use the zkBitcoin service in order to deploy a zkapp).

Bob starts by sending a request to the orchestrator:

```rust
pub struct BobRequest {
    /// The transaction authenticated by the proof, and that
    ↪ Bob wants to sign.
    /// This transaction should contain the zkapp as input,
    ↪ and a fee as output.
    /// It might also contain a new zkapp as output, in case
    ↪ the input zkapp was stateful.
    pub tx: Transaction,

    /// The transaction that deployed the zkapp.
    /// Technically we could just pass a transaction ID, but
    ↪ this would require nodes to fetch the transaction from
    ↪ the blockchain.
    /// Note that for this optimization to work, we need the
    ↪ full transaction,
    /// as we need to deconstruct the txid of the input of
    ↪ `tx`.
    pub zkapp_tx: Transaction,

    /// The index of the input that contains the zkapp being
    ↪ used.
    // TODO: we should be able to infer this!
    pub zkapp_input: usize,

    /// The verifier key authenticated by the deployed
    ↪ transaction.
    pub vk: plonk::VerifierKey,

    /// A proof of execution.
    pub proof: plonk::Proof,

    /// In case of stateful zkapps, the update that can be
    ↪ converted as public inputs.
    pub update: Option<Update>,

    /// List of all the [TxOut] pointed out by the inputs.
    /// (This is needed to sign the transaction.)
    /// We can trust this because if Bob sends us wrong data
    ↪ the signature we create simply won't verify.
    pub prev_outs: Vec<TxOut>,
}
```

where an update is useful only when a stateful zkapp is being used:

```rust
pub struct Update {
    /// The new state after update.
    pub new_state: String,

    /// The state of the zkapp being used.
    pub prev_state: String,

    /// The truncated txid should be rederived by the
    ↪ verifier.
    #[serde(skip)]
    pub truncated_txid: Option<String>,

    /// The amount being withdrawn from the zkapp.
    pub amount_out: String,

    /// The amount being deposited into the zkapp.
    pub amount_in: String,
}
```

The orchestrator validates the request and aborts if the request is not valid (proof does not verify, or txid has been spent, etc.). The orchestrator then hits the `/round_1_signing` endpoint of each MPC member (or a threshold of it) forwarding Bob's request as is.

A member of the MPC committee that receives such a request verifies the request as well, then starts a `LocalSigningTask` with a message set to the transaction to sign.

```rust
pub struct LocalSigningTask {
    /// So we know if we're processing the same request twice.
    pub proof_hash: [u8; 32],
    /// The smart contract that locked the value.
    pub smart_contract: SmartContract,
    /// transaction to sign.
    pub tx: Transaction,
    /// The previous outputs that are being spent by the
    ↪ transaction (needed to sign).
    pub prev_outs: Vec<TxOut>,
    /// The nonces behind these commitments
    pub nonces: round1::SigningNonces,
}
```

Members keep track of such signing tasks in a local map:

```rust
signing_tasks: RwLock<HashMap<Txid, LocalSigningTask>>
```

The commitments created at this point are sent back to the orchestrator:

```rust
pub struct Round1Response {
    pub commitments:
    ↪ frost_secp256k1_tr::round1::SigningCommitments,
}
```

Note that a committee member doesn't necessarily care about seeing different local tasks for the same `txid`. They'll just keep track of the last one. If they see a new request for the same `txid` incoming, they will ignore it if the request's proof matches, or go through the flow again if it's a new proof (keeping track of the last proof they've seen).

They also do not need to keep track of what round they are in. The existence of a `LocalSigningTask` means that there has been a proof that was verified, and that a transaction is being signed. If the `commitments` vector is not empty, then the first round has been completed. (But since the `LocalSigningTask` still exists the second round hasn't been completed, otherwise the member would have pruned it.)

The orchestrator continues until they collect a threshold of `SigningCommitments`, which they can convert into a `SigningPackage`. They will then send the

`SigningCommitments` to all the participants in that signature by hitting their `/round_2_signing` endpoint.

```
pub struct Round2Request {
    /// The txid that we're referring to.
    pub txid: Txid,

    /// Hash of the proof. Useful to make sure that we're
    ↪ signing the request/proof.
    pub proof_hash: [u8; 32],

    /// The FROST data needed by the MPC participants in the
    ↪ second round.
    pub commitments_map:
        BTreeMap<frost_secp256k1_tr::Identifier,
        ↪ frost_secp256k1_tr::round1::SigningCommitments>,

    /// Digest to hash.
    /// While not necessary as nodes will recompute it
    ↪ themselves, it is good to double check that everyone
    ↪ is on the same page.
    pub message: [u8; 32],
}
```

A member that receives such a request can recreate the `SigningPackage`, and perform the second round of the signature protocol, delete their `LocalSigningTask`, and respond to the orchestrator with their signature share.

```
pub struct Round2Response {
    pub signature_share:
    ↪ frost_secp256k1_tr::round2::SignatureShare,
}
```

The orchestrator will collect a threshold of signature shares, and will then send the aggregated signature back to Bob.

It is the responsibility of Bob to add the signature to the right input in their transaction, and broadcast it to the Bitcoin network.

### 4.4 Making FROST Compatible With Bitcoin's Taproot Upgrade

To perform a multi-party computation of a Schnorr signature, the MPC committee is instantiated with the FROST protocol.
To recap, the Schnorr signature scheme standardized in BIP 340 for Bitcoin works as follows:

- the elliptic curve used in Bitcoin is secp256k1 with its generator $G$
- a keypair is $(s, Y)$ such that $Y = [s]G$
- a signature is $(R, z)$ such that $R = [k]G$ and $z = k + s \cdot c$ with $k$ being a random nonce
- verifying a signature means checking that $[z]G - [c]Y - R = \mathcal{O}$

Unfortunately, vanilla FROST is not compatible with Bitcoin's Schnorr standard (BIP 340 and BIP 341) because of two additions in the Bitcoin version of the scheme.

The first change is that elliptic curve points lose information as they are transmitted on chain: the $y$ coordinate of every point is removed and they can only be identified by their x coordinate. As two points exists for the same x coordinate (i.e. if $(x, y)$ exists on the curve then $(x, -y)$ exists on the curve as well) Bitcoin specifies that recovered points always have even $y$ coordinates.

The second change is that Schnorr public keys are tweaked before being used. The tweak allows to hide more information in the public key, and is part of the BIP 341 standard. If $Y$ is the actual Schnorr public key, the tweaked public key $Y'$ is computed as follows:

$$Y' = Y + [t]G$$

With $t$ the tweak (we refer to BIP 341 for its calculation).

In practice, implementations compatible with Bitcoin will "correct" the different values used in the scheme depending on the derived published values. For example, if a signing operation produces an elliptic curve point $R = [k]G$ that has an odd $y$ coordinate, then $k$ is negated. This is because the point that the chain sees is $-R$ as it expects an even $y$ coordinate.

For FROST, this means that private values must be corrected in a number of places. First, signers must ensure that they're using the correct nonces (after $R$ was produced, as explained in the previous examples), and that

they're using the correct signer share. The latter point can be done right after key generation (depending on what the aggregated public key $Y$ looks like without its $y$ coordinate) or on the fly every time during signing.

Second, the aggregator must produce the second part of the signature ($[z]G$) correctly depending on the aggregated public key $Y'$ which could, without its $y$ coordinate, be interpreted with a negative tweak. We summarize all these edge cases in the following figure.

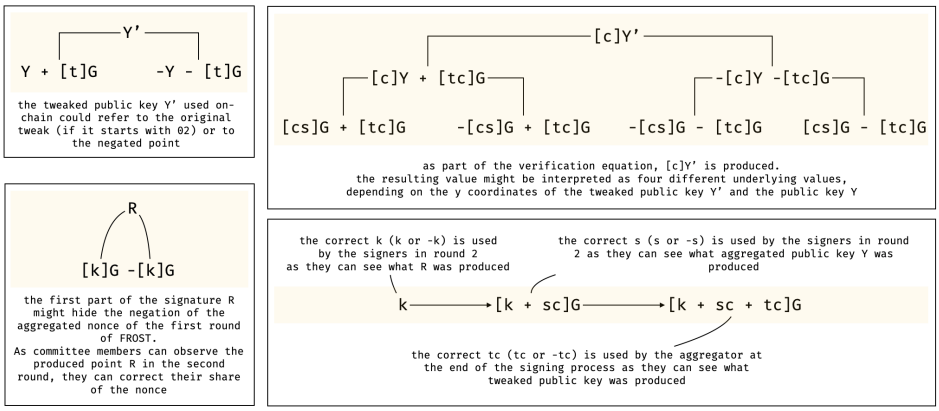



Figure 3: FROST edge-cases due to Bitcoin's Schnorr signature scheme

# 5 Security Considerations

Let's now discuss the security of the zkBitcoin system. Two properties must be maintained: safety and liveness. Both rely mostly on the security of the multi-party committee itself.

Safety is maintained as long as the configured threshold of the committee remains honest. That is, if an adversary corrupts the configured threshold of participants then it can recover the private key behind the zkBitcoin wallet and drain all of its funds. There are multiple metrics that we can tweak here: a configuration metric (number of participants, threshold), a defense-in-depth metric (run the MPC nodes in a trusted execution environment), a protocol metric (ensure that shares are refreshed periodically and that the committee is dynamically incentivized to behave correctly), and a social metric (choose a committee that people can trust).

Liveness is maintained in a similar fashion, but some of the trade-offs are opposed to the safety ones. For example, decreasing the threshold parameter would increase the liveness, but would decrease the safety.

As a Layer 2 protocol, zkBitcoin also does not currently offer a way to exit in case of liveness (or censorship) issues.
In addition, while the code is open-sourced, users should be able to verify that the zkBitcoin public key used by the client is indeed the one that was generated during a distributed key generation. This is possible if committee members publish enough information publicly.

# 6 Conclusion and Future Work

We have presented an extremely light solution to verifying zero-knowledge proofs on Bitcoin without relying on running Bitcoin nodes but solely on a multi-party computation protocol.

At present, it seems hard to imagine a better solution to support zero-knowledge proofs on Bitcoin without extending the Bitcoin scripting language (or the Bitcoin protocol itself).

In addition, multi-user zkapps suffer not only from the ZK update conflict issue, but from updates in general due to the UTXO-model of Bitcoin. Indeed, every update of a zkapp "moves" the updated zkapp state to a new place as its transaction ID changes.

In the initial version of zkBitcoin we support PLONK proofs built using circom and snarkjs with parameters supporting circuits of $2^{16}$ constraints maximum. We ignore proof systems like Groth16 which are heavily used on other networks like Ethereum, as it would mean supporting different parameters for different circuits. Our future plans include supporting larger circuits and more proof systems as there are no tangible blockers to enable that, finding an MPC committee that users can trust, and augmenting the protocol to allow for committee updates (in case of shares or members being compromised, and for new members to be able to join the committee).