ColliderVM: Stateful Computation on Bitcoin without Fraud Proofs

Victor I. Kolobov[1], Avihu M. Levy[1], and Moni Naor*[2]

[1]StarkWare, {victor.k,avihu}@starkware.co
[2]Weizmann Institute of Science, moni.naor@weizmann.ac.il

April 10, 2025

Abstract

Bitcoin script cannot easily access and store state information onchain without an upgrade such as BIP-347 (OP_CAT); this makes performing general (stateful) computation on Bitcoin impossible to do directly. Despite this limitation, several approaches have been proposed to bypass it, with BitVM being the closest to production. BitVM enables fraud-proof-based computation on Bitcoin, relying on a 1-*out-of-n* honesty assumption.

This left the question of whether it is possible to achieve computation under the same honesty assumption *without* requiring onlookers to ensure validity through fraud proofs. In this note, we answer this question affirmatively by introducing ColliderVM, a new approach for performing computation on Bitcoin today. Crucially, this approach eliminates some capital inefficiency concerns stemming from reliance on fraud proofs.

For our construction, a key point is to replace the Lamport or Winternitz signature-based storage component in contemporary protocols with a hash collision-based commitment. Our techniques are inspired by ColliderScript, but are more efficient, reducing the number of hash evaluations required by at least $\times 10000$. With it, we estimate that the Bitcoin script length for STARK proof verification becomes nearly practical, allowing it to be used alongside other, pairing-based proof systems common today in applications.

1 Introduction

Computation on Bitcoin is performed by writing the program logic in Bitcoin script, which is a Forth-like stack-based language. This programming language is very restrictive in its limited opcode expressibility and overall limitation to around 4 million opcodes that can fit in a Bitcoin block *across all transactions*, as well as each script being memory-restricted to a stack size of at most 1000 elements.

However, arguably the biggest limitation of Bitcoin script is in its lack of native support in performing what we term *stateful computation*. Roughly speaking, stateful computation refers to a computation's ability to persist onchain across multiple transactions and user interactions. More concretely, stateful computation consists of two components:

- **Data persistence** - The ability of a program to access and modify variables that persist beyond its own execution time. For example, accessing data that was stored onchain before the execution of the program, or leaving some data onchain for other programs to read.[1]

*Work done while consulting for StarkWare.

[1]While any program execution leaves some data on chain, we need this data to be *accessible from within the programs themselves*.

- **Logic persistence** - This is the ability of the program's logic to persist across multiple transactions, Bitcoin blocks, or user interactions. By default, this is not the case with Bitcoin script, as anyone with the ability to spend a UTXO by providing the locking script with the correct witness is able to lock the funds within new UTXOs with possibly completely unrelated locking scripts.

Stateful computation is an important primitive that enables the following useful ways to perform onchain computation:

- It allows splitting a computation longer than ~ 4 million opcodes, or one that doesn't fit into Bitcoin's 1000 stack element limitation into multiple transactions;[2]
- It allows the user to interact at different points in time with the same computational logic.

This, in turn, enables the implementation of many useful applications, such as vaults [39], smart contracts [36, 17], and L2 bridges [21, 43], with the least amount of security assumptions and protocol complexity.

1.1 Achieving stateful computation on Bitcoin

Below, we list the existing ways of achieving stateful computation on Bitcoin.

Multisig. This approach involves funneling all the funds in the system into the *multi-signature* of a predetermined set of n entities that oversee the computation. This kind of solution has the worst security in the list since it's a *t-out-of-n* system, with t honest entities required for both safety and liveness.[3]

While this approach has the benefit of being the most straightforward and most common [4], due to its reduced security, we consider it our baseline non-solution.

Covenants. This solution is second to Multisig in terms of simplicity, and it enjoys the *best* security in the list, but it requires an *upgrade to Bitcoin* by including an additional opcode (i.e., `OP_CAT` [14], `OP_CTV` [33], or `OP_TXHASH` [30], among others). Covenants [24, 26, 6, 38, 37, 16] allow the locking script to *constrain* the spending transaction, with the amount of constraining possible depending on the opcode.

The utility of covenants, if sufficiently powerful,[4] is twofold: They allow Bitcoin script to have both data and logic persistence without any additional security assumptions. Unfortunately, as of today, achieving proper covenants on Bitcoin seems infeasible, with all solutions needing some way to bypass utilizing covenants directly.

ColliderScript. This approach uses a prohibitive amount of computation to emulate covenants. In a similar vein to true covenants, it doesn't require any trusted entities, and thus it inherits mostly the same security guarantees. Moreover, while the ColliderScript [13] construction is complicated, it has *encapsulated complexity*, allowing it to be used in a black-box manner, similar to covenants.

[2] In fact, there are other limitations, like the notion of *non-standardness [3]*, which we ignore here.

[3] This introduces a tradeoff, as the system can tolerate at most t malicious actors for safety, and at most $n - t$ malicious actors for liveness. Thus, a smaller t is better for liveness, while a larger t is better for safety.

[4] For example, the covenants enabled through `OP_CAT`, which allow Bitcoin script to arbitrarily reason about most of the data of the spending transaction [28, 29].

Functional encryption-based schemes. In [31, 18], a general framework is proposed for emulating covenants by requiring heavy cryptographic tools such as Functional Encryption [7]. To the best of our knowledge, this approach hasn't yet coalesced into a concrete construction whose security can be examined. If possible, such a construction would inherit similar security guarantees as ColliderScript or covenants, with an additional trusted setup phase, due to the use of public key cryptography.

BitVM. Following the line of work on BitVM [22, 19, 23, 5, 20, 32], the most relevant version of this technique to our work is due to [23] and [5], which is known as *BitVM2*. Instead of a single set of n entities as in Multisig, here, there are n signers and m operators. The BitVM construction is secure as long as 1-out-of-n signers are honest, as well as 1-out-of-m operators. Crucially, in addition to the improved security tradeoff versus Multisig, this approach provides a separation of trust, as signers are used for *enforcing honest computation*, while operators are used for actually *performing computation* and taking *custody* of the funds.[5]

Moreover, BitVM2 uses a fraud proof system, allowing *any* onlooker to report and "slash" fraudulent computation execution by the operator. If our computation script f is split into subfunctions $f_1, \ldots, f_k$, the fraud proof system has the benefit of only requiring $\max_i |f_i|$ script size to execute onchain *in the worst case*, as opposed to the entire cost of $\sum_i |f_i|$.[6]

However, fraud proofs also have some downsides, a major one being capital inefficiency. In the context of BitVM, the operator *pays out of pocket per user withdrawal*, and waits for the duration of the fraud proof time window to be reimbursed. This places liquidity requirements on the operator, potentially making exiting the system very difficult. A major goal of this work is to address this deficiency.

2 Our contributions

At a high level, ColliderVM is a protocol that facilitates stateful computation on top of Bitcoin. As an application, our protocol also enables the computation of STARK verifiers, which can be used for the important goal of building trust-minimized L2 bridges.

For security, the guarantees we get are considerably better than a multisig, and our approach does not require fraud proofs. Our protocol involves n signers, and m (bridge) operators, such that as long as 1-out-of-n of the former is honest, the protocol is safe, and as long as 1-out-of-m of the latter is honest, the protocol is guaranteed liveness. In addition, our construction imposes hash collision challenges on potentially malicious operators, so, for safety, we need to assume this type of cryptographic puzzle is computationally difficult.

While our construction has notable attractive benefits, such as avoiding the capital inefficiency problem due to fraud proofs, it also has significant drawbacks. Nevertheless, we believe that our protocol is far from optimal, and this approach is fruitful for enabling new applications on top of Bitcoin.

Below, we give an outline of the ColliderVM protocol.

[5] Signers are expected to delete their keys after performing their role in the protocol, hence, only the operators have the ability to take custody over the funds.

[6] There is an additional cost linear in the input length to *all* subfunctions $\{f_i\}_i$, even if the script size cost is only $\max_i |f_i|$.

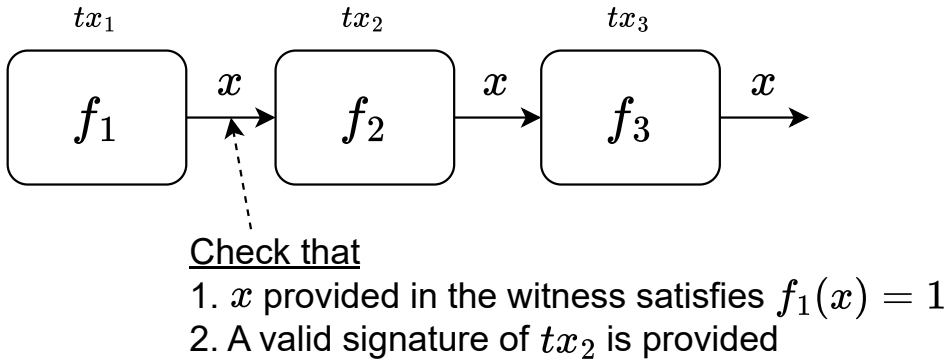

Figure 1: Suppose that $f(x) = f_1(x) \wedge f_2(x) \wedge f_3(x)$. The signer prepares presigned transactions such that each tx_i computes a single f_i and spends the *previous* transaction in the flow, and sends them to the operator in an *offline* (or *setup*) phase. In the *online phase*, to spend an individual tx_i, an operator needs to provide the signer's signature, together with an input x such that $f_i(x) = 1$. Because each transaction checks the signer's signature, the logic flow is fixed, as the signer didn't sign on any other transactions. If the transactions are provided with the same input x then the flow succeeds only if $f(x) = 1$. However, on its own, nothing protects against a malicious operator providing *potentially different* x_i to each tx_i.

2.1 The construction

We'll first describe our construction in a simple 2-party setting. To this end, consider a protocol between an *operator* and a *signer*. The operator knows an input x to some function f such that $f(x) = 1$, which the signer wants to see evaluated on top of Bitcoin, consequently *rewarding* the operator with some amount of Bitcoins for this computation. If the script implementing f fits in a single UTXO, the signer may simply set up a UTXO that evaluates f and checks whether it equates to 1, together with checking a signature associated with the operator's public key.

The difficulties arise when $f\colon \mathcal{X} \to \{0,1\}$ does not fit into a single transaction. Now, the signer instead splits the function f into multiple subfunctions, say f_1, f_2, f_3, such that $f(x) = f_1(x) \wedge f_2(x) \wedge f_3(x)$, where each f_i fits individually into a single transaction.[7] Then, the signer may use 3 *presigned transactions*, tx_1, tx_2, tx_3, where each one evaluates a single f_i, spends the previous transaction, and requires the signature of a signer. The signer sends these transactions to the operator, which, at a later date when x is known, evaluates all f_i on x. This process, illustrated in Figure 1, enforces the execution of f_i in order. Crucially, presigned transactions commit *only* to the function being computed and *not* the function's input. This is because a transaction's witness, which the function input is part of, is not being signed on.[8] We give a more complete description of this process in Section 3.2.

However, using presigned transactions alone doesn't give us security or "soundness". This is because this flow doesn't protect against an adversary capable of finding possibly *distinct* x_1, x_2, x_3,

[7] A more complete description on how to split functions we are interested in into smaller subfunctions is given in Section 4.3.

[8] To describe how presigned transactions enforce logic, suppose that tx_2 *spends* tx_1 in the presigned flow. tx_1 is only able to check that the spending transaction (hopefully tx_2) was signed on by the signers, but it has no way to enforce that it's really tx_2. Logic enforcement happens by tx_2, which specifies in an input the *txid* of tx_1, guaranteeing tx_2's locking script will only execute after tx_1 was already spent. Hence, if the signers only presign tx_1 and tx_2, and delete their private key, this guarantees no other transaction flow is possible.

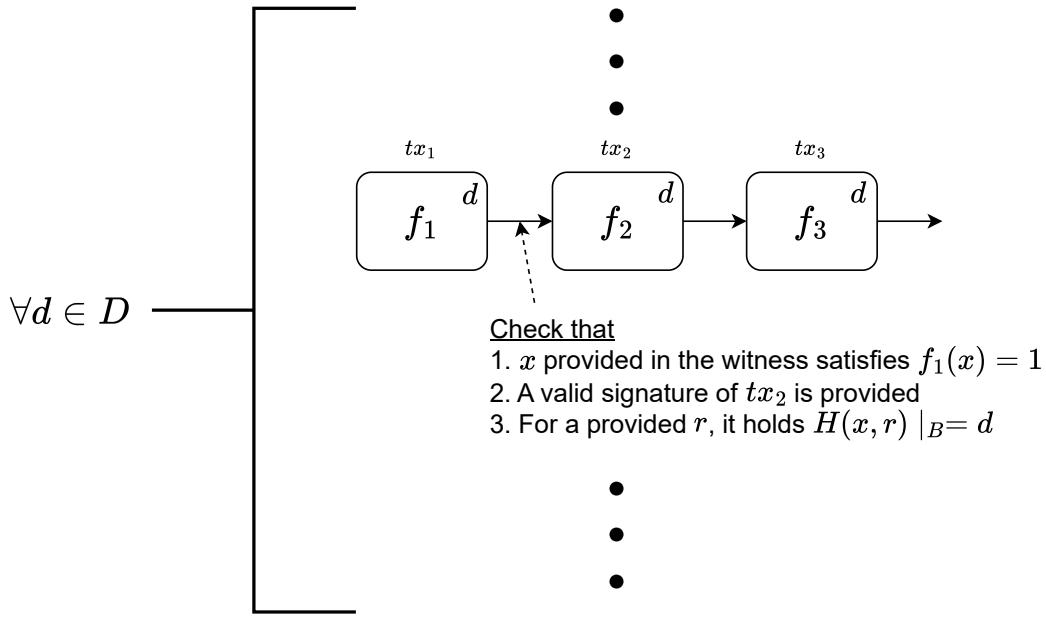

Figure 2: The offline (or setup) phase in the ColliderVM construction. This time, during the offline phase, the signer prepares $|D| = 2^L$ flows, each encoding an individual $d \in D$ across *all* transactions of a single flow. During the online phase, an operator willing to commit on x, needs to find an r such that $H(x,r)|_B = d \in D$. Then, the operator needs to use the specific flow associated with this d. The security relies on the idea that finding two distinct valid B-pairs with respect to the same $d \in D$, namely $(x,r) \neq (x',r')$, is more difficult than finding a single valid B-pair with respect to D.

provided to tx_1, tx_2, tx_3 respectively, such that $f_i(x_i) = 1$ holds *separately*, even though there is no x such that $f(x) = 1$.

To solve this, we make it computationally expensive for the operator to find more than a single x to be used. This way, to cheat the system, an adversary will need to find *two* values $x \neq x'$ that fit the requirement. Now, suppose that we're given a set D of size 2^L.[9] We call an input x with a nonce r a *valid B-pair with respect to D*, if, for some fixed hash function H, it holds that $H(x,r)|_B \in D$, namely, the first B bits are an element in D (therefore $D \subseteq \{0,1\}^B$).

Finding a single x satisfying the above requires trying 2^{B-L} hash evaluations in expectation, since each attempt with a random r_i has probability $1/2^{B-L}$ of hitting a member of D (assuming the hash function behaves as a random one). In contrast, finding for two distinct $x \neq x'$ and r, r' such that (x,r) and (x',r') are both valid B-pairs w.r.t. D *and* $H(x,r) = H(x',r')$ involves finding a collision in D, which requires $\sim 2^{L/2} \cdot 2^{B-L} = 2^{B-L/2}$ hash evaluations in expectation (the $2^{L/2}$ factor comes from the birthday paradox, e.g. Chapter 5 [25]).

Thus, if the presigned transactions tx_i could check that the input (x_i, r_i) given to it (where r_i is some nonce) is a valid B-pair with respect to *the same element in D*, the adversary would need to perform $B - L/2$ bits of work to cheat the system. On the other hand, an honest operator needs only $B - L$ bits of work to find a single valid B-pair (by bits of work we mean log the amount of expected work). To check for the same element in D, the signer needs to prepare and store a copy of the flow $f_1 \to f_2 \to f_3$ *for each element in D*, totaling in 2^L presigned transaction flows, each consisting of 3 transactions. This process, being somewhat informal, we call the *ColliderVM protocol*. Its offline phase we illustrate in Figure 2.

[9] We have freedom in choosing the elements of D, as long as it is easy to check membership in D via a Bitcoin script. For example D could include all integers between 0 and $2^L - 1$.

Generalizing the result to a function f split into k subfunctions $f_1, \ldots, f_k$, we conclude the following.[10]

Proposition 1. *Suppose that a function $f\colon \mathcal{X} \to \{0,1\}$ can be split into k subfunctions $f_1, \ldots, f_k$ such that $f(x) = \bigwedge_{i=1}^{k} f_i(x)$, and suppose that each f_i fits into a single Bitcoin transaction. Then, there is a ColliderVM protocol such that*

- *There are 2^L presigned flows of k presigned transactions in total, each evaluating a different subfunction f_i, for $1 \leq i \leq k$.*
- *An honest operator needs to perform $B - L$ bits of work to evaluate f.*
- *A malicious operator needs to perform $B - L/2$ bits of work to cheat the system.*

A reasonable setting of parameters could be $L = 46$ and $B = 120$. This way, we require 2^{46} presigned flows;[11] an honest operator needs to perform 74 bits of work to evaluate f, while a malicious operator needs 97 bits of work to break the system. Compare these numbers to ColliderScript [13], which didn't have any flexibility in parameter choice, always requiring 2^{56} bytes of storage, around 86 bits of work from an honest operator, and 110 bits of work from an adversary. Furthermore, ColliderScript has a computationally intensive preprocessing phase (83-93 bits of work) which is not required here.

Choosing a security parameter. A natural question to ask is how high should we set the bound on work required by an adversary to break the system. While a standard choice is 128-bit security, we argue that for some applications, such as L2 bridges in Section 3.2, a significantly lower value suffices. The construction in Section 3.2 has $B - L/2$ bits of security *per deposit*, meaning that if deposits are sufficiently small, the adversary is only limited to *griefing attacks* (i.e. those that are not profitable but simply intended to cause damage), as expending intensive computational resources of the same type needed to break the protocol is better utilized towards, say, mining for the Bitcoin network. Indeed, as of today (April 2025), mining a Bitcoin block requires, on average, 78 bits of work, yielding a block reward of 3.125 BTC.

Thus, the security parameter may be set even lower. For instance, for $L = 36$ and $B = 100$, we require only 2^{36} presigned flows; an honest operator needs to perform 64 bits of work to evalute f, while a malicious operator needs 82 bits of work to break the system. We illustrate the computational complexity gap between honest and malicious in Figure 3.

What properties of the computational gap are needed? The scheme we described works in expectation. But is this sufficient? It might be more appropriate if for a given upper bound on the adversary's computational power it will have only a small chance of successfully cheating (i.e. creating a convincing proof). For instance, if there is a 1 in a 1000 chance that the adversary succeeds in finding for the desired x and x' a collision (an r and r' so that $H(x,r)|_B = H(x',r')|_B$), then one in a thousand attempts will yield a faulty proof.

The advantage in our scheme stems from the birthday paradox, so it means that if an adversary invests T time, it has a chance of $T/2^{B-L}$ to succeed.

Another important issue is the amount of memory the attacker needs. In general, to find collisions, the best known algorithm with reasonable time requires memory proportional to the

[10] This way, to implement stateful computation, presigned transactions provide logic persistence, while the hash collision-based commitment provides data persistence.

[11] Since signature messages are hashed with SHA256 [40, 41], revealing this many signatures is still secure.

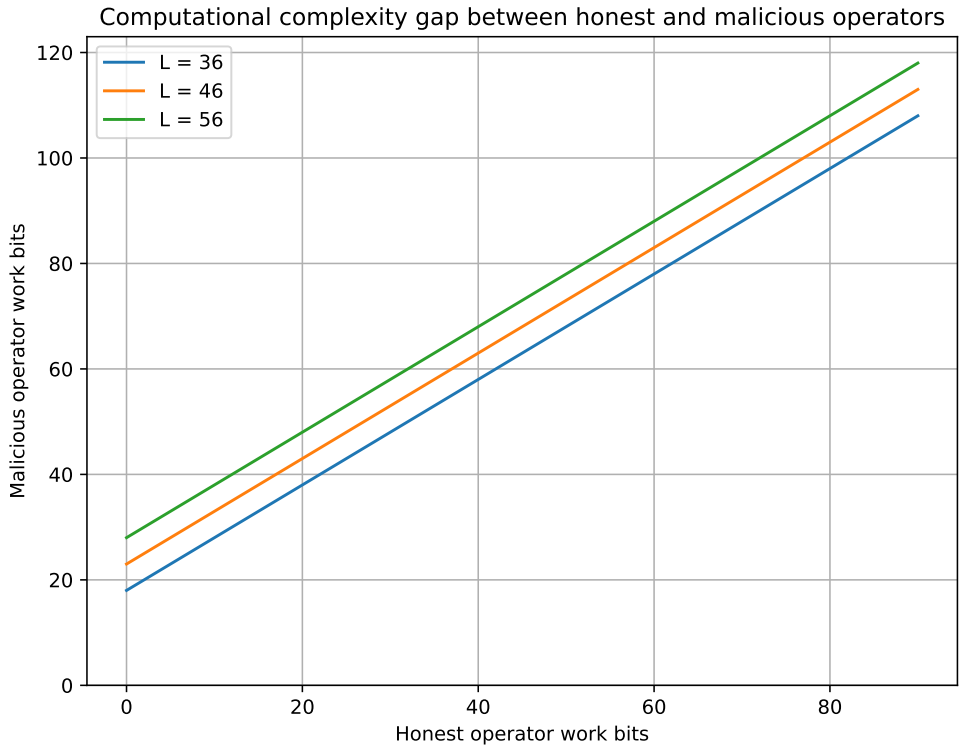

Figure 3: For chosen parameters B and L, and honest operator needs to perform $B - L$ bits of work, while a malicious operator needs to perform $B - L/2$ bits of work. Thus, the computational complexity gap between them is exactly $L/2$ bits of work, where 2^L is the number of presigned flows.

square root of the size of the range on which we are searching for collisions. In our case this is $2^{L/2}$, much smaller than the time which is $2^{B-L/2}$. Therefore, the memory requirement alone would not significantly hinder the attack. It is an interesting direction to find conditions where in order to be successful, the adversary is required to use a lot of memory.

2.2 From double to triple collisions

It is possible to further increase the work an adversary needs to do at the expense of redundancy in Bitcoin script. For this, we construct a scheme that is robust against an adversary capable of finding *two* valid B-pairs $(x, r) \neq (x', r')$, but not *three* of them.

Suppose that any *two* subfunctions (f_i, f_j) can fit into a single transaction and suppose that $f = \bigwedge_{i=1}^{k} f_i$. Then, we construct a ColliderVM protocol where there is a transaction for each pair of subfunctions (f_i, f_j) where $1 \leq i < j \leq k$. Figure 4 provides an example of such a construction for $k = 3$.

Consider a given flow of transactions $tx_1 \rightarrow \ldots \rightarrow tx_q$ for $q = \binom{k}{2}$, corresponding to the computation of $f = f_1 \wedge f_2 \wedge, \cdots f_k$. If only two types of inputs, x and x' were used for all the pairs, then no matter how x and x' are partitioned among the pairs of transactions, either x or x' was computed and approved on all of $\{f_1, f_2, \ldots f_k\}$.

If this is not true, then it cannot be the case that there is a subfunction f_i where for all the pairs of the form (f_i, f_j) the input was always x (and similarly for x'), since this would mean that x satisfies all $f_1, f_2, \ldots f_k$, contradicting the assumption. So breaking such a system necessarily implies finding *three* colliding valid B-pairs. We conclude the following.

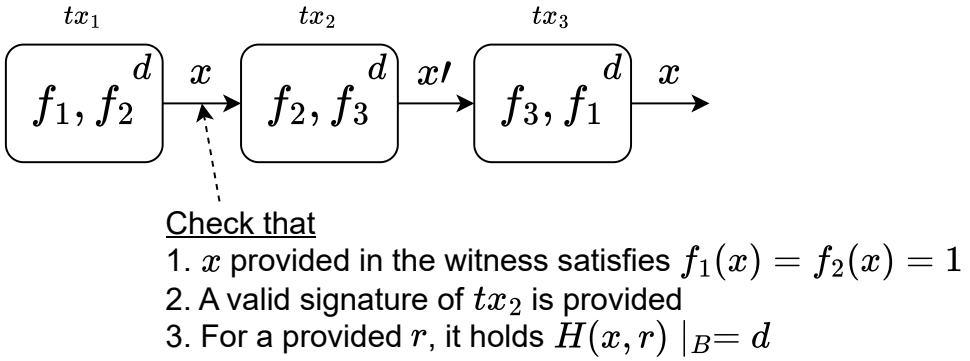

Figure 4: A ColliderVM protocol flow resilient against a double collision. No matter how x and $, x'$, where $x \neq x'$, are distributed among the transactions, at least one of them evaluates all subfunctions f_1, f_2, f_3. The protocol breaks only if a triple collision (x, x', x'') is found.

Proposition 2. *Suppose that a function $f\colon \mathcal{X} \to \{0,1\}$ can be split into k subfunctions $f_1, \ldots, f_k$ such that $f(x) = \bigwedge_{i=1}^{k} f_i(x)$, and suppose that* each pair *of subfunctions (f_i, f_j) fits into a single Bitcoin transaction. Then, the ColliderVM protocol described above satisfies:*

- *There are 2^L presigned flows of $\binom{k}{2}$ presigned transactions in total, each evaluating a different pair of subfunctions (f_i, f_j).*
- *An honest operator needs to perform $B - L$ bits of work to evaluate f.*
- *A malicious operator needs to perform $B - L/3$ bits of work to cheat the system.*

Here, much more lenient parameters can be chosen, allowing a larger gap for less preprocessing work. For example, $L = 39$ and $B = 110$. This way, we require 2^{39} presigned flows; an honest operator needs to perform 71 bits of work to evaluate f, while a malicious operator needs 97 bits of work.

On the flip side, this approach introduces a nontrivial Bitcoin script size overhead. While the computation consists of k subfunctions, the number of transactions we are using is $\binom{k}{2}$, where each transaction involves computing 2 subfunctions. Overall, this gives a multiplicative overhead of $2\binom{k}{2}/k = k - 1$. Despite the significant overhead, we believe this kind of construction is especially useful in cases where the bottleneck is not necessarily the script size but Bitcoin's 1000 element stack size limitation.

A lower bound. One might wonder whether a better splitting of $f = \bigwedge_{i=1}^{k} f_i$ into transactions is possible so that we achieve a better multiplicative overhead. We argue that as long as we're limited to two subfunctions per Bitcoin transaction, this is not possible, at least if we are pessimistic about the behavior of the subfunctions, meaning that it is possible to satisfy $k - 1$ of them and not all of them (for any $k - 1$ subset).

Proposition 3. *A ColliderVM protocol for $f = \bigwedge_{i=1}^{k} f_i$ as in Proposition 2 requires at least $\binom{k}{2}$ presigned transactions.*

Proof. Suppose to the contrary that such a protocol exists. This implies there is a pair of distinct subfunctions (f_i, f_j) that don't appear together in a transaction. If there are x and x' where x

satisfies all subfunctions but f_j and x' satisfies all subfunctions but f_i, then we claim that it is possible to cheat.

Suppose we have an adversary with two valid B-pairs, (x, r) and (x', r'). They can cheat the system as follows: the adversary partitions the presigned transactions into two sets: S_1, which includes all transactions containing f_i, and S_2, the rest (those not containing f_i). Then, by providing (x, r) as input to S_1 and (x', r') as input to S_2, the adversary guarantees that f_j is not evaluated on x and f_i is not evaluated on x', but the execution nevertheless succeeds, which is a contradiction. □

Higher order collisions. Similar constructions are possible for higher order collisions. We don't explore them in this work, since we believe the script size overhead to be too large.

3 Extensions and Applications

First, we note that the aforementioned ColliderVM protocol can be extended to *multiple* operators and signers. This is done as follows

- Instead of a single signer signing on the presigned transactions, all m signers are required to sign them, and the locking scripts of UTXOs check for the signature of all m signers;
- Instead of the signers creating $|D| = 2^L$ flows for a single operator, they do the same *for each* of the n operators.

This way, the ColliderVM protocol has *safety* as long as 1 out of m signers is honest (preventing m signatures on invalid transactions), and *liveness* as long 1 out of n operators is honest (making sure a presigned transaction flow reaches the Bitcoin chain by at least 1 active operator).

Below, we list important applications of the ColliderVM protocol, which applies to the extended m-signer n-operator version as well.

3.1 Onchain computation delegation

A prototypical use case for stateful computation, as enabled by ColliderVM, is for *computation delegation*, which is achieved through the use of a *proof system*. Performing computation directly onchain is expensive, hence, it is beneficial to *delegate* the computation to a prover, and only perform verification of said computation onchain, as enabled by the proof system. In fact, an exponential amount of computation can be verified onchain compared to the onchain footprint, which in practice makes the verifier cost "nearly constant."

Unfortunately, even a cheap verifier program is unlikely to fit into a single transaction on Bitcoin due to the strict Bitcoin script limitations, such as stack size, opcode availability, and locking script size. As a result, the verifier must be split into multiple transactions.

To this end, a promising proof system choice to be deployed on top of Bitcoin is Circle STARK [12], which uses a small field, making it feasibly computable.[12] In fact, an implementation [1] of a verifier for this proof system uses about 3 million opcodes.

Alas, the implementation [1] does not fall into our framework, as it assumes the existence of `OP_CAT` in Bitcoin script, allowing it to use hashes, such as SHA256 via `OP_SHA256`, counting as a

[12]Bitcoin script has reduced expressivity due to the absence of many common opcodes such as string operations, multiplication, and logic operations [35, 34]. This, in turn, necessitates the *emulation* of these common operations with a large number of opcodes. For example, multiplication over a 31-bit field can be done with ~ 1000 opcodes [8, 9]. This is a severe limitation, as due to the limited size of a Bitcoin block, at most ~ 4 million opcodes can fit inside it across all transactions combined.

single opcode each. Nevertheless, as there are only a few thousand invocations of `OP_SHA256` in that implementation, if we replace them with *emulations* of hashes in "Small Script",[13] The total program size only expands moderately. Indeed, by using BLAKE3, which admits a $\sim 45K$ opcode implementation, we estimate in Section 4.3 that the verifier program size becomes $\sim$ 80 million opcodes. Furthermore, by using a *Bitcoin-friendly hash function*, we conjecture the verification cost to be smaller by at least an order of magnitude.

Compared to BitVM2. In contrast, BitVM2-based protocols use a pairing-based SNARK, such as Groth16 [11], with verifier script sizes being in the gigabytes,[14] necessitating a fraud-proof system to only evaluate part of the verifier program onchain. BitVM2 protocols have a 26 bytes-per-input-bit overhead [2] in cross-transaction storage, due to the Winternitz signatures, requiring pairing-based proof systems, which enjoy a shorter proof size at the expense of onchain verifier program size, compared to that of a STARK. In contrast, in ColliderVM, the cost depends on the implementation of $H(x, r)|_B$. As an example, using BLAKE3 [15], which requires 45K opcodes to hash 512 bit messages, yields an overhead of 88 bytes-per-bit. However, we note that

- With a Bitcoin-friendly hash function, this cost can be brought significantly down;
- As mentioned in Section 4.3, for common applications such as a STARK verifier, it is possible to commit to only *parts* of the input, which further reduces the overhead.

3.2 L2 bridges

ColliderVM can also be used to implement bridges to L2 protocols. Below, we give a high-level outline of one possibility of what such a protocol can look like. We begin by presenting the participating parties:

- *Users* willing to deposit and withdraw from the L2 network using the bridge, as well as perform transactions on top of the L2 blockchain;
- *L2 Sequencers* responsible for creating new blocks for the L2 blockchain, as well as generating proofs of said blocks; They are needed for liveness.
- *(Bridge) operators* responsible for handling deposit and withdraw requests from users;
- *Signers* who provide signatures during each user deposit operation to facilitate the security of the protocol. They are not needed at withdrawal.

For the protocol setup, we assume there are n operators and m signers. As in the BitVM2 bridge, the security of our protocol relies on the assumption that at least 1 operator and 1 signer are honest. Our protocol can support any number of users, as well as that of L2 sequencers, assuming the presence of an L2 consensus protocol for the latter to govern L2 block creation.

Consider a simple end-to-end scenario involving two users, Alice and Bob, which proceeds as follows:

[13] As mentioned in [13], since 2010, Bitcoin script is a language split in two: "Big Script," a set of opcodes used to manipulate signatures, hashes and other cryptographic objects; and "Small Script," a set of opcodes that do arbitrary computations but only on 32-bit inputs. Because Big Script only supports equality checking with `OP_EQUAL`, we are forced to use Small Script and emulate the hashes with arithmetic opcodes.

[14] This costs stems from emulating large-field arithmetic in Bitcoin, as required in pairing-based proof systems, see for example [42]. STARKs, which can operate with significantly smaller field sizes, lend themselves to more efficient Bitcoin script emulation [8, 9].

1. Alice deposits 1 BTC into the L2;
2. Over the L2, Alice transfers 1 BTC to Bob;
3. Bob withdraws 1 BTC from the L2.

Below, we describe how the protocol parties act to facilitate the operations involved in this scenario. The description provided in this section is only for illustrative purposes, as a full specification of such a bridge protocol is beyond the scope of this note.

Deposit phase. Alice participates with the signers to sign transactions as follows:[15]

1. Alice creates a 1 BTC deposit transaction tx_{Alice}, transferring the funds to a multi-signature of the m signers;
2. For each of the n operators:
   (a) 2^L presigned transaction flows are created according to Figure 2 or Figure 4 of some program V (to be specified later), split into $q+1$ transactions for each flow, $\{tx_1^i, \ldots, tx_q^i, tx_{End}^i\}_{i \in D}$, where
   - tx_j^i each computes a subprogram of the verifier V;
   - The public key the signature of the transactions tx_j^i are checked against (as specified in Figure 2) is that of the m signers and the given operator;
   - The final presigned transaction in each flow, tx_{End}^i, is different than the others:
     - It does not compute a subprogram of the verifier;
     - The final UTXO pays Alice's 1 BTC to the given operator;
     - It has a second input which spends Alice's deposit transaction tx_{Alice}.

   (b) The presigned transaction flows are stored by the given operator.
3. Alice uploads her deposit transaction tx_{Alice} onchain, which at the end of the deposit process is the only new transaction existing onchain.

The transaction flows above, which are illustrated in Figure 5, will be used during a future withdraw phase to redeem the operator for the 1 BTC he paid out of pocket to Bob by transferring Alice's 1 BTC to them. Therefore, the program V, on the correct execution of which the operator's payment is conditioned on, should reflect the fact that the operator indeed paid 1 BTC to Bob. Naturally, following Section 3.1, to reduce onchain computational cost, it is preferable that V is a verifier of a proof system rather than a program that checks a statement of interest directly.

In our case, the verifier program V, before releasing the funds to the operator, will check the validity of the following inputs:

1. A Bitcoin SPV proof for transaction tx_{Bob} where the operator pays 1 BTC to Bob (it is associated with Alice's deposit transaction tx_{Alice};
2. Proof of all the state transitions of the L2 from the moment of deposit;
3. Proof that Bob indeed burned 1 BTC over the L2 (to withdraw it).

[15] While we also check for an operator's signatures during the transaction flow, we don't require operator interaction during deposit, and can instead have the operator sign during withdrawal (i.e., the online phase).

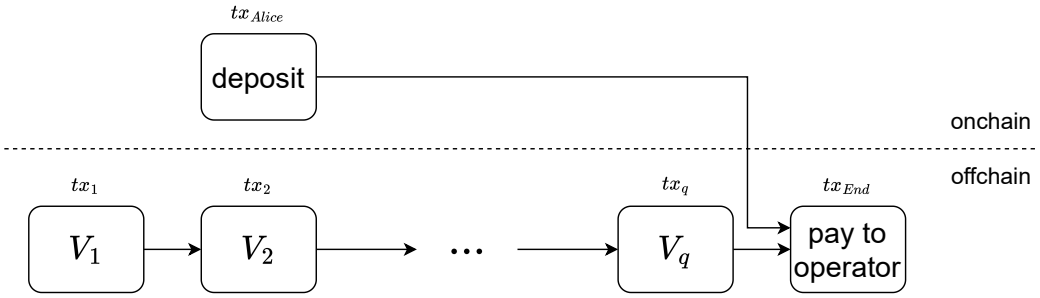

Figure 5: The presigned transaction flow which is set up during the deposit phase, at the end of which, only Alice's transaction is uploaded onchain. The verifier program V is split into q subprograms $V_1, \ldots, V_q$, such that each transaction in the flow computes one of them. The ColliderVM construction is used to ensure input consistency between verifier subprograms, on which we explicitly elaborate in Section 4.3.

We don't provide an argument here on why the above is sufficient to ensure the protocol is protected against malicious behavior, but, nevertheless, some remarks are still in order.

Remark 1. *In Step 1, the requirement to associate the* tx_{Alice} *with* tx_{Bob} *is to uniquely couple the deposit and withdraw together. Without it, a malicious operator could redeem multiple deposits with a single withdrawal.*

Remark 2. *Step 2 guarantees that the L2 chain provided is* correct*, but it does not guarantee that it is also* canonical*. This was addressed by Alpen Labs [5] via proposing to post the canonical L2 state to Bitcoin, ensuring uniqueness through Bitcoin's consensus.*[16]

Remark 3. *In fact, SPV proofs on their own, as mentioned in Remark 1, are not secure over long periods of time, as an attacker could mine a private fork of Bitcoin. This is a nontrivial problem, which, as mentioned in [23], would be solved via the introduction of a new opcode* `OP_BLOCKHASH`*.*

In the absence of the above, multiple teams have proposed ways to bypass this problem, with varying tradeoffs, including superblocks [23], anchor blocks [5], and watchtowers [10].

L2 processing phase. Here, the sequencers process the L2 transactions. In particular, for our scenario, they need to perform the following

1. Mint 1 BTC for Alice by providing a Bitcoin SPV proof of her deposit into the bridge;
2. Process Alice's request to transfer her 1 BTC to Bob;
3. Process Bob's request to burn his 1 BTC.

Throughout the sequencing of the L2 blockchain, the state is continuously being proven. In addition, following Remarks 2 and 3, we incorporate the necessary changes to the base protocol.

Withdrawal phase. To facilitate Bob's withdrawal, the operator begins by performing the following:

[16] In case of multiple L2 states being posted to Bitcoin simulatenously, one can employ a *fork choice rule* to deterministically choose the canonical state among them.

1. Operator pays Bob out of pocket via a transaction tx_{Bob}, associating it with Alice's deposit tx_{Alice};
2. Executes a prover program P to generate a proof π for V;[17]
3. Initiates the deposit redeem process on Alice's transaction tx_{Alice} by doing the following:
   (a) Find r such that (π, r) is a valid B-pair $H(\pi, r)|_B = d \in D$;
   (b) Given π and r, execute the deposit redeem flow $tx_1^d \to \ldots \to tx_q^d \to t_{End}^d$, where tx_{End}^d also spends tx_{Alice}, transferring the funds to the operator.

Remark 4. *Using ColliderVM as above to build an L2 bridge brings a* constant denomination UX problem*, as deposits (and withdrawals) in the system must always be for an integral multiple of a fixed quantity of Bitcoins, which hurts the user's convenience of using the bridge.*

Here, this quantity can be arbitrarily small, while in BitVM2, which also has a similar problem, there is a moderate lower bound on this quantity due to the cost of the assert transaction *[23, Section 5].*

4 Feasibility and Tradeoffs

In this section we compare ColliderVM to BitVM2 for the goal of bridging from Bitcoin to an L2. Currently, the feasibility of our approach depends on the existence of a *Bitcoin friendly hash function*, namely a hash function that can be implemented with as few Bitcoin script opcodes as possible. In addition, we conjecture that our approach is far from optimal, and further research into it could reduce or eliminate many of its drawbacks.

Remark 5 (Comparison with BitVMX)**.** *BitVMX [19, 20] is an improved version of BitVM1 [22], which we choose not to compare to ColliderVM. This is mainly due to the prevalence of BitVM2 in the industry, which also has a better security model than BitVMX by virtue of allowing* any onlooker *to validate the protocol, while BitVMX has a closed validator set.*

Nevertheless, we remark here that the BitVMX approach trades this off with other benefits. For instance, the closed validator set allows for interaction *in the defrauding process, which makes the use of a proof system* optional*, as well as avoids the* 26 *bytes-per-bit Winternitz storage overhead of BitVM2* completely*.*

Below, we highlight the main benefits and drawbacks of our approach for building an L2 bridge. Then, we discuss its overall feasibility.

4.1 Benefits

Capital efficiency. The main selling point of our approach is the bridge operator's capital efficiency. In BitVM2, the bridge operator pays the withdrawer out-of-pocket, needing to wait for the fraud proof window to get reimbursed, which should be sufficiently long to be secure. For ColliderVM, in contrast, the operator can process a user's withdrawal and get reimbursed immediately.

[17] Only parts of π will be available to each individual verifier subprogram V_i. Indeed, it is only necessary to share a *commitment* to π across transactions and decommit the relevant part. We discuss in Section 4.3 how to perform this for a STARK verifier.

Operator (non)interaction. During transaction presigning, the signers only need to know the long-term public keys of the operators. This simplifies communication, as the operators don't need to be online during this phase to interact with the signers. In contrast, in BitVM2, the operators must share one-time signature public keys to be encoded in the locking script.

Cross-transaction storage overhead. Since our approach doesn't use one-time signatures, instead replacing it with a hash collision puzzle, our storage overhead depends on how many opcodes are required to evaluate the hash. As mentioned above, if we use a $\sim$ 45K implementation of BLAKE3, as it hashes 512-bit message blocks, it yields an overhead of 88 bytes-per-bit, compared to just 26 bytes-per-bit in BitVM2. Nevertheless, we believe this to be an overall benefit, as the overhead becomes more favorable if we only commit to parts of the input (as in the STARK verifier application in Section 4.3), and it can get even cheaper if an efficient hash function is used.

Simple flow. As seen in Figure 5, when compared to [23, Figure 7], ColliderVM does not rely on fraud proofs and thus enjoys a drastically simpler presigned transaction flow compared to that of BitVM2. This is beneficial not just in terms of safe implementation but also for analyzing the security model, which, in the case of ColliderVM, does not involve economic incentives for the onlookers.

Additional benefits. Other benefits for ColliderVM include not needing to perform the following, which are required in the BitVM2 case:

- Keeping the presigned transactions public, to allow onlookers to defraud the computation;
- Having the operators keep their (possibly many) Winternitz signatures private, or forfeit their ability to get reimbursed by the protocol.

4.2 Drawbacks

While ColliderVM enjoys some very attractive benefits, the current approach also has considerable drawbacks. We discuss these from a more practical perspective in Section 4.3.

Presigned transaction processing and storage. ColliderVM requires a significant amount of signer compute and operator storage for each deposit into the bridge. This places a limit on the total number of deposits into the bridge that weren't yet reimbursed due to withdrawals.

Offchain computational cost To reimburse a withdrawal, an operator needs to find a hash preimage, which, due to the computational intensity of this task, could limit the withdrawal capacity of the bridge.

Onchain computational cost. As discussed in the introduction, if our computation script f is split into subfunctions $f_1, \ldots, f_k$, BitVM2 has the benefit of only requiring $\max_i |f_i|$ script size to execute onchain *in the worst case*, as opposed to the entire cost of $\sum_i |f_i|$, which is always required by ColliderVM.

4.3 Feasibility

We take the OP_CAT-based Circle STARK verifier [1] for a Fibonacci-related statement as an example. While it does not verify something generic, it also doesn't employ some optimizations, so we believe that a verifier for a complex statement, as in Section 3.2, might end up in the same ballpark, if not lower in cost.

Our considered verifier has the following parameters

- Verifier script length is ~ 3.5 million opcodes;
- There are ~ 1700 calls to `OP_SHA256`;
- The proof size is 26KB.

STARK verification mostly consists of M31 arithmetic and hash calls. Thus, if we replace calls to `OP_SHA256` with emulated hashes, the cost will largely depend on the script size of the hash. For example, since BLAKE3 costs 45K opcodes, the total cost becomes

$$3,500,000 + 1,700 \times 45,000 = 80,000,000.$$

However, if, say, a Bitcoin-friendly hash function costs 4K opcodes, the total cost becomes just

$$3,500,000 + 1,700 \times 4,000 = 10,300,000,$$

which is attractive since building a rollup on top of such a verifier can amortize this cost away.

Splitting the proof. As noted above, the proofs for the STARK verifier have a relatively long length of 26KB. If we were to share the entirety of the proof across transactions, with BLAKE3, this would result in an overhead of 88 bytes-per-bit to compute BLAKE3, incurring an additional script cost of 18.3MB, which is non-negligible. As discussed above, switching to a Bitcoin-friendly hash function could reduce it tenfold to just 1.8MB.

Finally, we note that ColliderVM allows "splitting the proof" for a STARK verifier program, i.e., making it so only parts of the proof are read by each subprogram. This means that less data is shared across transactions, reducing storage overhead. Formally, instead of computing $f(x) = f_1(x) \wedge f_2(x) \wedge f_3(x)$, where all subfunctions read the same input, it is possible to generalize function evaluation to the equation

$$f(x, a, b, c) = f_1(x, a) \wedge f_2(x, b) \wedge f_3(x, c),$$

where each subfunction can have an exclusive part of the input, meaning it does not need to be shared across the transactions. In Figure 6 we give an example of how to save cross-transaction storage overhead for the common task of multiple Merkle decommitments, a major cryptographic task in STARK verification. Moreover, a general discussion over this equation and a comparison to how splitting computation is done in BitVM is given in Appendix A.

Operating costs. The main drawback of our approach is its fairly expensive operating costs. As mentioned in Section 2, for 97 bits of security (which, as discussed, might be too conservative), we can choose the parameters $L = 46$ and $B = 120$. This will require, per deposit, 2^{46} presigned flows to be stored on a hard drive, and an operator will need to evaluate 2^{74} hashes to process a withdrawal. The tradeoff between security and the number of presigned flows is shown in Figure 3.

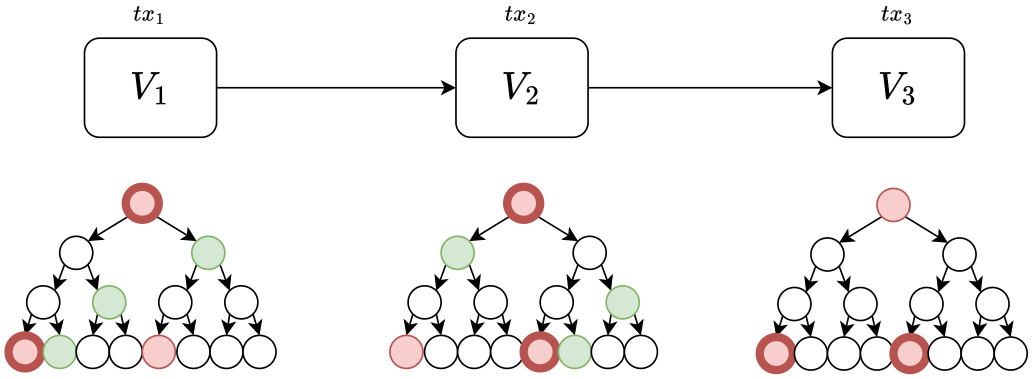

Figure 6: A common operation in a STARK verifier is to read values from a Merkle tree and perform some algebraic checks on the accessed values. In this diagram, V_1 is responsible for reading one value from the tree, V_2 is responsible for reading another value, and V_3 performs an algebraic check with these values. In colors, **red** nodes represent values shared among transactions, with **bold** nodes being actively accessed. In contrast, **green** nodes represent values that are given as regular witnesses to a transaction. Note that the **green** values don't need to be shared across transactions. In practice, these Merkle decommitments constitute a significant fraction of a STARK proof.

If, instead, we opt for a *triple collision* construction, we will set the parameters $L = 39$ and $B = 110$. This will require storing 2^{39} presigned flows per deposit on a hard drive, and an operator will need to evaluate 2^{71} hashes to process a withdrawal. The tradeoff here is the increased Bitcoin script size. More concretely, suppose the STARK verifier is optimized enough and can be split into 4 subfunctions, each ~ 2 million in size (meaning the verifier costs ~ 8 million opcodes). Then, the construction will use $\binom{4}{2} = 6$ transactions, each $\sim 2 \times 2 = 4$ million in size, costing ~ 24 million opcodes overall. This construction significantly improves as the verifier cost decreases.

Estimating monetary costs. We would like to argue that the costs of operating ColliderVM are low to moderate and that the costs of a successful adversarial attack are high. Fortunately for the analysis, the dominant operations both for the legitimate users and for the attack are the hash computation and these are the same as in Bitcoin mining. As a result, we can easily price these operations in Bitcoin terms.

Take for example setting the parameters to $L = 36$ and $B = 100$. This requires 2^{36} presigned flows; an honest operator needs to perform 64 bits of work to evaluate f, while a malicious operator needs 82. As mentioned, currently (April 2025) mining Bitcoin require 78 bits of work for which the miner gets 3.125 BTC. Therefore the attacker needs to spend work equivalent to around 50 BTC. The legitimate operator, at least for the hashing part, needs work that is $\approx 3.125/2^{14}$ BTC, which is very roughly and very noisily \$15.

We also need to consider about the signature generation cost from the presigned flow. Estimating this in Bitcoin is harder, but the computationally dominant part of generating a Schnorr signature is generating a random multiple of the generator. This is independent of the actual value signed and can be computed in the background, so we do not expect it to require significant costs. There are also many optimizations possible (e.g. batch exponentiation [27]).

5 Conclusion

In this note, we have shown that presigned transactions, together with hash collision puzzles, can be used to verify computation onchain and build L2 bridges without relying on fraud proofs. Furthermore, it enables STARK verification on Bitcoin, which was previously too expensive to use due to STARK's longer proofs not combining favorably with contemporary methods to share data across transactions.

Our construction relies on a 1-out-of-n honesty assumption and on the hardness of finding hash collisions. While it has notable attractive benefits, such as avoiding the capital inefficiency problem due to fraud proofs, it also has significant drawbacks. Nevertheless, we believe that our protocol is far from optimal, and this approach is fruitful for enabling new applications on top of Bitcoin. In particular, we identify the following avenues for further research:

1. Finding Bitcoin-friendly hash function implementations;
2. Optimizing STARK verifier implementations on top of Bitcoin;
3. Reducing the overall work of the signers;
4. Reducing the amount of data needed to be stored on a hard drive;
5. Increasing the computational gap between honest and malicious operators.

As a final note, interestingly, our construction can also be used in *fraud proof mode of operation*, by including a defrauding process to the Bitcoin script. While this retains some of the fraud proof limitations, it results in an overall simpler flow compared to current techniques of using fraud proofs on top of Bitcoin. We leave research in this direction to future work.

5.1 Acknowledgments

We thank Weikeng Chen for helpful feedback and for sharing with us data about his STARK verifier implementation [1], on which we relied in Section 4.3. We also thank Ittay Dror, Ariel Elperin, Ethan Heilman, and John Light for valuable discussions and feedback.

References

[1] Bitcoin Wildlife Sanctuary. https://github.com/Bitcoin-Wildlife-Sanctuary/.

[2] SNARK Verifier in Bitcoin Script. https://bitvm.org/snark.html.

[3] Transactions. https://developer.bitcoin.org/devguide/transactions.html.

[4] Wrapped Tokens: A multi-institutional framework for tokenizing any asset, 2019. https://www.wbtc.network/assets/wrapped-tokens-whitepaper.pdf.

[5] Alpen Labs Team. Introducing the Strata bridge. https://www.alpenlabs.io/blog/introducing-the-strata-bridge.

[6] M. Bartoletti, S. Lande, and R. Zunino. Bitcoin covenants unchained. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III 9*, pages 25–42. Springer, 2020.

[7] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In Y. Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, 2011.

[8] W. Chen. How to do Circle STARK math in Bitcoin?, 2024. https://hackmd.io/@l2iterative/SyOrddd9C.

[9] W. Chen. New multiplication algorithm for Circle STARK, 2024. https://hackmd.io/@l2iterative/Byg8h1MsC.

[10] Citrea Team. Citrea's BitVM-Based Bitcoin Bridge - Clementine's Latest Design. https://www.blog.citrea.xyz/citrea-bitvm-bitcoin-bridge-clementine-latest-design/.

[11] J. Groth. On the size of pairing-based non-interactive arguments. In M. Fischlin and J. Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[12] U. Haböck, D. Levit, and S. Papini. Circle starks. *IACR Cryptol. ePrint Arch.*, page 278, 2024.

[13] E. Heilman, V. I. Kolobov, A. M. Levy, and A. Poelstra. ColliderScript: Covenants in bitcoin via 160-bit hash collisions. Cryptology ePrint Archive, Paper 2024/1802, 2024.

[14] E. Heilman and A. Sabouri. BIP-347 - OP_CAT in Tapscript, Dec. 2023. github.com/bitcoin/bips/blob/master/bip-0347.mediawiki.

[15] M. Jonas. Optimizing Algorithms for Bitcoin Script, June 2024. bitvmx.org/knowledge/optimizing-algorithms-for-bitcoin-script.

[16] U. Kirstein, S. Grossman, M. Mirkin, J. Wilcox, I. Eyal, and M. Sagiv. Phoenix: A formally verified regenerating vault. *arXiv preprint arXiv:2106.01240*, 2021.

[17] V. I. Kolobov. The path to general computation on Bitcoin, 2024. https://starkware.co/blog/general-computation-on-bitcoin/.

[18] M. Komarov. Bitcoin PIPEs - Covenants and ZKPs on Bitcoin Without Soft Fork, May. 2024. www.allocin.it/uploads/placeholder-bitcoin.pdf.

[19] S. D. Lerner, R. Amela, S. Mishra, M. Jonas, and J. Á. Cid-Fuentes. Bitvmx: A cpu for universal computation on bitcoin. *arXiv preprint arXiv:2405.06842*, 2024.

[20] S. D. Lerner, M. Jonas, and A. Futoransky. ESSPI: ECDSA/Schnorr Signed Program Input for BitVMX. 2025. https://bitvmx.org/files/esspi-ecdsa-input-bitvmx.pdf.

[21] J. Light. Validity rollups on bitcoin. .

[22] R. Linus. Bitvm: compute anything on bitcoin, 2023. bitvm.org/bitvm.pdf.

[23] R. Linus, L. Aumayr, A. Zamyatin, A. Pelosi, Z. Avarikioti, and M. Maffei. Bitvm2: bridging bitcoin to second layers, 2024. bitvm.org/bitvm_bridge.pdf.

[24] G. Maxwell. Coincovenants using scip signatures, an amusingly bad idea. *Bitcoin-talk*, Aug 2013. https://bitcointalk.org/index.php?topic=278122.0.

[25] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2017.

[26] M. Möser, I. Eyal, and E. Gün Sirer. Bitcoin covenants. In *Financial Cryptography and Data Security: FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers 20*, pages 126–141. Springer, 2016. https://maltemoeser.de/paper/covenants.pdf.

[27] D. M'Raïhi and D. Naccache. Batch exponentiation: A fast dlp-based signature generation strategy. In L. Gong and J. Stearn, editors, *CCS '96, Proceedings of the 3rd ACM Conference on Computer and Communications Security, New Delhi, India, March 14-16, 1996*, pages 58–61. ACM, 1996.

[28] A. Poelstra. CAT and Schnorr Tricks I, Janary 2021. www.wpsoftware.net/andrew/blog/cat-and-schnorr-tricks-i.html.

[29] A. Poelstra. CAT and Schnorr Tricks II, Feb 2021. www.wpsoftware.net/andrew/blog/cat-and-schnorr-tricks-ii.html.

[30] S. Roose. BIP-Proposed - OP_TXHASH and OP_CHECKTXHASHVERIFY, Sept. 2023. github.com/bitcoin/bips/pull/1500.

[31] J. Rubin. FE'd Up Covenants, May. 2024. rubin.io/public/pdfs/fedcov.pdf.

[32] J. Rubin. Un-FE'd Covenants: Char-ting a new path to Emulated Covenants via BitVM Integrity Checks, 2024. https://rubin.io/public/pdfs/unfedcovenants.pdf.

[33] J. Rubin and J. OBeirne. BIP-119 - CHECKTEMPLATEVERIFY, Jan. 2020. github.com/bitcoin/bips/blob/master/bip-0119.mediawiki.

[34] N. Satoshi. CVE-2010-5137: OP_LSHIFT crash, Aug 2010. en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2010-5137.

[35] N. Satoshi. "misc changes", Aug 2010. github.com/bitcoin/bitcoin/commit/4bd188c43.

[36] sCrypt. Bitcoin OP_CAT Use Cases Series #4: Recursive Covenants, 2024. https://scryptplatform.medium.com/bitcoin-op-cat-use-cases-series-4-recursive-covenants-6a3127a24af4.

[37] J. Swambo, S. Hommel, B. McElrath, and B. Bishop. Bitcoin covenants: Three ways to control the future. *arXiv preprint arXiv:2006.16714*, 2020.

[38] J. Swambo, S. Hommel, B. McElrath, and B. Bishop. Custody protocols using bitcoin vaults. *arXiv preprint arXiv:2005.11776*, 2020.

[39] Taproot Wizards. A Prototype Vault using CAT. https://github.com/taproot-wizards/purrfect_vault.

[40] P. Wuille, J. Nick, and A. Towns. BIP-341 - Taproot: SegWit version 1 spending rules, Janary 2020. github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.

[41] P. Wuille, J. Nick, and A. Towns. BIP-342 - Validation of Taproot Scripts, Janary 2020. github.com/bitcoin/bips/blob/master/bip-0342.mediawiki.

[42] D. Zakharov, O. Kurbatov, M. Bista, and B. Bist. Optimizing big integer multiplication on bitcoin: Introducing w-windowed approach. *Cryptology ePrint Archive*, 2024. eprint.iacr.org/2024/1236.

[43] M. Šinkec. Implementing a Bridge Covenant on OP_CAT-Enabled Bitcoin: A Proof of Concept, 2024. https://starkware.co/blog/implementing-a-bridge-covenant-on-op-cat-bitcoin/.

A Splitting computation in ColliderVM

Bitcoin scripts are limited by at most 4 million opcodes, and are required to use no more than 1000 stack elements. Therefore, if a function f doesn't fit these requirements, we must split it into n subfunctions $f_1, \ldots, f_n$. One way to do it is as follows: For a function $f\colon \mathcal{X} \to \mathcal{Y}$ and a *fixed* $y \in \mathcal{Y}$, it holds that $\exists x : f(x) = y$ if and only if there are additional values $z_1, \ldots, z_{n-1}$ such that $f_1(x) = z_1, f_2(z_1) = z_2, \ldots, f_n(z_{n-1}) = y$. This splitting satisfies that $f(x) = f_n(f_{n-1}(\ldots f_1(x)\ldots))$, which is how computation splitting is done in BitVM2 [22]. Here, however, we use *another* equivalent way to split computation. To this end, suppose that $f\colon \mathcal{X} \to \mathcal{Y}$, and let $g_1, \ldots, g_n : \mathcal{A} \to \{0,1\}$ be *Boolean functions*. For our splitting, we'll require that

$$\exists x : f(x) = y \iff \exists \alpha, a_1, \ldots, a_n : g_1(\alpha, a_1) = 1 \wedge \ldots \wedge g_n(\alpha, a_n) = 1.$$

We show below that these two ways to split computation into n subfunctions are equivalent.

$\{f_i\}_i \Rightarrow \{g_i\}_i$: Let α be a some commitment[18] on $(x, z_1, \ldots z_{n-1}, y)$, and let a_i be the decommitment on z_{i-1} and z_i (where we view $z_0 = x$ and $z_n = y$). Then, define g_i such that $g_i(\alpha, a_i) = 1$ if and only if $f_i(z_{i-1}) = z_i$.

$\{g_i\}_i \Rightarrow \{f_i\}_i$: Let $z_0 = x = (\alpha, a_1)$, $z_n = y$, and $z_i = (\alpha, a_{i+1})$ for $i = 1, \ldots, n-1$. Then, define f_i such that $f_i(z_{i-1}) = z_i$ if and only if $g(\alpha, a_i) = 1$.

Write-once memory. To directly obtain a splitting of f into binary functions $\{g_i\}_i$ we'll assume f is a Bitcoin script program with an additional access to *write-once memory*, only allowing to set each memory address once. Then, we let g be a binary function that accepts a *commitment* to this memory (including the input), together with decommitments to each memory address. Instead of setting memory values itself, g will only check that the values are set up correctly, according to a valid execution of f, and return 1 if all checks pass (implying g returns 1 if and only if $f(x) = y$).

Finally, we can split g into $\{g_i\}_i$ by performing only some of these checks in each g_i. Figure 6 illustrates this process for STARK verification. We also note that this allows us to use functions f whose Bitcoin script implementation is not limited to at most 1000 stack elements, simply because each g_i can decommit only parts of the whole memory.

[18] To commit a vector $(x, z_1, \ldots, z_{n-1}, y)$ to a value α, we'll simply let α be the root of a Merkle tree whose leaves correspond to this vector. This allows decommitting each entry individually.