Enforcing arbitrary constraints on Bitcoin transactions

Federico Barbacovi[1] and Enrique Larraia*

Teranode Group
f.barbacovi@teranode.group, elarraia@gmail.com

Abstract. The challenge of enforcing constraints on Bitcoin transactions has recently gained a lot of attention. The current approach to solve this problem falls short in certain aspects, such as privacy and programmability. We design a new solution that leverages zkSNARKs and allows enforcing arbitrary constraints on Bitcoin transactions while maintaining some information private. Our approach also bypasses the non-Turing completeness of Bitcoin Script, allowing the enforcement of unbounded constraints, namely constraints that repeat a certain operation an unbounded number of times.

Keywords: Bitcoin · Zero Knowledge · Smart Contract

1 Introduction

Bitcoin [18] is a PoW blockchain whose scripting language, Bitcoin Script, is non-Turing complete, and therefore has limited functionalities (e.g., loops are not allowed). The restricted capabilities of Bitcoin Script meant that for a long time only very simple scripts like [P2PK] and [P2PKH] have been used. Recently, there has been a growing interest in expanding the capabilities of Bitcoin Script [1, 13, 15, 16]. We continue this line of work by introducing a new techinque that we call reftx, which allows the enforcement of *arbitrary conditions* on Bitcoin transactions.

In Bitcoin, the conditions under which funds can be spent are encoded in *locking scripts*, which are programmes written in Bitcoin Script. As Bitcoin Script does not natively support access to the spending transaction, it is not obvious that locking scripts can enforce conditions on it.[1] The standard approach to construct Bitcoin scripts that enforce conditions on the spending transaction is to force the spender to push the spending transaction as part of the unlocking script (which is the input over which the locking script is evaluated) [11, 17].

While this approach is effective in most cases, it comes short in many aspects: it roughly doubles the size of the spending transaction,[2] it is incompatible

* Formerly nChain researcher. Work done while at nChain.

[1] Except for signing conditions, e.g., the locking script [P2PK] hard-codes a public key P, so that the funds locked with this script can only be spent if the spending transaction is signed with the secret key corresponding to P.

[2] This is because a digest of the spending transaction has to be added to the unlocking script.

with applications that require keeping some information private, and it is often complex to implement as the conditions to be enforced must be programmed in Bitcoin Script. Most importantly, enforcing conditions via Bitcoin Script means we are still restricted by the non-Turing completeness of the language. In particular, we cannot enforce *unbounded conditions*, by which we mean conditions that involve a computation that has been repeated an unknown and unbounded number of times. For example, in Bitcoin Script we can encode the conditions "$H = F^i(x)$ for $i \leq n$," but we cannot encode the condition "$H = F^i(x)$ for some $i \in \mathbb{N}$."

Our techniques address these problems, enabling users to enforce (possibly unbounded) conditions which (may) involve private off-chain data. Furthemore, user can enforce these conditions in the language of their choice (we use Rust), without diving into the intricacies of Bitcoin Script.

The solution we propose is based on zkSNARKs, and in particular on deploying a zkSNARK verifier on-chain.[3] The idea is to build a circuit which encodes all the conditions we want to enforce on the spending transaction, and then verify on-chain a zero-knowledge proof π that the spending transaction satisfies the circuit. The problem we have to face is that of linking the spending transaction to the public instance against which π is verified. For this, we introduce the notion of an *integrity mechanism* which is *script compatible*.

As an application of our techinques, we show how to *provably burn* a token on Bitcoin. That is, we construct a locking script the enforces that one of the inputs to the spending transaction is a descendant of a certain *genesis* transaction, and that such input is burnt, i.e., it is spent to an `OP_0 OP_RETURN` output. We remark that this condition is unbounded, as at the point of constructing the locking script there is no way to know how many times the output of the genesis transaction has been spent before being burnt. Therefore, provably burning a token on Bitcoin would not be possible without our techniques.[4]

Acknowledgements The authors would like to thank L. Lunardon and W. Zhang for carefully reviewing a first version of this paper.

2 Preliminaries

2.1 Bitcoin

The Bitcoin blockchain [18] is formed by a sequence of blocks $\{B_i\}_{i\in\mathbb{N}}$, each containing an ordered sequence of transactions $B_i = (\mathsf{tx}_1, \mathsf{tx}_2, \dots)$.

As Bitcoin is based on the Unspent Transaction Output (UTXO) model, transactions are made of inputs and outputs. Each transaction tx can spend

[3] As our techniques rely on deploying a zkSNARK verifier on-chain, they are best suited to tackle use cases where either privacy is required, or the computation would be more expensive (or impossible) to encode in Bitcoin Script than in a circuit.

[4] By Bitcoin here we mean BSV [2], which does implement SegWit. In BTC, if `OP_CAT` is re-enabled, provably burning a token is possible via CAT20 locking scripts [17].

the outputs of another transaction tx' by referencing them. More precisely, a transaction has the following form[5]

$$\mathsf{tx} = (\mathsf{version}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{locktime})$$

where inputs is the list of transaction inputs (TXIN) and outputs is the list of transaction outputs (TXO). A TXO is simply a couple $\mathsf{txo} = (\mathsf{amount}, \mathsf{lock})$ of an amount, and a *locking script* specifying the conditions of spending. A TXIN is given by[6]

$$\mathsf{txin} = (\mathsf{prev_tx}, \mathsf{prev_index}, \mathsf{unlock}, \mathsf{sequence})$$

where $(\mathsf{prev_tx}, \mathsf{prev_index})$ identifies which output is being spent, and unlock is the *unlocking script*, containing the data required to spend the output.

An input txin successfully spends an output txo if txin is the first input to spend txo (no double-spending), and the spending conditions dictated by lock are satisfied.

2.2 Bitcoin Script

Spending conditions in Bitcoin are specified via locking scripts, which are programmes written in Bitcoin Script[7] [4]. Bitcoin Script is a non-Turing complete, stack-based programming language composed of a handful of instructions called opcodes [5]. Bitcoin Script programmes are written using reverse Polish notation and are executed by the Bitcoin Script Engine. The execution starts by pushing onto the stack the data contained in the unlocking script (which must contain *only* data), and then continues with the execution of the locking script. The spending condition of a UTXO is satisfied if the data contained in the unlocking script is an input for which the execution of the locking script terminates with true.

We introduce some notation that will be used throughout the rest of the paper:

- $\langle\langle l \rangle\rangle$ denotes data that is hard-coded in the locking script
- $[\![a]\!]$ denotes data on top of the stack
- $[\![b]\!] \leftarrow [\![a]\!]$ [foo] means that if we execute [foo] on a stack whose top element is a, then after the execution the top of the stack is given by the element b

2.3 SNARKs

Fix a finite field $\mathbb{F}$. An arithmetic circuit over $\mathbb{F}$ is a map $\mathsf{C}\colon \mathbb{F}^n \times \mathbb{F}^h \to \{0,1\}$. The *circuit satisfiability problem* associated to C is defined by the relation

$$\mathcal{R}_{\mathsf{C}} = \{(x, w) \in \mathbb{F}^n \times \mathbb{F}^h : \mathsf{C}(x, w) = 1\}$$

[5] For the meaning of version and locktime, see [6].
[6] For the meaning of sequence, see [6].
[7] Different versions of Bitcoin have different scripting languages. In this work, we focus on BSV [2], which supports large number arithmetic and string manipulation.

A pre-processing, succinct, non-interactive, argument of knowledge (SNARK) for $\mathbb{F}$-arithmetic circuit satisfiability is a triplet of algorithms $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ that allow proving and verifying statements of the form "$\exists w : (x, w) \in \mathcal{R}_{\mathsf{C}}$".

More precisely, on input a security parameter and an arithmetic circuit C, G returns a couple $(\mathsf{pk}, \mathsf{vk})$ of proving/verifying keys. On input $(x, w) \in \mathcal{R}_{\mathsf{C}}$ and the proving key pk, P produces a proof $\pi \leftarrow \mathsf{P}(\mathsf{pk}, x, w)$. On input x, π and the verifying key vk, V outputs $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{V}(\mathsf{vk}, x, \pi)$, indicating whether they accept the statement "$\exists w : (x, w) \in \mathcal{R}_{\mathsf{C}}$".

The argument is non-interactive in the sense that P only needs to send a single message to V: the proof π.

Succinctness means that the length of the proof is constant, i.e., if λ is the security parameter, then $|\pi| \sim O_\lambda(1)$, and that $\mathsf{V}(\mathsf{vk}, x, \pi)$ runs in time proportional to the length of the public instance: $O_\lambda(|x|)$.

On-chain SNARKs Because of their properties, SNARKs are well suited for on-chain deployment:

- The fact that the proof size is constant means that a minimal amount of data must be published on-chain to verify the existence of a valid witness w for the public instance x.
- As the description of the verifier V is independent of the circuit C, a Bitcoin Script implementation of V can be re-used for different circuits by hard-coding in the script different verifying keys.

Recently, there has been a strong interest in implementing SNARK verifiers in Bitcoin Script. As our implementations are based on BSV, we use the Bitcoin Script implementation of Groth16 [12] presented in [1].

3 Enforcing constraints on transaction data

Throughout this section we refer to the transaction that is currently being validated by the miners as the *spending transaction* stx, and denote the output that is being spent as stxo.

3.1 Types of transaction constraints

On-chain data location On-chain data y is anything that appears in a Bitcoin transaction. When validating the spending of an output stxo in transaction stx, we split on-chain data in three parts $y := (l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}})$.

- *Locking data* l_{stxo}. The data present in the locking script of the output stxo. That is, data in the locking script that is currently being executed.
- *Unlocking data* u_{stx}. The data present in the unlocking script of the input (of stx) that is currently being verified.
- *Spending data* s_{stx}. The data present in other fields of the spending transaction stx, excluding the unlocking scripts of *any* input of stx.

Figure 1 depicts locking and unlocking data in for a couple of transactions tx, stx. The spending data s_{stx} is given by the entire transaction stx, except for its unlocking scripts.

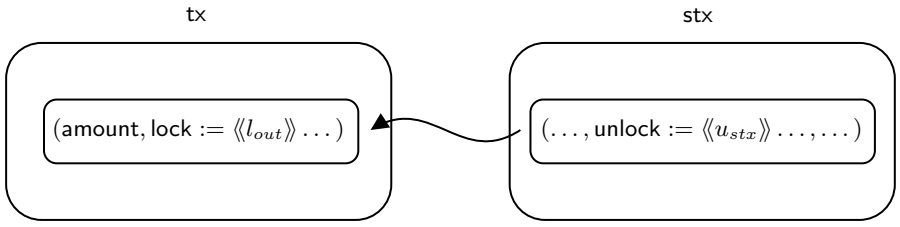


Fig. 1. Locking data l_{stxo} and u_{stx} in a couple of transactions tx, stx.

Transaction constraints A transaction constraint is any relation that holds between on-chain data. Based on where the data appears in the transaction, we distinguish two types of constraints.[8]

- *Spending condition constraint (type 1).* The on-chain data appears only as locking and unlocking data $y = (l_{\mathsf{stxo}}, u_{\mathsf{stx}}, \bot)$. This is how standard spending conditions deal with data. For example, a [P2PK] script has a hard-coded public key in it and reads the signature from the unlocking script.
- *Inter-transaction constraint (type 2).* The on-chain data is present everywhere $y = (l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}})$. That is, we are dealing with relations concerning locking, unlocking and spending data.

3.2 Constraints as NP relations

A constraint imposed to on-chain data y is modelled as an arithmetic circuit C with public instance y and private (off-chain) witness w. The constraint is satisfied if $\mathsf{C}(y, w) = 1$. Thus, the NP relation we consider is

$$\mathcal{R}_{\mathsf{C}} = \{(y; w) \mid \mathsf{C}(y, w) = 1 \ \wedge \ y = (l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}})\}. \tag{1}$$

Its associated language is $\mathcal{L}_{\mathsf{C}} = \{y \mid \exists w \text{ s.t. } (y; w) \in \mathcal{R}_{\mathsf{C}}\}$. Our goal is to verify on-chain SNARK proofs for statements of the form "$y \in \mathcal{L}_{\mathsf{C}}$".

[8] There is a third type of constraint that we do not take into account, when the on-chain data appears only in the spending transaction $y := (\bot, u_{\mathsf{stx}}, s_{\mathsf{stx}})$. An example of such a constraint is a limit on the output amounts of stx. The difference between this type of constraint and inter-transaction constraints is that the former is only concerned with the spending transaction, and does not require knowledge of any locking data when specifying the constraint.

On proving constraints off-chain. Off-chain proving and on-chain verifying permits to deploy on Bitcoin applications that do not allow disclosing sensitive data and yet must guarantee that certain relations holds between the on-chain data y and the off-chain data w. We note that even if these relations can be coded up as a Bitcoin script, we cannot enforce tem unless we reveal w on-chain.

3.3 Building block: integrity mechanism

At the core of our technique lies a mechanism to check the integrity of the spending data s_{stx} *in-script*. By this we mean that the scheme can be implemented in the Bitcoin script language, and that the verification algorithm takes as implicit input data s_{stx}.

A *transaction integrity* scheme TI is basically a binding commitment for spending data. It consists of two algorithms:

- $\mathsf{TI.commit}(s_{\mathsf{stx}}, \sigma) \to \tau$: On input spending data s_{stx} and auxiliary info σ, it outputs an integrity tag τ.
- $\mathsf{TI.verify}(s_{\mathsf{stx}}, \sigma, \tau) \to \{\mathsf{accept}, \mathsf{reject}\}$: On input spending data s_{stx}, auxiliary info σ, and integrity tag τ, either accepts or rejects.

The auxiliary information σ can be seen as the opening information in standard commitment schemes, the difference being that above is an input, not an output, of the commit algorithm.

Standard properties These are known properties borrowed from cryptographic commitment schemes.[9]

Completeness. For all s_{stx} there exists σ such that

$$\mathsf{accept} \leftarrow \mathsf{TI.verify}(s_{\mathsf{stx}}, \sigma, \mathsf{TI.commit}(s_{\mathsf{stx}}, \sigma)).$$

Computationally binding. It is unfeasible to compute $s_{\mathsf{stx}} \neq s'_{\mathsf{stx}}$ with the same integrity tag. Thus, for any probabilistic polynomial-time algorithm $\mathcal{A}$

$$\mathbb{P}\left[\begin{array}{l} s_{\mathsf{stx}} \neq s'_{\mathsf{stx}} \\ \mathsf{accept} \leftarrow \mathsf{TI.verify}(s_{\mathsf{stx}}, \sigma, \tau) \\ \mathsf{accept} \leftarrow \mathsf{TI.verify}(s'_{\mathsf{stx}}, \sigma', \tau) \end{array} \middle| \; (s_{\mathsf{stx}}, s'_{\mathsf{stx}}, \sigma, \sigma', \tau) \leftarrow \mathcal{A}(\lambda) \right] \leq \mathsf{negl}(\lambda) \qquad (2)$$

where λ denotes a security parameter.

On-chain properties We define two extra properties for a transaction integrity mechanisms. The first one is a mandatory property, the second one is optional.

[9] We do not require *computational hiding* for the integrity tag because the spending data is public anyway.

Script compatibility. The integrity mechanism TI is *script compatible* if there exists two Bitcoin scripts [`TI.commit`] and [`TI.verify`] functionally equivalent to $\mathsf{TI.commit}$ and $\mathsf{TI.verify}$, respectively. Specifically

$$[\![\tau]\!] \leftarrow [\![\sigma, s^*]\!]\ [\texttt{TI.commit}] \text{ iff } \tau \leftarrow \mathsf{TI.commit}(s^*, \sigma) \tag{3}$$

$$[\![\mathsf{true}]\!] \leftarrow [\![\tau]\!]\ [\texttt{TI.verify}] \text{ iff } \mathsf{accept} \leftarrow \mathsf{TI.commit}(s_{\mathsf{stx}}, \sigma, \tau) \tag{4}$$

The notation $[\![e]\!] \leftarrow [\![d]\!]\ [\texttt{script}]$ above means that the top of the stack starts with data d, and after executing the script has data e, see Subsection 2.2. Note that in condition (4) the script [`TI.verify`] takes the spending data s_{stx} and auxiliary information σ as an implicit input. Only the integrity tag τ is explicitly present in the stack.

Succinctness. The integrity mechanism has *on-chain succinctness* if the commitment scheme is length-reducing and the description of the verification script is independent of the spending data s_{stx}. Thus, $|[\texttt{TI.verify}]|, |\tau| \in \mathcal{O}(\lambda)$, where λ is the security parameter for binding.

For the ease of exposition, we defer to Section 3.6 the instantiation of the TI scheme and its corresponding scripts.

3.4 Verifying transaction constraints

Let us suppose that there exists a script [$\texttt{SNARK.verify}\ \mathsf{C}$] to verify proofs π of a given SNARK scheme for circuit C. Relation $\mathcal{R}_{\mathsf{C}}$ in (1) defines the purportedly spending data s_{stx} as part of its instance. However, we must guarantee that the proof that is verified on-chain (pushed to the stack from the unlocking script) attests to the satisfiability of C on input s_{stx}. This is because data from the unlocking script cannot be trusted unless it is validated. This is where the integrity mechanism TI comes to rescue us.

To enforce that C is applied to the spending data s_{stx}, we adopt two standard tricks:

- *Compute off-chain & Verify on-chain*: we push an integrity tag τ via the unlocking script, and we verify its integrity on-chain with [`TI.verify`].
- *Pass a SNARK instance as a witness*: we augment C so that it takes u_{stx}, l_{stxo} and τ as public inputs, s_{stx} and σ as witnesses, and on top of enforcing the constraints of C, it also verifies that τ is the commitment for s_{stx} with auxiliary data σ. Thus, the NP relation is[10]

$$\mathcal{R}_{\mathsf{C}}^{(\mathsf{reftx})} = \{((l_{\mathsf{stxo}}, u_{\mathsf{stx}}, \tau); (s_{\mathsf{stx}}, \sigma, w)) \mid \mathsf{C}_{\mathsf{reftx}}(l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}}, \tau, \sigma, w) = 1\} \tag{5}$$

The augmented circuit $\mathsf{C}_{\mathsf{reftx}}$ is defined in Algorithm 1. In Lemma 1 we will show that statements of the form "$(l_{\mathsf{stxo}}, u_{\mathsf{stx}}, \tau) \in \mathcal{L}_{\mathsf{C}}^{(\mathsf{reftx})}$" are implicitly statements of the form "$y := (l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}}) \in \mathcal{L}_{\mathsf{C}}$", namely, statements about the on-chain data.

[10] We call the augmented relation the reftx relation, as it gets a *reference* to s_{stx}, namely, the integrity tag τ.

Algorithm 1 $\mathsf{C}_{\mathsf{reftx}}$: Succinctly prove constraint C on the spending transaction

Public input (instance): Locking data l_{stxo}, unlocking data u_{stx}, integrity tag τ.
Private input (witness): Spending data s_{stx}, auxiliary information σ, off-chain witness w.
Steps:

1: Check $\mathsf{TI.verify}(s_{\mathsf{stx}}, \sigma, \tau) = \mathsf{accept}$
2: Check $\mathsf{C}((l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}}), w) = 1$
3: If both checks pass, output 1. Else output 0.

The locking script that we deploy on-chain is:

$$\begin{aligned}[\texttt{Enforce constraint C}] := \texttt{OP_DUP OP_TOALTSTACK} \\ \langle\!\langle l_{\mathsf{stxo}} \rangle\!\rangle\ [\texttt{SNARK.verify } \mathsf{C}_{\mathsf{reftx}}] \\ \texttt{OP_FROMALTSTACK } [\texttt{TI.verify}]\end{aligned} \tag{6}$$

where $\langle\!\langle l_{\mathsf{stxo}} \rangle\!\rangle$ means that l_{stxo} is hard-coded in the locking script. The script $[\texttt{Enforce constraint C}]$ is executed on input $\langle \pi \rangle\ \langle u_{\mathsf{stx}} \rangle\ \langle \tau \rangle$. First, $[\texttt{SNARK.verify } \mathsf{C}_{\mathsf{reftx}}]$ verifies that the proof π is valid for input $(l_{\mathsf{stxo}}, u_{\mathsf{stx}}, \tau)$. If this check passes, the tag τ is verified against the real spending data.

Recall from Subsection 2.3 that the description of the verification algorithm of a pre-processing SNARK is independent of the circuit that is verified. Indeed, we only know we are verifying the right circuit when inputting the right verification key to $\mathsf{SNARK.verify}$. To ensure that indeed $\mathsf{C}_{\mathsf{reftx}}$ is verified, we hard-code its verification key vk in the Script verifier. Thus, in (6) we define:

$$[\texttt{SNARK.verify } \mathsf{C}_{\mathsf{reftx}}] := \langle\!\langle vk \rangle\!\rangle\ [\texttt{SNARK.verify}].$$

This is like hard-coding the ECDSA public key in a $[\texttt{P2PK}]$ script to ensure funds are sent to the right address. We use the implementation of the script $[\texttt{SNARK.verify}]$ for Groth16 described in [1].

Lemma 1 (Constraint enforced on on-chain data). *Assume that the script* $[\texttt{Enforce constraint C}]$ *from (6) with hard-coded verification key vk pushes* `true` *to the top of the stack on inputs* $\pi, u_{\mathsf{stx}}, \tau$*. Then,* $y := (l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}}) \in \mathcal{L}_{\mathsf{C}}$ *with very high probability, where* y *is the on-chain data.*

Proof. By the knowledge soundness of the SNARK, we can extract a witness $(s^*_{\mathsf{stx}}, \sigma^*, w)$ such that with high probability

$$\mathsf{TI.verify}(s^*_{\mathsf{stx}}, \sigma^*, \tau) = \mathsf{accept} \quad \text{and} \quad \mathsf{C}((l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s^*_{\mathsf{stx}}), w) = 1$$

As the script $[\texttt{Enforce constraint C}]$ pushed `true` to the stack, it means that

$$[\![\mathsf{true}]\!] \leftarrow [\![\tau]\!]\ [\texttt{TI.verify}] \overset{(4)}{\iff} \mathsf{accept} \leftarrow \mathsf{TI.commit}(s_{\mathsf{stx}}, \sigma, \tau)$$

Therefore, (2) implies $s^*_{\mathsf{stx}} = s_{\mathsf{stx}}$ with high probability, and $y = (l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}}) \in \mathcal{L}_{\mathsf{C}}$.

On-chain succinctness. The verification script [$\texttt{SNARK.verify}\ \mathsf{C}_{\mathsf{reftx}}$] and the proof π do not depend on the size of locking data l_{stxo} and spending data s_{stx}. If the integrity mechanism is also succinct (cf. Section 3.3) then the size of [$\texttt{Enforce constraint}\ \mathsf{C}$] is independent of the size of locking data l_{stxo} and spending data s_{stx}.

3.5 Protection against replay attacks

For a given circuit C, let use write $\widetilde{s}_{\mathsf{stx}}$ for the subset of s_{stx} upon which C enforces constraints. For example, if C enforces the first output of stx to be a [P2PK] output to Alice, then $\widetilde{s}_{\mathsf{stx}}$ is the public key hard-coded in the first output, which must be Alice's. However, stx could have more than one output, in which case C enforces conditions only on the first output.

A proof π attesting to the fact that $\mathsf{C}(l_{\mathsf{stxo}}, u_{\mathsf{stx}}, s_{\mathsf{stx}}) = 1$ can be replayed on any spending transaction stx' such that $\tilde{s}_{\mathsf{stx}} = \tilde{s}_{\mathsf{stx}'}$. This is a problem for the user broadcasting stx because they are succeptible to man-in-the-middle attacks (by e.g. malicious miners).

Our circuit $\mathsf{C}_{\mathsf{reftx}}$ is not succeptible to this type of attack as long as $w \neq \perp$. Indeed, Step (1) of Algorithm (1) enforces a condition on the full spending data[11] s_{stx}, which means that the proof π cannot be replayed on a different stx'.

We note that for a circuit C, if we assume that $l_{\mathsf{stxo}} = \perp$, then C enforces conditions only on u_{stx}, s_{stx}, and w, which means that the data over which C operates is fully specified by the spender. In this case, it is not clear how to leverage an off-chain witness w to prevent a replay attack. However, if $l_{\mathsf{stxo}} \neq \perp$, then the problem is solved. For example, l_{stxo} could be a public key P, and C might enforce that w is the corresponding private key, so that only the owner of P can create a valid proof. The downside of assuming $l_{\mathsf{stxo}} \neq \perp$ is that we need to fix some additional data at the point of locking script construction.

The Proof of Burn application, Section 4, introduces a different solution in which the right of spending a UTXO locked with (6) is demonstrated by spending the tip of a *transaction chain*.

3.6 Implementing the integrity mechanism

In Bitcoin, the opcode `OP_CHECKSIG` [8] verifies a signature present in the stack against the spending transaction stx. The message against which `OP_CHECKSIG` verifies the signature is called *presighash* and is a digest of the spending transaction stx and of the UTXO stxo that is being spent.

More precisely, the function $\mathsf{PreSighash}(-,-,-)$ takes as input the spending transaction stx, the index ix of the input of stx referencing stxo, and the sighash flag[12] flag [7]. Then, it fetches stxo and returns $\mathsf{PreSighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{flag})$ constructed according to the algorithm [3].

[11] This is true only if the integrity mechanism is implemented using the flag $\mathsf{sighash_all}$, see Section 3.6. In case other flags are used, the replay attack can still be carried out.

[12] Sighash flags are used to specify which parts of stx should be signed.

The double sha256 digest of the presighash is called the *sighash*, and we write

$$\mathsf{sighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{flag}) := \mathsf{sha256}(\mathsf{sha256}(\mathsf{PreSighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{flag})))$$

The sighash is a cryptographic commitment to stx, and to build a transaction integrity mechanism we take $\tau = \mathsf{sighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{flag})$. The verification TI.verify is then a functionally-equivalent version of $\mathsf{ECDSA.sign}(\mathsf{sighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{flag}), 1, 1)$ (signing with private key and ephemeral key equal to 1) followed by `OP_CHECKSIG`. The commitment TI.commit is a functionally-equivalent version of $\mathsf{sighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{flag})$. The binding property of the integrity mechanism then reduces to the collision resistance of sha256. We emphasize that the idea of injecting the spending transaction to the stack is already known, see e.g. [11,17,19] and references therein.

We remark that both TI.commit and TI.verify depend on the sighash flag flag. Different flags can be used for different purposes, as they allow to strike a balance between the complexity of the algorithms and what conditions can be enforced on stx. A version of the commit algorithm TI.commit for sighash flag sighash_all, which signs the entire stx, is described in Algorithm 2.

Algorithm 2 TI.commit with flag sighash_all.

Inputs: Spending data s_{stx}, auxiliary information σ
Output: Integrity tag τ
Steps:

1: Parse $\sigma = (\mathsf{amount}, \mathsf{lock})$ as a Bitcoin output: a couple (amount, script).
2: Construct $\mathsf{PreSighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{sighash_all})$ using stx, and σ in place of stxo.
3: $\tau = \mathsf{sighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{sighash_all})$

The verification TI.verify is very similar to TI.commit. It receives the same inputs, and additionally a purported integrity tag τ'. On top of Steps 1 - 3, TI.verify performs two more steps that ensure the integrity of τ'.

Algorithm 3 TI.verify with flag sighash_all.

Inputs: Spending data s_{stx}, auxiliary information σ, purported integrity tag τ'.
Output: true or false
Steps:

1: $\tau \leftarrow \mathsf{TI.commit}(s_{\mathsf{stx}}, \sigma)$
2: $\mathsf{sig} := \mathsf{ECDSA.sign}(\tau', 1, 1)$
3: $\mathsf{ECDSA.verify}(\mathsf{sighash}(\mathsf{stx}, \mathsf{ix}, \mathsf{sighash_all}), \mathsf{sig}, G_{\mathsf{btc}})$ ▷ G_{btc} is the base point of secp256k1

The Bitcoin Script implementation of TI.verify consists in signing τ' with private key and ephemeral key equal to 1, and then running `OP_CHECKSIG` with

the base point G_{btc} hard-coded as the public key. For a detailed explanation of the signing process, see [11].

Optimisation. Note that the locking script (6) is going to be large because it includes a description of the SNARK verifier. As the algorithm $\mathsf{TI.verify}$ involves generating the sighash, it requires hashing the current executing locking script, i.e., (6). We can avoid hashing (most of) it by placing the code separator opcode right before the checksig

$$[\texttt{TI.verify}] := [\texttt{ECDSA.sign}(-, 1, 1)]\ \langle\langle G_{\mathsf{btc}} \rangle\rangle\ \texttt{OP_CODESEPARATOR}\ \texttt{OP_CHECKSIG}$$

This tells `OP_CHECKSIG` to skip anything preceding the separator when forming the script to be hashed. In this way, we reduce both the size of the auxiliary information σ, and the complexity of the off-chain prover for circuit $\mathsf{C}_{\mathsf{reftx}}$ (cf. Algorithm 1) that needs to prove $\mathsf{sha256}$ preimage knowledge.

4 Application: Proof of Burn

A typical approach to blockchain bridging is that of burn-and-mint. That is, to move a token from a blockchain to another, the token is burnt on the first blockchain, and minted on the second one.

Token protocols developed on top of Bitcoin are often structured as follows. The token is minted at a *genesis transaction* $\mathsf{tx}_{\mathsf{genesis}}$, and then it is transferred from one user to another according to some rules of spending. For example, the rule might be: the token is minted at the first output of $\mathsf{tx}_{\mathsf{genesis}}$, and every time a UTXO holding at token is spent in stx, the token is transferred to the first output of stx.

When tokens on Bitcoin are developed according to these framework, it is hard to employ a burn-and-mint approach. The reason is what we call the *back to genesis problem*. If a token is minted at $\mathsf{tx}_{\mathsf{genesis}}$ and exchanged among various users, how can we ensure that a purported transaction output is indeed holding the token? If the rule of transfer is the one above, we should check that the first output of the transaction tx we are being shown is linked to the first output of $\mathsf{tx}_{\mathsf{genesis}}$ via a series of spendings in which every time we spend a first output. In other words, we have to trace the first output of tx back to $\mathsf{tx}_{\mathsf{genesis}}$, and this is the back to genesis problem.

The obvious solution to this problem is to request the full chain of spending transactions linking $\mathsf{tx}_{\mathsf{genesis}}$ to tx. However, this chain gets longer the more the token is transferred. The *back to genesis problem* is concerned with proving the link between $\mathsf{tx}_{\mathsf{genesis}}$ and tx without supplying the entire chain of transactions linking them.

What we notice is that if we can solve this problem, then we can encode the solution in a smart contract on another chain, e.g., Ethereum, and deploy a burn-and-mint bridge. Indeed, when a user wants to move to Ethereum, they burn the first output of tx by spending it to an `OP_0 OP_RETURN` output. Then, they fetch the *burning transaction* $\mathsf{tx}_{\mathsf{burning}}$ (the transaction in which they burnt

the token) and show to the smart contract the Merkle proof of inclusion for it. The smart contract checks the Merkle proof of inclusion, verifies that the first input of $\mathsf{tx}_{\mathsf{burning}}$ can be traced back to $\mathsf{tx}_{\mathsf{genesis}}$, and checks that the first output of $\mathsf{tx}_{\mathsf{burning}}$ is unspendable.

A solution to the back to genesis problem was proposed in [14] using Proof Carrying Data systems [9]. In a nutshell, the idea is to produce a proof π_{i+1} every time that the token is transferred from tx_i to tx_{i+1} that attests to the validity of the following claim: "tx_{i+1} is linked to $\mathsf{tx}_{\mathsf{genesis}}$". Then, instead of providing the entire chain of transactions linking $\mathsf{tx}_{\mathsf{genesis}}$ to tx_{i+1}, it is enough to show the proof π_{i+1}.

The solution proposed in [14] is an off-chain solution. Using the techniques described in this paper, we turn it into an on-chain solution, and we leverage it to construct a locking script that attests to the burning of a token.

Fix a genesis transaction $\mathsf{tx}_{\mathsf{genesis}}$. To concretely describe the circuit $\mathsf{C}^{\mathsf{tx}_{\mathsf{genesis}},\mathsf{burn}}$ we need to recall the following definition, adapted from [14, Definition 2].

Definition 1. *A sequence of transactions* $(\mathsf{tx}_1, \ldots, \mathsf{tx}_n)$ *is a transaction chain if the first input of* tx_{i+1} *spends the first output of* tx_i *for any* $1 \leq i \leq n-1$.

Let $(\mathsf{pk}, \mathsf{vk})$ a proving/verifying key pair for the PCD solving the back to genesis problem. That is, the prover P of the PCD, on input the proving key pk, a genesis transaction $\mathsf{tx}_{\mathsf{genesis}}$, two transactions tx_{n-1}, tx_n, and a prior proof π_{n-1}, outputs a proof π_n attesting to the statement: "There exists a transaction chain $(\mathsf{tx}_{\mathsf{genesis}}, \ldots, \mathsf{tx}_n)$". The verifier V, on input $(\mathsf{vk}, \mathsf{txid}_{\mathsf{genesis}}, \mathsf{txid}_n, \pi)$, either accepts or rejects, where accept means that V believes to the statement "There exists a transaction chain $(\mathsf{tx}_{\mathsf{genesis}}, \ldots, \mathsf{tx}_n)$ such that txid_n is the transaction id of tx_n, and $\mathsf{txid}_{\mathsf{genesis}}$ is the transaction id of $\mathsf{tx}_{\mathsf{genesis}}$."

Then, the circuit $C^{\mathsf{tx}_{\mathsf{genesis}},\mathsf{burn}}$ is given by Algorithm (4).

Algorithm 4 $C^{\mathsf{tx}_{\mathsf{genesis}},\mathsf{burn}}$: Proof of burn

Public input (instance): $l_{\mathsf{stxo}} = \mathsf{txid}_{\mathsf{genesis}}$, $u_{\mathsf{stx}} = \perp$, stx.
Private input (witness): w.
Steps:

1: Parse w as a PCD proof π.
2: Set txid_n to be the txid referenced by the first input of stx.
3: Verify $\mathsf{V}(vk, \mathsf{txid}_{\mathsf{genesis}}, \mathsf{txid}_n, \pi) = 1$.
4: Verify the locking script of the first output of stx is `OP_0 OP_RETURN`.
5: If both checks pass, return 1, else 0.

5 Implementation

We implemented $\mathsf{C}_{\mathsf{reftx}}$ (1) using arkworks [10]. More precisely, we implemented R1CS versions of Bitcoin structures and operations, and used

arkworks to construct the integrity mechanism TI and the circuit $\mathsf{C}_{\mathsf{reftx}}$. The code is avaiable at `https://github.com/nchain-innovation/bitcoin_r1cs`. The implementation of TI generates 439046 constraints when implemented with sighash_all for a transaction with one input and two [P2PKH] outputs. We also implemented $C^{\mathsf{tx}_{\mathsf{genesis}},\mathsf{burn}}$ (4), whose code is avaiable at `https://github.com/nchain-innovation/bitcoin_r1cs/blob/ark-3.0/src/bitcoin_predicates/proof_of_burn.rs`.

References

1. Barbacovi, F., Larraia, E., Germouty, P., Zhang, W.: Pratical implementation of pairing based zksnark in bitcoin script. In: to appear in Computer Security. ESORICS 2024 International Workshops. Springer Cham (2025)
2. Bitcoin SV: `https://github.com/bitcoin-sv/bitcoin-sv`
3. Bitcoin SV: Message Digest Algorithm. `https://github.com/bitcoin-sv/bitcoin-sv/blob/master/doc/abc/replay-protected-sighash.md`
4. Bitcoin SV Wiki: Bitcoin Script. `https://wiki.bitcoinsv.io/index.php/Script`
5. Bitcoin SV Wiki: Bitcoin Script OP_CODES. `https://wiki.bitcoinsv.io/index.php/Opcodes_used_in_Bitcoin_Script`
6. Bitcoin SV Wiki: nLocktime. `https://wiki.bitcoinsv.io/index.php/Bitcoin_Transactions`
7. Bitcoin SV Wiki: SIGHASH flags. `https://wiki.bitcoinsv.io/index.php/SIGHASH_flags`
8. Bitcoin Wiki: OP_CHECKSIG. `https://en.bitcoin.it/wiki/OP_CHECKSIG`
9. Chiesa, A., Tromer, E.: Proof-carrying data and hearsay arguments from signature cards. In: Proceedings of the 1st Symposium on Innovations in Computer Science. p. 310–331 (2010)
10. arkworks contributors: `arkworks` zksnark ecosystem (2022), `https://arkworks.rs`
11. Federico Barbacovi: `OP_CHECKSIG` beyond signature validation: covenants, transaction introspection and PUSHTX. `https://hackmd.io/@federicobarbacovi/By6zkFmfyl`
12. Groth, J.: On the size of pairing-based non-interactive arguments. In: Advances in Cryptology - EUROCRYPT (2016)
13. Heilman, E., Kolobov, V.I., Levy, A.M., Poelstra, A.: ColliderScript: Covenants in bitcoin via 160-bit hash collisions. Cryptology ePrint Archive, Paper 2024/1802 (2024), `https://eprint.iacr.org/2024/1802`
14. Kiraz, M.S., Larraia, E., Vaughan, O.: Nft trades in bitcoin with off-chain receipts. In: Applied Cryptography and Network Security Workshops. pp. 100–117. Springer Nature Switzerland, Cham (2023)
15. Kolobov, V.I., Levy, A.M., Naor, M.: ColliderVM: Stateful computation on bitcoin without fraud proofs. Cryptology ePrint Archive, Paper 2025/591 (2025), `https://eprint.iacr.org/2025/591`
16. Linus, Robin and Aumayr, Lukas and Zamyatin, Alexei and Pelosi, Andrea and Avarikioti, Zeta and Maffei, Matteo: BitVM2: Bridging Bitcoin to Second Layers. `https://bitvm.org/bitvm_bridge.pdf`
17. Poelstra, Andrew: CAT and Schnorr Tricks I. `https://medium.com/blockstream/cat-and-schnorr-tricks-i-faf1b59bd298`
18. S. Nakamoto: Bitcoin: A Peer-to-Peer Electronic Cash System
19. Zhang, W.: PUSHTX and its building block. `https://nchain.com/wp-content/uploads/2022/03/WP1605_PUSHTX-and-its-Building-Blocks.pdf`