ColliderScript: Covenants in Bitcoin via 160-bit hash collisions

Ethan Heilman[1], Victor I. Kolobov[2], Avihu M. Levy[2], and Andrew Poelstra[3]

[1]ethan.r.heilman@gmail.com
[2]StarkWare, {victor.k,avihu}@starkware.co
[3]Blockstream, apoelstra@blockstream.com

November 8, 2024

Abstract

We introduce a method for enforcing covenants on Bitcoin outputs without requiring any changes to Bitcoin by designing a hash collision based *equivalence check* which bridges Bitcoin's limited *Big Script* to Bitcoin's *Small Script*. This allows us evaluate the signature of the spending transaction (available only to Big Script) in Small Script. As Small Script enables arbitrary computations, we can introspect into the spending transaction and enforce covenants on it.

Our approach leverages *finding collisions* in the 160-bit hash functions: SHA1 and RIPEMD. By the birthday bound this should cost $\sim 2^{80}$ work. Each spend of our covenant costs $\sim 2^{86}$ hash queries and $\sim 2^{56}$ bytes of space. For security, we rely on an assumption regarding the hardness of finding a 3-way collision (with short inputs) in 160-bit hash functions, arguing that if the assumption holds, breaking covenant enforcement requires $\sim 2^{110}$ hash queries. To put this in perspective, the work to spend our covenant is ~ 33 hours of the Bitcoin mining network, whereas breaking our covenant requires $\sim 450,000$ years of the Bitcoin mining network. We believe there are multiple directions of future work that can significantly improve these numbers.

Evaluating covenants and our equivalence check requires performing many operations in Small Script, which must take no more than 4 megabytes in total size, as Bitcoin does not allow transactions greater than 4 megabytes. We only provide rough estimates of the transaction size because, as of this writing, no Small Script implementations of the hash functions required, SHA1 and RIPEMD, have been written.

1 Introduction

Since its inception, Bitcoin has contained a scripting language used to express conditions under which coins are spent. In 2010, in response to multiple bug reports, Satoshi removed several opcodes from the language [43, 42]. While it was not realized at the time, this removal split the language into two parts: "Big Script," a set of opcodes used to manipulate signatures, hashes and other cryptographic objects; and "Small Script," a set of opcodes which do arbitrary computations but only on 32-bit inputs.

This bifurcation is important because Big Script's `OP_CHECKSIG` opcode, which verifies a signature on the full transaction data, is the only opcode capable of full *transaction introspection*. By transaction introspection, we mean the ability of a Bitcoin script to examine the bytes of transaction that is attempting to spend that output, *i.e.,* the spending transaction. The "Schnorr trick" [35, 36] provides a way to structure a signature provided to `OP_CHECKSIG` such that we can extract the hash of the spending transaction (*aka,* the sighash) from the signature.

Big Script is not expressive enough to enforce arbitrary spending conditions on the bytes of the spending transaction using the sighash (hash of the spending transaction). Small Script is expressible enough to enforce these conditions, but because `OP_CHECKSIG` requires a signature much larger than 32 bits, Small Script cannot perform operations on such signatures.

Scripts which can introspect into the data of the spending transaction are termed *covenants* [32], and have been a recent major source of controversy and research in Bitcoin [6, 47, 46, 22, 40]. Covenants, if possible in Bitcoin, would enable new functionality such as rate-limited wallets and vaults [33] and more efficient layer-two protocols [1].

The most direct way to enable covenants would be to extend the Bitcoin Script language to include transaction introspection opcodes, which directly copy transaction data onto the stack to be processed by other opcodes. A less direct way would be to heal the split between Big Script and Small script, e.g. by enabling the `OP_CAT` concatenation opcode. However, any changes to Bitcoin must have consensus across all economic stakeholders, and because of the controversy around covenants in particular, consensus on any changes may not be achieved quickly.

In this paper, we propose *ColliderScript*, a novel way to bridge Big Script and Small Script, allowing transaction signatures to be manipulated in such a way that any signed transaction data can be extracted. Our key idea is exploit SHA1 and RIPEMD[1] collisions to show that a signature encoded as a vector of 32-bit Small Script elements and a Big Script signature represent the same signature. We refer to this as an equivalence check. The key idea behind this equivalence check is that $\mathtt{H}(s_1) = \mathtt{H}(\pi) = \mathtt{H}(s_2)$ where $\pi \neq s_1$ implies that $s_1 = s_2$ even if you can't directly compare s_1 and s_2. For 160-bit hash functions, finding single collisions to perform our equivalence check is practical but finding the triple collisions needed to break out equivalence check is not practical. Using this idea we construct a *small-to-big equivalence check* showing that a Big Script object is the same as a Small Script object. Our parameters are not exact, as our construction depends on the Small Script implementation of SHA1 and RIPEMD, which, as of this writing, hasn't been written yet.

In particular, we show the following

Proposition 1 (Informal)**.** *Under some assumptions on SHA1 and RIPEMD, there is a Bitcoin locking script capable of small-to-big equivalence check, such that*

- *The script uses less than* 4 *million weight units (bytes)* i.e., *small enough to be a valid transaction.*
- *To prove small-to-big equivalence check, the spender needs to provide the locking script with input which requires computing* $\sim 2^{86}$ *hash queries, and using* $\sim 2^{56}$ *bytes of memory.*
- *An adversary needs to perform significantly more work to fool the equivalence check, roughly* 2^{110} *queries.*

In addition, this construction requires a globally one-time *precomputation of* $2^{83} - 2^{93}$ *hashes. Moreover, a covenant locking script exists with roughly the same parameters (see Figure 8)*

Throughout this paper when we talk about Bitcoin script we mean Bitcoin Tapscript (`BIP-342` [53]). Our scheme only works for Tapscript and can not be used for pre-Tapscript Bitcoin script. One of reasons it does not work on pre-tapscript is that we rely on Schnorr signatures to recover bytes of the spending transaction from the signature and Schnorr signatures are not available in pre-Tapscript.

We treat SHA1 and RIPEMD as ideal 160-bit hash functions despite collision attacks on SHA1 [45, 29] that are well below the 2^{80} cost of a generic attack on an ideal 160-bit function. We discuss in Section 7.1 why we consider this a reasonable assumption to make.

Our paper is organized as follows. In Section 1.1 we provide an outline of how our technique works. We follow this with a comparison to related work in Section 1.2. Section 2 defines notation and gives an overview of Bitcoin and Bitcoin scripts, and transaction introspection. In Section 3, we present definitions for covenants and equivalence checks in Bitcoin. Then we introduce Bitcoin equivalence tester sets (Section 4), our equivalence check construction (Section 5) and propose parameters to instantiate our construction with (Section 6). Section 7 discusses the security of our scheme. Finally we conclude and discuss future directions (Section 8)

1.1 Outline of techniques

Here we provide an algorithm description for checking the equivalence of a signature in "Big Script" to the same signature in "Small Script" and how it relates to our covenant construction. We emphasize that the full fledged covenant construction is more complicated than what we present in this section but it follows the same template.

Our covenant has the following template:

- The transaction, tx, attempting to spend the covenant provides:

[1] Throughout, RIPEMD refers to `RIPEMD-160`.

- s_1, a signature on the spending transaction[2];
- $\langle s_2 \rangle_{32}$, a small-script representation of s_1. That is, we chop s_2 into 32-bit chunks, which is the largest element size that Small Script can manipulate;
- $\pi = (\omega, t)$, the equivalence witness used to show s_1 is equivalent to $\langle s_2 \rangle_{32}$ *i.e.,* $s_1 = s_2$.

- The covenant being spent:
  - Verifies s_1 represents a valid signature of the spending transaction, by running `OP_CHECKSIG`
  - Applies our equivalence check using the equivalence witness $\pi = (\omega, t)$ to verify that indeed $\langle s_2 \rangle_{32}$ is a small-script representation of s_1.
  - Knowing that $\langle s_2 \rangle_{32}$ is a small-script representation of the valid signature on the spending transaction, uses auxiliary information, TxData, to extract the actual data of the spending transaction, tx.
  - Check conditions on the tx, and accept/reject accordingly

Big Script and Small Script have different limitations with respect to the operations they can perform. In Big Script we can do very few operations: hashing, checking equality, and in some cases treating elements as public keys or signatures to perform signature verification. The latter being accomplished by the `OP_CHECKSIG` opcode, which, importantly, *is the only way to access data related to the spending transaction in Bitcoin script.* In Small Script we don't have access to the spending transaction data but we can compute arbitrary logic (though due to the limited number of available opcodes, this comes at a cost). For complex operations such as implementing cryptographic hash functions, this severely limits the number of Small Script hash function calls a single script can make.

With the above limitations in mind, we define a special set $\mathcal{D}$, indexed by $\pi = (\omega, t)$, where ω is a *Small Script element* to be hashed iteratively, alternately using RIPEMD or SHA1 according to the bits of a binary vector $t = (t_0, \ldots, t_{||t||-1})$. In pseudo code, to generate $d_{\omega,t} \in \mathcal{D}$ from (ω, t) we perform the following algorithm, to which we refer as $\mathcal{D}.\mathsf{Gen}$:

1. Let $\mathsf{res} \leftarrow \omega$
2. For $i = 0, \ldots, (||t|| - 1)$:
   - $\mathsf{res} \leftarrow \begin{cases} \text{SHA1}(\mathsf{res}), & t_i = 0 \\ \text{RIPEMD}(\mathsf{res}), & t_i = 1 \end{cases}$
3. Return $d_{\omega,t} \leftarrow \mathsf{res}$

Crucially, as ω is readable in Small Script, this algorithm can be implemented both in Big Script (using `OP_RIPEMD160` and `OP_SHA1`) and in Small Script (using explicit implementations of these hash functions)[3]. For ease of collisions we need a hash function that can take a input of roughly 100-bits.

An illustration of how $\mathcal{D}.\mathsf{Gen}$ is computed is given in Figure 1 (Right). To prove s_1 (an element of Big Script) and s_2 (an array of Big Script elements) are equivalent using the set $\mathcal{D}$, the core idea is to find $d \in \mathcal{D}$ such that $\text{SHA1}(s_1) = d = \text{SHA1}(s_2)$. Here the first equality can be verified in Big Script and the second equality in Small Script. We claim that this is sufficient to imply that $s_1 = s_2$.

Whenever $s_1 = s_2$ then clearly $\text{SHA1}(s_1) = \text{SHA1}(s_2)$, and if we assume $\mathcal{D}$ is large enough and that we have freedom to vary s_1, finding an appropriate d is feasible, as it reduces to finding a single collision. To argue soundness, suppose that $s_1 \neq s_2$. That means that an attacker wishing to prove false equivalence will need to find $\text{SHA1}(s_1) = \text{SHA1}(s_2) = d$ with s_1, s_2, d distinct. Soundness follows from the assumption that finding a triple collision (with short inputs) is hard.

Next, we outline how a (regular) collision in SHA1 can be found efficiently. Note that the size of SHA1 digest is 160 bit. Hence one straightforward way to find collisions is to precompute and store a subset of $\mathcal{D}$ of size 2^{80}, and then perform a search on the set of signatures s to find a collision. This requires (on average) $\sim 2^{80}$ work and $\sim 2^{80}$ bits of memory, the latter making it infeasible.

[2]We don't use the security properties of the signature here; we use it only to allow Bitcoin script to access the spending transaction data, as the message being signed is a hash of the spending transaction data (also known as a *sighash*).

[3]We utilize hash functions as these are the *only opcodes capable of mutating an element in Big Script.* Furthermore, we need their explicit implementations in Small Script. Hence, we restrict ourselves to only SHA1 and RIPEMD, as we expect them to have a reasonable Small Script implementation cost.

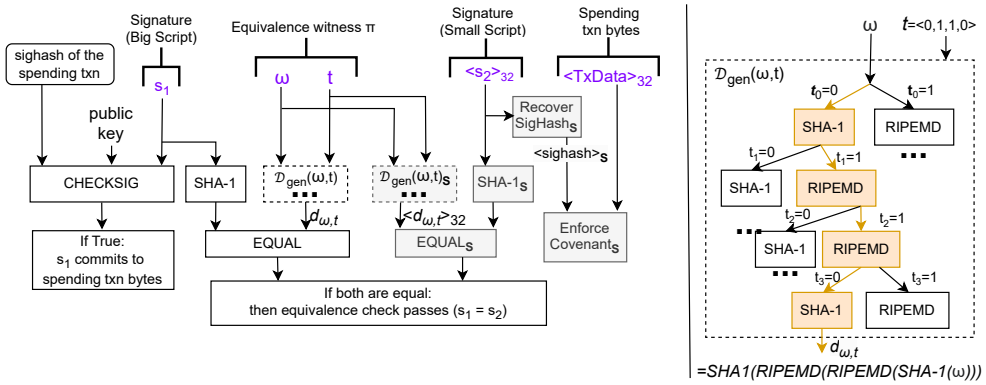

Figure 1: (Left) Visualization of the Tapscript of our equivalence check based covenant. Functions denoted with subscript **S** and grey background are evaluated in small script, *i.e.,* built out of Small Script opcodes such as `OP_ADD` that operate on Small Script elements (0 to 33-bit values). Elements pushed on the stack from the spending transaction are colored purple.
(Right) Our function $\mathcal{D}.\mathsf{Gen}$ which recursively hashes a seed ω using either SHA1 or RIPEMD based on the bitstring. For brevity t we use a t of length 4 of bits. In practice t is 70 bits. The path executed through $\mathcal{D}.\mathsf{Gen}$ when $t = \langle 0, 1, 1, 0\rangle$ is highlighted in orange.

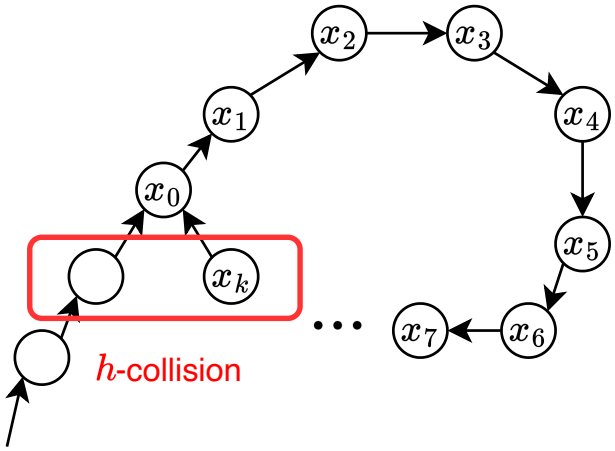

Figure 2: A cycle where each arrow is an evaluation of h, which by the birthday bound, is expected to be of length $\sim 2^{80}$. Since the two values in red evaluate to x_0, they constitute a collision for h. With probability $1/2$ it will be a collision between the set $\mathcal{D}$ and the signatures of the spending transaction.

To significantly reduce the memory requirement define a function h, that given some input, outputs either $\mathrm{SHA1}(s)$, where s is a valid signature on the spending transaction, or some $d \in \mathcal{D}$. More concretely,

$$h(x) = \begin{cases} \mathrm{SHA1}(s(x)), & \text{first bit of } x \text{ is } 0, \\ \mathcal{D}.\mathsf{Gen}(\omega(x), t(x)), & \text{first bit of } x \text{ is } 1, \end{cases}$$

where there is some deterministic way to derive $s = s(x)$, $\omega = \omega(x)$, and $t = t(x)$ from x. Then, the goal is to find a cycle in the repeated application of h, namely, values

$$x_0, x_1 = h(x_0), x_2 = h(x_1), \ldots, x_{k+1} = h(x_k),$$

such that $x_{k+1} = x_0$. This will, with a probability of $1/2$, result in two inputs, s and $d = \mathcal{D}.\mathsf{Gen}(\omega(x), t(x))$, such that $\mathrm{SHA1}(s) = d$ (for example, $x_0 = \mathrm{SHA1}(s)$ and $x_{k+1} = d$). Figure 2 illustrates collision detection using a cycle. Relying on known cycle search algorithms [23, 49] it yields, for $\mathcal{D}$ sufficiently large, an algorithm with $\sim 2^{80}$ running time complexity and a manageable memory cost. Nevertheless, analyzing the best concrete parameters for this construction is the main challenge of this work.

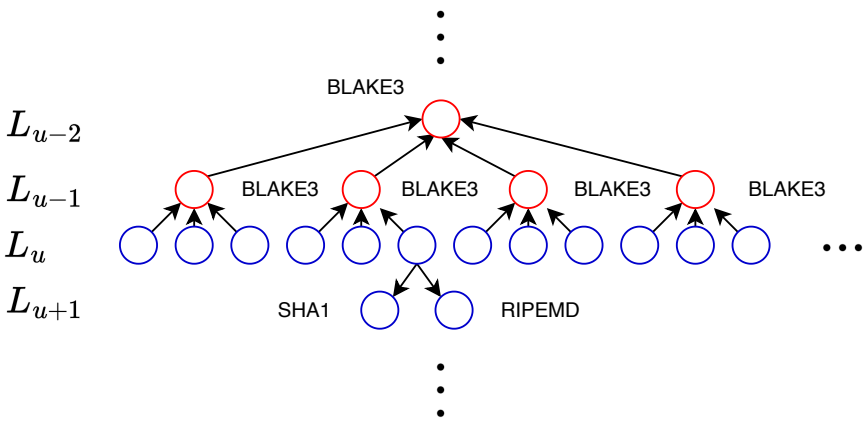

Figure 3: A Small Script cost saving technique for generating $\mathcal{D}$'s elements. Instead of computing all $\log_2 |\mathcal{D}|$ layers in Small Script, the first u layers are committed to a BLAKE3 (*truncated to* 128-*bit hashes*) Merkle tree. To generate an element in this new approach, a value from $\mathcal{D}$'s binary tree's u-th layer, L_u, is decommitted from the Merkle tree, followed by $\log_2 |\mathcal{D}| - u$ layers computed as before, using SHA1 and RIPEMD. Blue circles are 160-bit, while red circles are 128-bit. Because BLAKE3's input size is 512-bit, we can pack three 160-bit values (leaf) or four 128-bit values (inner node) together.

To summarize, our covenant, as shown in Figure 1 (Left), proceeds as follows, given input s_1 (in Big Script), s_2 (in Small Script), $(\omega, t = (t_0, \ldots, t_{||t||-1}))$ (in Small Script), and some auxiliary data:

1. In Big Script, verify that s_1 represents a valid signature of the spending transaction, tx
2. In Big Script, compute $d_{\omega,t}$ and check that $\text{SHA1}(s_1) = d_{\omega,t}$
3. In Small Script, compute $d_{\omega,t}$ and check that $\text{SHA1}(s_2) = d_{\omega,t}$
4. In Small Script, use the auxiliary information to extract from s_2 the transaction data, tx
5. In Small Script, check conditions on the data tx, and accept/reject accordingly

To spend the above covenant, a spender will need to run an off-chain algorithm to find a collision between the signatures of the spending transaction and $\mathcal{D}$'s elements. This algorithm will produce a valid s_1 and (ω, t) for the spender to use.

Reducing Small Script costs. Our equivalence check requires the implementation of SHA1 and RIPEMD in Small Script. As of writing, no one has published source code for Small Script implementations of these hash functions. Even though we expect their implementation to be somewhat efficient, their size could still end up being prohibitive. To mitigate this, we propose to precompute a subtree of the binary tree generating the elements of $\mathcal{D}$, committing it to a Merkle tree using *any* collision resistant hash function which admits an efficient Small Script implementation[4].

More concretely, we propose to commit the first u layers of the binary tree of $\mathcal{D}$ into a Merkle tree root which is *hardcoded* into the covenant script. Then, generating an element of $\mathcal{D}$ will require decommiting the u-th layer from the Merkle tree, followed by $\log_2 |\mathcal{D}| - u$ computations of either SHA1 or RIPEMD in Small Script. An illustration of our approach is given in Figure 3.

While the above technique requires a *globally one time* precomputation of size 2^u, the savings in Small Script cost are significant. Computing each layer of the binary tree of $\mathcal{D}$ incurs the cost of *both* SHA1 and RIPEMD, due to how Bitcoin script length is measured. Here, for the first u layers, we instead reduce the cost per layer to a *single* hash function computation. To instantiate our scheme, we choose BLAKE3, which has an efficient Small Script implementation [5] of $\sim$ 45K opcodes.

Finally, we use a *quaternary* Merkle tree (radix 4), as opposed to the binary tree used to generate the elements of $\mathcal{D}$, thus obtaining further savings, as even fewer BLAKE3 evaluations are then required.

[4]This change applies only to the Small Script generation of $\mathcal{D}$'s elements. In Big Script, we still use `OP_SHA1` and `OP_RIPEMD160` to generate $\mathcal{D}$'s elements.

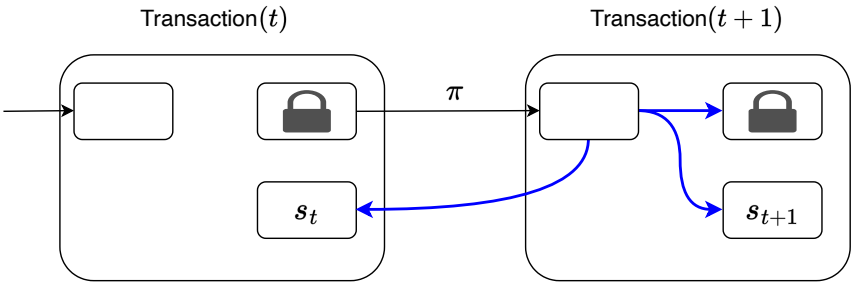

Figure 4: Sufficiently powerful covenants, such as the ones enabled through `OP_CAT`, lend themselves to general smart contract designs on Bitcoin [24]. In this example, a locking script (visualized as a lock) can enforce that: (i) it can only be spent by a transaction with the same format and locking script – providing logic persistence on-chain; and (ii) the state transition from s_t to s_{t+1}, was attested by π to be a valid state transition – providing state or data persistence on-chain. Our covenant also enables this kind of simple smart contract design.

1.2 Related Work

In the following, we briefly survey related work on Bitcoin covenants, including research related to `OP_CAT` and BitVM. In addition, we mention research into hash functions, together with work on their Small Script implementation, which is relevant to our work.

Covenants. Covenants were first proposed on the Bitcoin-talk forum in 2013 [32]. [33] provided a comprehensive treatment of covenants in an academic paper and launched a long line of research [6, 47, 46, 22]. There have been several proposed upgrades to Bitcoin that would introduce covenants including: `OP_PUSHTXDATA` [27], `OP_CTV` (BIP-119) [40], `OP_VAULT` (BIP-345) [34], and `OP_TXHASH` and `OP_CHECKTXHASHVERIFY` [38].

A less direct path to having covenants on Bitcoin involves re-enabling previously disabled Bitcoin opcodes, which *indirectly* yield covenants. An example of this is `OP_CAT` [18] which using the *Schnorr trick* [35, 36] allows for covenants. Another example is the "Great Script Restoration" (GSR) [41]. GSR calls to re-enable `OP_CAT` together with other useful, previously disabled, opcodes such as `OP_MUL`, enabling not only covenants but also a more expressive on-chain logic for Bitcoin.

OP_CAT. Research into the power of Bitcoin script with `OP_CAT` has been a very fruitful direction. It has been demonstrated that `OP_CAT` is sufficiently powerful to almost arbitrarily constrain the spending transaction. In particular, this technique is sufficiently powerful to force having the *same locking script* and *retaining state* across multiple transactions. `OP_CAT`-based stateful computation, as seen in Figure 4, has been successfully implemented and referred to by multiple names, such as "state caboose" [8], "recursive covenants" [44], and "finite state machine" [48], among others.

Moreover, `OP_CAT` in particular has been identified as very advantageous for *scaling* computation on Bitcoin, via STARK-based proof systems, such as [16], which lend themselves to an efficient verifier implementation in an `OP_CAT`-enabled Bitcoin Script [3].

The BitVM paradigm. This is a proposal by Linus [30] to obtain stateful computation on Bitcoin without requiring support for "true covenants" by instead utilizing presigned transactions. This technique achieves stateful computation in the following way:

- **Data persistence** - This is achieved via *Lamport signatures*, through an anti-equivocation mechanism. Essentially, the operator commits to partial values of some encoding of the program execution through Lamport signatures. These partial values are effectively stored in a *global key-value store*, which provides a form of persistent data storage, as these values can be accessed from any Bitcoin script (or *any other blockchain*, for that matter). Because of the equivocation mechanism, it is a fraud proof-based system, with some "slashing" enacted on the operator in case of equivocation to disincentivise fraudulent execution.

- **Logic persistence** - This is achieved through the predefined validators presigning multiple transactions which follow a certain template. Then, during runtime, the operators commit to the inputs and partial values of computation. Hence, participants are guaranteed that the others have no way to spend the locked funds in a way that can break the intended logical flow. As pointed out by [28], this is related to the concept of *connectors* introduced in the Ark protocol [1].

  The requirement for all n validators to sign template transactions is the source of the 1*-out-of-*n *honesty assumption in the BitVM protocol.*

BitVM was originally presented in a two-party setting with an operator and a verifier but has since been extended into multiple protocol variants with multiple predefined operators and multiple verifiers. Unlike the original BitVM proposal which uses NAND-gate circuits, BitVM1 and BitVMX [28] work by fraud proving on-chain the execution of a *virtual CPU* to m predefined verifiers. In contrast, BitVM2 [31] uses a different design, and allows *any onlooker* to act permissionlessly as a verifier in case of fraudulent execution, which yields an improved security guarantee. Unlike BitVM1 and BitVMX, BitVM2 requires a different protocol for each circuit, and thus BitVM2 directly considers fraud proving the execution of a proof system verifier, which can remain the same while executing different (provable) programs. Hence, BitVM1 and BitVMX provide potentially cheaper operator off-chain costs via direct CPU execution, while BitVM2 yields a stronger security guarantee.

BitVM2 in particular has given rise to multiple new scaling solutions for Bitcoin. For a comprehensive list of Bitcoin layer 2s, consult the Bitcoin Layers website [2]. For an overview and background on BitVM and its variants, we refer to [37, 28] and references therein.

Covenants without a soft fork. There are proposals to bring covenants into Bitcoin without a soft fork, such as [39, 25]. However, these require heavy cryptographic tools such as Functional Encryption [7]. Our work only involves finding collisions in 160-bit hash functions.

Hash functions and collisions. Both for our construction and security analysis, we rely on analyzing the effectiveness of finding collisions in hash functions. For our covenant construction, we use a parallel collision finding algorithm based on distinguished points, in similar vein to [49]. Our algorithm exhibits a time-memory trade-off [19], improving upon which is an important future research direction.

For our security, we rely on hardness of finding a 3-way collision in ideal hash functions. This is in spite of SHA1 having nontrivial collision attacks [45, 29], and, in general, finding 3-way collision sometimes being as easy as finding a regular collision [21]. Nevertheless, the adversary is required to supply short inputs in our setting, so we believe our assumptions to hold.

The most expensive aspect of our construction is the $\sim 2^{80}$ off-chain compute cost. We expect possible improvements can come from two directions:

- Non-trivial collision attacks on SHA1 (or RIPEMD), which will lower our covenant off-chain cost (while still keeping finding 3-way collisions infeasible).
- Precomputation methods to reduce the off-chain per-spend cost of the covenant[5]. This is not trivial because we find collisions of the form $f(\pi) = g_{tx}(\rho)$, where f is always the same hash function, while g_{tx} depends on the transaction data (hence only known during runtime).

Finally, a major component in our scheme is the computation of Merkle trees. While we use the standard construction, the improvements shown in [13, 12] could be useful to gain more efficiency in our setting.

Small Script implementation. We base our construction on the implementation of hash functions in Small Script. The BitVM project [4] has compiled a large library of nontrivial Small Script computations, including BLAKE3 and SHA256 [20, 14]. Our construction also relies on implementations for SHA1 and RIPEMD, which, as the writing of this paper, don't exist yet.

[9, 10] design a small integer multiplication system for Bitcoin's Small Script, while [54] design a large integer multiplication system.

Another possibly relevant work in this domain is that of evaluating Lamport signatures in pre-taproot scripts [17].

[5] In our construction, we only use precomputation to reduce the Small Script cost of our covenant. Performing precomputation to also reduce off-chain compute cost can be very useful.

2 Preliminaries

2.1 Notation

Elliptic curve arithmetic. We'll use capital Latin letters to denote points on Bitcoin's elliptic curve, `secp256k1`. We'll also denote by aB the multiplication of a `secp256k1` point B by a scalar a

Bitwise operations. $\|x\|$ denotes the size of a value in bits. We use the following notation for bitwise manipulation: By $x\,|_a$, we denote the bit at the index a in x. Extending this notation to ranges, $x\,|_a^b$ denote the bit string in x that starts at position a and continues to position b (inclusive). To represent the concatenation of x to y we write $x\|y$.

Bitcoin script. We denote by $\mathcal{S} = \bigcup_{i=0}^{32}\{0,1\}^i$ the set of Small Script elements, and by $\mathcal{B} = \bigcup_{i=0}^{4160}\{0,1\}^i$ the set of Big Script elements. Observe that $\mathcal{S} \subset \mathcal{B}$. We'll usually use lowercase Greek letters to for elements of $\mathcal{S}$, such as $\sigma \in \mathcal{S}$, and lowercase Latin letters for elements of $\mathcal{B}$ (which may also lie in $\mathcal{S}$). For elements $v \in \mathcal{B}$, we'll denote their encoding as arrays of k-bit elements by $\langle v\rangle_k$, where common choices will be $k = 1$ (boolean array) and $k = 32$ (so $\langle v\rangle_{32} \in \mathcal{S}^*$). We refer to $\langle v\rangle_{32}$ as the *small-script representation of* v.

We refer to the set of Script opcodes whose allowable inputs all lie in $\mathcal{S}$ as "Small Script opcodes", and all others as "Big Script opcodes". In addition, we'll denote by $\text{Func}_\mathcal{S} = \{f_\mathcal{S} : \mathcal{S}^* \to \mathcal{S}^*\}$ the set of functions implementable using exclusively Small Script opcodes, and by $\text{Func}_\mathcal{B} = \{f_\mathcal{B} : \mathcal{B}^* \to \mathcal{B}^*\}$ the set of functions implementable using exclusively Big Script opcodes[6]. In addition, we'll use $\mathcal{X} = \{0,1\}^*$ to denote the domain of functions implementable by off-chain algorithms (for instance, on an x86 processor).

Furthermore, we will abuse notation, and not distinguish between $f_\mathcal{S}$ and the Small Script implementing it, with the right notion being clear from context. Similarly, for $g_\mathcal{B}$, we'll associate it with its Big Script implementation. Moreover, we'll write $\|f_\mathcal{S}\|$ to refer to the length of the script.

Since Small Script can express any computation, in particular for any $f_\mathcal{B} \in \text{Func}_\mathcal{B}$ there is $f_\mathcal{S} \in \text{Func}_\mathcal{S}$ such that *for every* $b \in \mathcal{B}$ it holds that $\langle f_\mathcal{B}(b)\rangle_{32} = f_\mathcal{S}(\langle b\rangle_{32})$. We refer to $f_\mathcal{S}$ as the *small-script equivalent* of $f_\mathcal{B}$[7]. If it were possible in Script to demonstrate equivalence of an element $b \in \mathcal{B}$ and $\langle b\rangle_{32} \in \mathcal{S}^*$, this would allow us to do arbitrary computations on elements b of Big Script, by using Small Script opcodes on $\langle b\rangle_{32}$.

Algorithm notation. Our algorithms may run in *four* different settings. To distinguish between these settings, we attach a subscript to the algorithm name. If the algorithm runs off-chain, we simply denote it as $\mathsf{Algorithm}$, without subscript. When an algorithm is specifically intended to run in Big Script, we attach a subscript to its name as $\mathsf{Algorithm}_\mathcal{B}$. Similarly, if it is intended to run in Small Script, we denote it as $\mathsf{Algorithm}_\mathcal{S}$. Finally, we'll use $\mathsf{Algorithms}_{\mathcal{S},\mathcal{B}}$ to denote algorithms implementable in Bitcoin script, using all available opcodes.

For algorithms with binary output $\{0,1\}$, we'll sometimes write "success" or "accept" when referring to the value 1, and, similarly, we'll replace 0 by "fail" or "reject".

2.2 Bitcoin overview

2.2.1 UTXO model

The Bitcoin blockchain consists of a series of *blocks*, each of which are a series of *transactions*, each of which atomically creates and destroys a set of *unspent transaction outputs* (UTXOs). UTXOs consist of a *scriptPubKey* (also known as a *locking script*), which is described below, and an integer amount measured in *satoshis*, which are 10^{-8} BTC. The UTXOs that a transaction creates are called (transaction) *outputs* while those that it destroys are called (transaction) *inputs*. Inputs refer to the outputs of previous transactions, identifying them by the *transaction identifier (TXID)* and an index within the list of that transaction's outputs. To be accepted by the network, the total value

[6] In Section 2.2.3 we give explicit definitions of Small Script and Big Script opcodes, and we provide a full list of all opcodes in Appendix A. Also, while we defined $\text{Func}_\mathcal{S}$ and $\text{Func}_\mathcal{B}$ as function families implementable via a *subset* of available opcodes, an actual Bitcoin script could compute both.

[7] In fact, *any* function $f : \mathcal{X} \to \{0,1\}^*$ has a small-script equivalent $f_\mathcal{S}$.

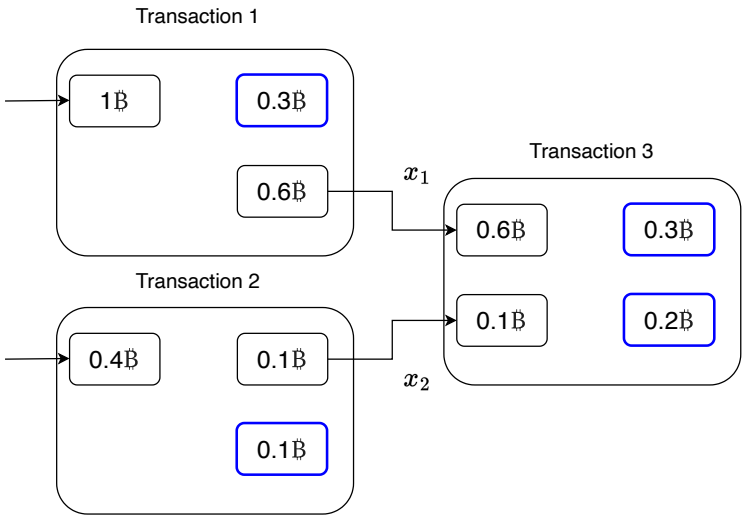

Figure 5: Transactions in the UTXO model. Transaction 3 spends UTXOs from Transactions 1 and 2 using witnesses x_1 and x_2, respectively, given to their respective locking scripts (scriptPubKey). The UTXOs after the creation of Transaction 3 are highlighted in blue.

of a transaction's outputs must be less than or equal to the total value of its inputs. The difference between the two is the *network fee*, which is effectively transferred to the creator of the block which includes the transaction.

There is one special kind of input called a *coinbase input*, which is allowed to exist in the first transaction of a block. The coinbase input implicitly has a value equal to the total of the network fee of all transactions in the block, plus a *block subsidy*, currently equal to 3.125 BTC, which is how coins are initially introduced into the network. If the first transaction in the block itself has a network fee, i.e. the block creator does not claim the total available amount, the excess is destroyed.

This model of transactions, in which UTXOs are atomically created and destroyed by transactions, is termed the *UTXO model*. The state of the Bitcoin network consists entirely of its the set of UTXOs, with the blockchain serving as a transcript of the transactions that led to this state. An illustration of the UTXO model is given in Figure 5.

It differs from the *account model*, in which coins are distributed among labeled *accounts* with variable balances. In the account model, transactions increase and decrease accounts. Individual transactions typically debit a single account and may credit multiple accounts. Here the state of the network consists of the total balance on all accounts. The most notable example of the account model is the Ethereum network, whose state additionally includes a global key-value store used to implement a "smart contracting" system.

Unlike the account model, in which transactions may be arbitrarily reordered prior to an inclusion in a block (subject to the constraint that no account balance goes below zero), in the UTXO model transactions that manipulate "the same coins" have a strict ordering. Transactions which spend UTXOs must come after the transactions which created them, and because UTXOs are labelled by TXIDs, any change to an earlier transaction in the chain will invalidate a later transaction. The UTXO model is simpler and more efficient to implement, and makes atomicity easier to reason about, while generally being more complex to use for multi-transaction protocols.

2.2.2 Locking scripts

Above we described a UTXO as a pair of an amount and a *scriptPubKey*, or "script public key". A scriptPubKey is a locking script for a UTXO. Every input in a transaction must provide a *witness* for the scriptPubKey in its UTXO, which consists of an initial stack of input data for which the script accepts. The script is defined in an opcode language called *Bitcoin Script* and is evaluated by a script interpreter implemented in every validating node on the network. Formally, we write

$$\mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}(w;\mathrm{tx}) \to \{0,1\}$$

where w represents the witness and tx the *spending transaction* which spends the UTXO. Technically, the interpreter may abort rather than returning either value, but our purposes an abort is equivalent to returning the value 0 (in both cases the spending transaction is disallowed by the network). We will therefore ignore this possibility to keep our notation simple.

Bitcoin Script is a stack-based language which is conceptually similar to Forth, and shares many of its opcodes, while being more limited. In particular, Script has no looping facilities, hence its running time is proportional to its length. It also has limited facilities for doing general computation.

Additionally, there are several resource limits which apply to Bitcoin Script: the stack may not exceed 1000 elements at any point during execution; no stack element may exceed 520 bytes in size; and the transaction itself (including all scripts) may not exceed 4 million "weight units", where each weight unit roughly corresponds to one opcode or one byte of explicit data.

The limitation of Script which is most salient to our purposes is that Bitcoin Script has very little access to the transaction context. In Big Script, which contains only a small set of cryptographic operations, we can validate digital signatures on transaction data; in Small Script we can do arbitrary computations, but only with objects of size at most 4 bytes, and with no access to transaction data.

2.2.3 Big Script and Small Script

In this section, we overview the capabilities of Big Script versus Small Script. Explicitly, Big Script, which can implement any function in $\mathrm{Func}_{\mathcal{B}}$, contains opcodes for

- Checking signatures on transaction data: `OP_CHECKSIG` and its variants `OP_CHECKSIGVERIFY`, `OP_CHECKSIGADD` (in Taproot) and `OP_CHECKMULTISIG` (pre-Taproot).
- Computing SHA1, SHA256 and RIPEMD hashes of arbitrary data, and variants HASH160 (SHA256 followed by RIPEMD) and HASH256 (SHA256 composed with itself).
- Checking equality between two stack elements of arbitrary size: `OP_EQUAL`.
- Determining the size, in bytes of any element: `OP_SIZE`. This opcode provides a mapping from Big Script elements to Small Script elements, but not a useful one because all signatures (and all hashes, etc.) have the same size.
- Interpreting arbitrarily-sized elements as booleans, with all-bits-zero elements being *false* and all other values being *true*[8]. Such booleans are used by the `OP_IF` opcode and its variants, which enable branching.

  As with `OP_SIZE`, the branching opcodes provide a mapping from Big Script into Small Script, but not a useful one, because the set of elements which evaluate to *false* is very small and does not overlap with any valid signatures or public keys.

Here the term "arbitrary" refers to stack elements of byte-length 0 up to 520. In practice, we never need to manipulate objects larger than 65 bytes long, so this upper limit is irrelevant to us.

Meanwhile, Small Script, which can implement any function in $\mathrm{Func}_{\mathcal{S}}$, contains opcodes for

- Adding and subtracting 32-bit signed-magnitude integers, but notably not multiplying or dividing them, as well as some other simple arithmetic;
- Numerical comparisons of such integers, as well as minimizing and maximizing, and variants;
- Equality checking via `OP_EQUAL`, as in Big Script;
- Hash functions, the same as in Big Script, though the output of hash functions are not accessible to Small Script;
- A limited form of transaction introspection in the form of the `OP_CHECKSEQUENCEVERIFY` and `OP_CHECKLOCKTIMEVERIFY` opcodes, which do an inequality check between a 40-bit signed-magnitude number and the *sequence* (resp. *locktime*) fields of a transaction. These fields affect the time at which a transaction becomes valid, but not its semantics, making this particular form of introspection useless for covenants.

[8] More correctly, given a stack element of arbitrary size, if any bit is one *except* the first bit, then the element is considered to be *true*. In particular, the empty string is *false*.

Since the `OP_LESSTHANOREQUAL` opcode is complete in the sense that any finite computation can be expressed as a circuit of such operations, it is possible to express any computation in Small Script. The extra opcodes such as addition allow us to do so more efficiently, but even so, a basic operation such as 31-bit multiplication requires about 500 opcodes to implement [10].

A list of Big Script and Small Script opcodes can be found in Appendix A.

2.2.4 Schnorr signature messages

In Bitcoin, the Schnorr signature algorithm uses as a message a *tagged hash* applied to a serialization of the spending transaction, specified in Bitcoin's Taproot Schnorr signature standard [51, 52]. This notion is of central importance to this work, as this tagged hash holds the necessary information to perform transaction introspection and, consequently, covenants. In Definition 8 of Appendix B, we give a full description of this function. Here, instead, we abstract it away using the following definition

Definition 1 (Schnorr sighash)**.** We denote by $\text{SchnorrHash}^{R,P} : \mathcal{X} \to \mathcal{B}$ the algorithm, parameterized by two elliptic curve points R and P, which takes as input a transaction tx, serializes it, and hashes it according to the description in Appendix B. Similarly, we will define by $\text{SchnorrHash}_{\mathcal{S}}^{R,P}$ the small script equivalent, such that

$$\text{SchnorrHash}_{\mathcal{S}}^{R,P}(\langle tx \rangle_{32}) = \langle \text{SchnorrHash}^{R,P}(tx) \rangle_{32}.$$

Whenever R and P are both equal the generator G, we'll simply omit them and write SchnorrHash.

2.2.5 Tapleaf tragedy

In later sections, we will find ourselves writing scripts that need to index into very large lookup tables. The natural way to do this is with a Merkle tree, which we will implement using BLAKE3 (see Section 5.2 for more information). However, implementing Merkle tree lookups directly in Script is very expensive. A natural suggestion is to lever Taproot's built-in Merkle tree lookup functionality.

More specifically, rather than having one script which commits to a the root of a Merkle tree and uses witness data to index into the tree, we could instead have a large Taptree, each branch of which is an individual script. The individual scripts would each commit to a single piece of data (or the root of a sub-tree), and indexing would be done by simply choosing which Tapbranch to use in a given spend.

Unfortunately, such a trick will not work for our purposes, due to the way we construct our covenant. Our construction requires the index (ω, t) of some $d_{\omega,t} \in \mathcal{D}$ to be *independent* of the transaction's signature, to be able to utilize collision attacks on 160-bit hash functions. Using a Taptree eliminates this possibility, as it, by definition, encodes information about (ω, t). A more formal discussion of this point can be found in Remark 2.

We refer to this problem as the *Tapleaf tragedy*. If it were possible to bypass it,

the on-chain cost of our covenant would mostly vanish.

Instead of $\sim$ 4 million opcodes, the covenant logic cost would be less than a quarter of that, and the expensive precomputation phase would not be required. This is because the vast majority of our on-chain cost comes from generating elements from the set $\mathcal{D}$ in Small Script.

Finally, we make two observations about the tapleaf tragedy:

- If there was a sighash mode which did *not* commit to the Tapleaf, such as the proposed `SIGHASH_ANYPREVOUTANYSCRIPT` [11], the tragedy would not apply and we may be able to implement some forms of covenant this way.
- If we were not looking up the hash of a signature, but instead the hash of some other data, the tragedy would not apply. This excludes any form of covenant functionality, since transaction data is only available to Script through signatures, but it may still enable some interesting use cases. See Appendix C for an example.

2.3 Hash functions

In this work, we will focus on the following hash functions:

1. SHA1 : $\mathcal{X} \to \{0,1\}^{160}$;
2. RIPEMD : $\mathcal{X} \to \{0,1\}^{160}$;
3. SHA256 : $\mathcal{X} \to \{0,1\}^{256}$;
4. BLAKE3 : $\mathcal{X} \to \{0,1\}^{256}$.

Below, we list the properties these hash functions have, that are important for our constructions.

Big Script. SHA1 and RIPEMD are available as opcodes in Big Script, while BLAKE3 is not. SHA256, HASH160, and HASH256 are also available as opcodes, but we don't utilize them to generate the set $\mathcal{D}$, as they all depend on a Small Script implementation of SHA256, which is costly.

Small Script. Small Script implementations for BLAKE3 and SHA256 implementations are available in BitVM's repository, requiring, for a single block input of 512 bit, $\sim$ 45K and $\sim$ 296K opcodes [5], respectively[9]. Recently, the SHA256 script was improved by Tomer Giladi [14] in two key ways:

- The Small Script cost was reduced to just $\sim 211K$ opcodes;
- The stack size usage was reduced from 979 elements (out of the 1000 total limit of Bitcoin), to just 732 elements, which is much more workable.

In contrast to the above, SHA1 and RIPEMD don't have a Small Script implementation as of the writing of this paper. Nevertheless, we believe that an efficient implementation of them is possible, so we make the following conjecture:

Conjecture 1. *SHA1 and RIPEMD are implementable in Small Script. For a single block input of* 320 *bit, the former requires* $\sim 70k$ *opcodes, while the latter requires* $\sim 90k$ *opcodes.*

Security model. For simplicity of analysis, we consider all aforementioned hash functions as ideal random functions, namely we work in the *random oracle model*. For some of them, SHA1 in particular, there are non-trivial 2-way collision attacks [45]. Moreover, [21] showed that sometimes finding an n-way collision is not even harder than finding 2-way collisions. Nevertheless, all of these attacks require long inputs. Here, in contrast, the adversary is only ever allowed to supply short inputs, at the length of at most a few blocks, so these attacks do not apply. Hence, the best possible attacks on hash functions in our model follow the birthday bound:

Fact 1. *Let* $f : \mathcal{X} \to \{0,1\}^m$ *be a random function. Given an oracle access to* f*:*

- *Finding distinct* x_0, x_1 *such that* $f(x_0) = f(x_1)$ *requires on average* $\Theta(2^{m/2})$ *queries to* f*;*
- *Finding distinct* x_0, x_1, x_2 *such that* $f(x_0) = f(x_1) = f(x_2)$ *requires on average* $\Theta(2^{2m/3})$ *queries to* f*.*

2.4 Transaction grinding and the Schnorr trick

In similar fashion to the Schnorr trick [35, 36], in this work we'll be interested in grinding transaction data until the transaction hash satisfies certain conditions. In the following, we define a function TxGrind, which is able to alter some data in a transaction while keeping its semantic meaning intact.

Definition 2 (Transaction grinding)**.** Let TxGrind(tx, ρ) be a function with input transaction data $tx \in \mathcal{B}$ and random data $\rho \in \mathcal{S}^*$, that always outputs transactions tx' that are equivalent to tx, but tx' also embeds the data ρ inside it in some unimportant data field.

[9] Unfortunately, we cannot completely avoid computing SHA256 in Small Script, as SchnorrHash, which is required for transaction introspection, depends on it.

Example 1. *An example* TxGrind *could be embedding the data ρ inside the* nLockTime *field. Alternatively, we can put it in an* `OP_RETURN` *script in an additional output.*

Next, we move on to describing the Schnorr trick, beginning by outlining the `OP_CHECKSIG` opcode for Taproot.

Definition 3. Let $\mathit{SchnorrCheckSig}_{\mathcal{B}} : \mathcal{B} \times \mathcal{B} \to \{0,1\}$ be the `OP_CHECKSIG` opcode for Taproot, as defined in Definition 7 of Appendix B, via the following formula

$$\mathit{SchnorrCheckSig}_{\mathcal{B}}(R\|s,P) = \begin{cases} 1, & sG = R + \mathrm{Hash}_{\mathrm{BIP0340/challenge}}(R\|P\|\mathrm{MsgHash}(tx))P, \\ 0, & \text{otherwise} \end{cases}$$

where tx is the spending transaction.

The significance of the Schnorr trick lies in it being a way to force the spender to provide a slightly-altered, hashed form of tx as part of a signature provided as input to the locking script. These slight alterations can be undone and the transaction data pulled apart. We outline how this is done below.

Schnorr trick stage I. The locking script asks the spender to provide

$$s = \mathrm{SchnorrHash}(tx) = \mathrm{Hash}_{\mathrm{BIP0340/challenge}}(G\|G\|\mathrm{MsgHash}(tx)) + 1,$$

the authenticity of which it can check by running $\mathit{SchnorrCheckSig}_{\mathcal{B}}(G\|s,G)$, *i.e.,* by validating a signature where both the public key P and nonce R are equal to the generator G.

Note that in Big Script there is no way to check that indeed $R = G$, without a special opcode such as `OP_CAT`. Nevertheless, since we can show small-to-big equivalence, we can just check that $R = G$ is enforced in Small Script.

Schnorr trick stage II. As SchnorrHash outputs a $\mathcal{B}$ element, it is not possible to get from s to $s-1$ or vice versa. In the original `OP_CAT`-based Schnorr trick, this was dealt with using an additional invocation of `OP_CAT` along with the Small Script opcode `OP_1ADD`. However, since in this work we can show small-to-big equivalence, we'll implement $\texttt{OP\_1ADD}_{\mathcal{S}}$ which increments a 256-bit input by 1. More formally,

$$\texttt{OP\_1ADD}_{\mathcal{S}}(\langle x \rangle_{32}) = \langle x+1 \rangle_{32}.$$

This is sufficient to bridge the gap from s (which appears in the above signature signature) to $s-1$ (which is a hash of our transaction data). Below we argue that this operation is efficient.

Fact 2. *For $\texttt{OP\_1ADD}_{\mathcal{S}}$ on an 256-bit input it holds that $\|\texttt{OP\_1ADD}_{\mathcal{S}}\| \leq 400$.*

Remark 1. *Since the* `OP_CAT`*-based Schnorr trick uses* `OP_SHA256` *in Big Script, it is limited to transactions whose serialized size is at most 520 bytes, which is the maximum size of a single stack element. In contrast, we compute* SHA256 *in Small Script; hence, in our case, the bottleneck is instead the Small Script cost of evaluating* SHA256 *on the serialized transaction.*

3 Equivalence check in Bitcoin

In this section we define the *Bitcoin equivalence check*, a notion that is the main focus of this work. Later, we show how to construct a covenant given a construction for such an equivalence check.

The definition we end up with is somewhat involved, the reason being that we're not able to prove the equivalence of any arbitrary signature s and its small-script representation $\langle s \rangle_{32}$ directly. Instead, we need to grind our transaction until we're able to find *a semantically equivalent* transaction whose Schnorr signature's small-to-big equivalence we can prove.

More concretely, we'll use an off-chain algorithm Prove which will simultaneously find the following:

- Randomness ρ, such that the spending transaction tx is grinded with this randomness, obtaining a semantically equivalent transaction $tx = tx(\rho)$;
- $\pi = (\omega, t)$ such that the element $d_{\omega,t} \in \mathcal{D}$ collides (over SHA1) with the signature s of $tx(\rho)$.

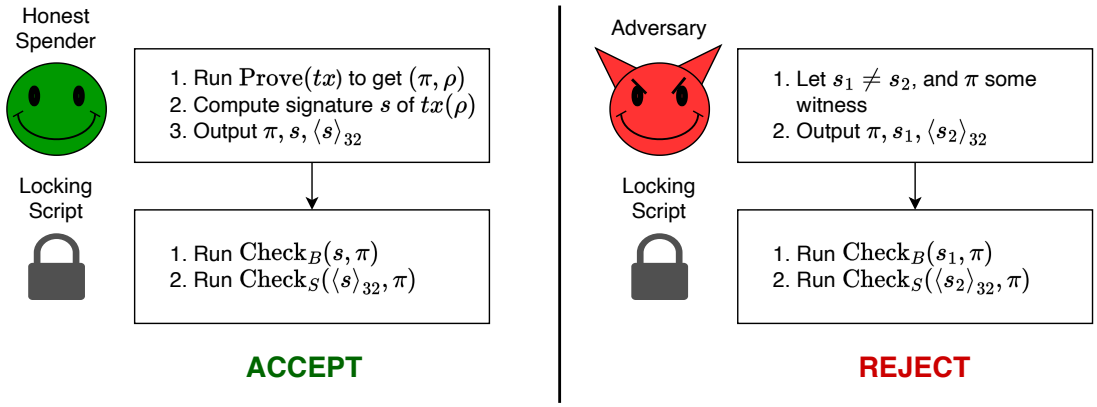

Figure 6: (Left) Equivalence check in the honest spender case. After running Prove, the spender can convince the locking script that a transaction's signatures are equivalent. (Right) Equivalence check in the malicious case. An adversary shouldn't be able to find signatures $s_1 \neq s_2$ whose equivalence they can prove via some witness π.

Once we find such ρ and π by running Prove, as spenders, we'll provide π to the locking script, together with signatures of $tx(\rho)$, namely s and $\langle s \rangle_{32}$. Then, to check the equivalence of the provided signatures s and $\langle s \rangle_{32}$, the locking script will run $\mathsf{Check}_{\mathcal{B}}$ and $\mathsf{Check}_{\mathcal{S}}$, which, using π, will check their collision with d_π in Big Script and Small Script, respectively, as was described in our equivalence check template from Section 1.1. In Figure 6 we illustrate this process, and, below, we formally define it.

Definition 4. A *Bitcoin equivalence check* is a tuple of algorithms $\Pi = (\mathsf{Prove}, \mathsf{Check}_{\mathcal{B}}, \mathsf{Check}_{\mathcal{S}})$, relative to a transaction grinder TxGrind, with the following syntax:

- $\mathsf{Prove}(tx) \to (\rho, \pi)$: On input transaction tx, the algorithm outputs randomness $\rho \in \mathcal{X}$ and a witness $\pi \in \mathcal{S}^*$.
- $\mathsf{Check}_{\mathcal{B}}(R||s, \pi) \to \{0,1\}$: On input $s \in \{0,1\}^{256}$, elliptic curve point R, and witness π, the algorithm either accepts (outputs 1) or rejects (outputs 0).
- $\mathsf{Check}_{\mathcal{S}}(\langle R||s \rangle_{32}, \pi) \to \{0,1\}$: On input $\langle R||s \rangle_{32}$, where $s \in \{0,1\}^{256}$ and R is an elliptic curve point, and witness π, the algorithm either accepts (outputs 1) or rejects (outputs 0).

We require Π to satisfy the following properties:

- **Completeness:**[10] For any transaction tx, if $(\rho, \pi) \leftarrow \mathsf{Prove}(tx)$, then

$$\Pr\left[\mathsf{Check}_{\mathcal{B}}(G||s_{tx,\rho}, \pi) = \mathsf{Check}_{\mathcal{S}}(\langle G||s_{tx,\rho} \rangle_{32}, \pi) = 1\right] \geq 1/2,$$

  where $s_{tx,\rho} = \text{SchnorrHash}(\text{TxGrind}(tx, \rho)) + 1$.

- **Soundness:** We say Π is (S, T, ϵ)-*sound* if for any adversary $\mathcal{A}$ bounded by time T and space S it holds that

$$\Pr\left[\begin{array}{c} R'||s' \neq R||s_{tx} \\ \text{and} \\ \mathsf{Check}_{\mathcal{B}}(R'||s', \pi) = \mathsf{Check}_{\mathcal{S}}(\langle R||s_{tx} \rangle_{32}, \pi) = 1 \end{array} \middle| (tx, R, R', s', \pi) \leftarrow \mathcal{A} \right] \leq \epsilon,$$

  where $s_{tx} = \text{SchnorrHash}^{R,G}(tx) + 1$.

Efficiency. In this work, we'll be interested in the efficiency of $\Pi = (\mathsf{Prove}, \mathsf{Check}_{\mathcal{B}}, \mathsf{Check}_{\mathcal{S}})$. The time complexity of Prove will be measured by total *hash function evaluations*, and the space complexity will be measured by the number of memory bytes required. The efficiency of $\mathsf{Check}_{\mathcal{B}}, \mathsf{Check}_{\mathcal{S}}$ will be measured by how many Bitcoin opcodes and stack spaces are required for execution.

[10] For simplicity of analysis we choose $\geq 1/2$ as the completeness probability threshold. By running Prove multiple times, this can be extended to $\geq 1 - \epsilon$ with a multiplicative overhead of $\log(1/\epsilon)$ to the spender's time complexity.

3.1 A Tapscript covenant

By default, a Bitcoin locking script, $\mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}(w; tx)$, does not depend on tx, with the exception of timelock opcodes. However, in this work, we'll construct a covenant using a Taproot script, allowing the locking script to enforce conditions on the spending transaction tx.

To define a covenant, first, we need to define how it chooses a transaction to accept or reject. We'll abstract it a way using a predicate P, such that $\mathsf{P}(tx) = 1$ means the transaction tx is accepted by the covenant, and $\mathsf{P}(tx) = 0$ means it is rejected.

As before, here, our full-fledged definition will be somewhat involved. This is because, similar to the Schnorr trick [35, 36] covenant, as spenders, we're not able to provide the covenant locking script with an arbitrary valid transaction. Instead, we need to choose a desired valid transaction and grind semantically equivalent transactions, until we find one which the covenant will accept. Below, we call this grinding algorithm $\mathsf{Offchain}$, and unlike the `OP_CAT`-based construction, $\mathsf{Offchain}$ in our construction will need to do quite a bit of work (in particular, it will compute Prove from our equivalence check).

Indeed, the definition we give below can also be applied to capture the `OP_CAT`-based covenant.

Definition 5. Let $\mathsf{P}_{\mathcal{S}}$ be a predicate over transactions such that, for the transaction grinder TxGrind, it holds that for all ρ and tx,

$$\mathsf{P}_{\mathcal{S}}(tx) = 1 \Rightarrow \mathsf{P}_{\mathcal{S}}(\mathrm{TxGrind}(tx, \rho)) = 1.$$

A *covenant*, relative to TxGrind and $\mathsf{P}_{\mathcal{S}}$, is a tuple of algorithms $\Sigma = (\mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}, \mathsf{Offchain})$ with the following syntax:

- $\mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}(w; tx) \to \{0, 1\}$: On input witness $w \in \mathcal{B}^*$, and being spent by a transaction tx, the locking script either accepts (outputs 1) or rejects (outputs 0).
- $\mathsf{Offchain}(tx) \to (w, \rho)$: On input transaction tx, output witness w and randomness ρ.

The algorithms of Σ should satisfy the following requirements:

- **Completeness:** If $\mathsf{P}_{\mathcal{S}}(tx) = 1$ and $(w, \rho) \leftarrow \mathsf{Offchain}(tx)$, then
$$\Pr\left[\mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}(w; \mathrm{TxGrind}(tx, \rho)) = 1\right] \geq 1/2$$
- **Soundness:** We say Σ is (S, T, ϵ)-*sound* if for any adversary $\mathcal{A}$ bounded by time T and space S it holds that
$$\Pr\left[\ \mathsf{P}_{\mathcal{S}}(tx^*) = 0 \text{ and } \mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}(w^*; tx^*) = 1\ \middle|(w^*, tx^*) \leftarrow \mathcal{A}\right] \leq \epsilon$$

In Algorithm 1 we provide pseudocode for performing a covenant in Bitcoin Tapscript.

Proposition 2. *Let Π be a Bitcoin equivalence check with (S, T, ϵ) soundness. Then, Σ from Algorithm 1 is a covenant with (S', T', ϵ') soundness, where $S \approx S'$, $T \approx T'$, and $|\epsilon - \epsilon'|$ is bounded by the success probability of the adversary to find a collision for SHA256*[11].

Proof. We argue correctness and soundness separately.

Correctness: Let tx be a transaction such that $\mathsf{P}_{\mathcal{S}}(tx) = 1$. Since $(\rho, \pi) \leftarrow \Pi.\mathsf{Prove}(tx)$, by Π's completeness it holds, with probability at least $1/2$, that

$$\Pi.\mathsf{Check}_{\mathcal{B}}(G||s_{tx,\rho}, \pi) = \Pi.\mathsf{Check}_{\mathcal{S}}(\langle G||s_{tx,\rho}\rangle_{32}, \pi) = 1,$$

where $s_{tx,\rho} = \mathrm{SchnorrHash}(\mathrm{TxGrind}(tx, \rho)) + 1$. Hence, since $R_1 = R_2 = G$, steps 1–4 don't fail. Moreover, by the definition of $\mathit{SchnorrCheckSig}_{\mathcal{B}}$, steps 5–6 don't fail. Finally, by assumption on tx, $\mathsf{P}_{\mathcal{S}}(\mathrm{TxGrind}(tx, \rho)) = 1$, so step 7 doesn't fail.

Soundness: Let $\mathcal{A}$ be an adversary that wins the security game of Σ with probability ϵ, requiring S space and T time. In other words, by step 1, $\mathcal{A}$ outputs $w^* = (\langle tx\rangle_{32}, \pi, R_1||s_1, \langle R_2||s_2\rangle_{32})$ and tx^* such that

$$\Pr\left[\ \mathsf{P}_{\mathcal{S}}(tx^*) = 0 \text{ and } \mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}(w^*; tx^*) = 1\ \right] = \epsilon.$$

Denote the event when the above happens by $\mathcal{E}$. We'll build an adversary $\mathcal{A}'$ to break Π with roughly the same probability. $\mathcal{A}'$, which has space and time complexity similar to $\mathcal{A}$, does the following:

[11] In the random oracle model, for an adversary doing at most T' queries to SHA256, this is approximately $(T'^2)/2^{257}$.

Algorithm 1 A Bitcoin covenant for a spending transaction tx that uses our equivalence check.

Notation: Let Π be a Bitcoin equivalence check relative to TxGrind.

$\mathsf{Offchain}(tx)$:

1. Let $(\pi, \rho) \leftarrow \Pi.\mathsf{Prove}(tx)$
2. Let $s = \text{SchnorrHash}(\text{TxGrind}(tx, \rho)) + 1$
3. Output witness $(\langle \text{TxGrind}(tx, \rho) \rangle_{32}, \pi, G||s, \langle G||s \rangle_{32})$ and randomness ρ

$\mathsf{LockingScript}_{\mathcal{S},\mathcal{B}}(\langle tx \rangle_{32}, \pi, R_1||s_1, \langle R_2||s_2 \rangle_{32}; \textit{spender\_tx})$:

1. If the witness does not have the correct format (number of elements and their length): return Fail
2. If $\Pi.\mathsf{Check}_{\mathcal{B}}(R_1||s_1, \pi) \neq 1$: return Fail
3. If $\Pi.\mathsf{Check}_{\mathcal{S}}(\langle R_2||s_2 \rangle_{32}, \pi) \neq 1$: return Fail
4. If $\langle R_2 \rangle_{32} \neq \langle G \rangle_{32}$: return Fail
5. If $\textit{SchnorrCheckSig}_{\mathcal{B}}(R_1||s_1, G) \neq 1$: return Fail
6. If $\text{SchnorrHash}_{\mathcal{S}}(\langle tx \rangle_{32}) + 1 \neq \langle s_2 \rangle_{32}$: return Fail
7. If $\mathsf{P}_{\mathcal{S}}(\langle tx \rangle_{32}) \neq 1$: return Fail ▷ Inspect the data of $\langle tx \rangle_{32}$ according to covenant requirements
8. Return Success

---

1. Execute $\mathcal{A}$ to receive $w^* = (\langle tx \rangle_{32}, \pi, R_1||s_1, \langle R_2||s_2 \rangle_{32})$ and tx^*.
2. Return (tx, R_2, R_1, s_1, π).

We claim that

$$\Pr\left[\begin{array}{c} R_1||s_1 \neq R_2||s_{tx} \\ \text{and} \\ \mathsf{Check}_{\mathcal{B}}(R_1||s_1, \pi) = \mathsf{Check}_{\mathcal{S}}(\langle R_2||s_{tx} \rangle_{32}, \pi) = 1 \end{array} \middle| \mathcal{E} \right] \geq 1 - \epsilon_{\text{SHA256}},$$

where $s_{tx} = \text{SchnorrHash}^{R_2,G}(tx) + 1$ and ϵ_{SHA256} is the success probability of the adversary to find a collision for SHA256. Since we're in event $\mathcal{E}$, the negation of steps 1–7 holds. In particular, because of steps 1–2, it is sufficient to show that $R_1||s_1 \neq R_2||s_{tx}$ with high probability to break the soundness of Π. By step 6, we know that $s_2 = s_{tx}$. Moreover, by step 5, it holds that if $R_1 = kG$ then

$$s_1 = k + \text{SchnorrHash}^{kG,G}(tx^*).$$

If $k \neq 1$, by step 4, it holds that $R_1 \neq R_2$ and our claim follows. Suppose then, instead, that $k = 1$. Hence, for $s_1 = s_2$ to hold it must be that

$$\text{SchnorrHash}(tx) = \text{SchnorrHash}(tx^*).$$

However, by step 7 we deduce that $tx \neq tx^*$ as they evaluate to different values of $\mathsf{P}_{\mathcal{S}}$. Thus, by the security of SHA256, the probability an adversary can find such tx, tx^* is at most ϵ_{SHA256}. Taking the negation, with probability at least $1 - \epsilon_{\text{SHA256}}$ we have that $R_1||s_1 \neq R_2||s_{tx}$, breaking the soundness of Π, as required. □

[12] Checking element length can be done with `OP_SIZE`.

Remark 2 (Tapleaf tragedy)**.** *The difficulty of using Taproot's built-in Merkle tree lookup functionality, which can potentially provide significant bitcoin script savings, lies in the difficulty of implementing* Prove *for the equivalence check in this setting. As long as the transaction, tx, does not depend on π,* Prove *in our construction is essentially looking for a pair (π, ρ) that satisfies an equality of the form $f(\pi) = g_{tx}(\rho)$, where $f, g_{tx} : \mathcal{B} \to \{0,1\}^{160}$ behave like hash functions. This is solvable via collision finding algorithms, which we utilize. However, when using Taproot's Merkle tree functionality, the transaction description becomes dependent on π, $tx = tx(\pi)$. This requires a solution of the form $f(\pi) = g_{tx}(\rho, \pi)$, which is no longer amenable to collision finding attacks.*

4 Realizing Bitcoin equivalence tester sets

In this section we formalize the notion of a set $\mathcal{D}$ from the introduction, which will be useful for our equivalence check.

Definition 6. A set $\mathcal{D} = \{d_{\omega,t} \in \mathcal{B}\}_{\omega \in \mathcal{S}, t \in \mathcal{B}}$ is a *Bitcoin equivalence tester set*[12] if it satisfies that

- the mapping $(\omega, \langle t \rangle_1) \mapsto d_{\omega,t}$ is in $\text{Func}_{\mathcal{B}}$, whose implementation we'll denote by $\mathcal{D}.\mathsf{Gen}_{\mathcal{B}}$,
- the mapping $(\omega, \langle t \rangle_1) \mapsto \langle d_{\omega,t} \rangle_{32}$ is in $\text{Func}_{\mathcal{S}}$, whose implementation we'll denote by $\mathcal{D}.\mathsf{Gen}_{\mathcal{S}}$.

We'll also denote by just $\mathcal{D}.\mathsf{Gen}$ the off-chain computation of $d_{\omega,t}$.

Next, we move on to constructing a Bitcoin equivalence tester set, taking inspiration from the GGM pseudorandom function construction [15]. The reason for such a construction is that SHA1 and RIPEMD are the only hash functions that simultaneously (a) exist as opcodes in Big Script and (b) we expect them to have a reasonable Small Script implementation cost.

Algorithm 2 Bitcoin equivalence tester set $\mathcal{D}$

$\mathsf{Gen}_{\mathcal{B}}(\omega, \langle t \rangle_1)$:

1. Let $d \leftarrow \omega$
2. For each t_i compute
   - $d \leftarrow \begin{cases} \text{SHA1}_{\mathcal{B}}(d), & t_i = 0 \\ \text{RIPEMD}_{\mathcal{B}}(d), & t_i = 1 \end{cases}$
3. Return d

$\mathsf{Gen}_{\mathcal{S}}(\omega, \langle t \rangle_1)$:

1. Let $\delta \leftarrow \omega$
2. For each t_i compute
   - $\delta \leftarrow \begin{cases} \text{SHA1}_{\mathcal{S}}(\delta), & t_i = 0 \\ \text{RIPEMD}_{\mathcal{S}}(\delta), & t_i = 1 \end{cases}$
3. Return δ

Proposition 3. *Algorthim 2 is a Bitcoin equivalence tester set $\mathcal{D}$ such that*

- *Evaluating $\mathcal{D}.\mathsf{Gen}_{\mathcal{B}}$ requires at most $||t||$ hash opcodes;*
- *Assuming Conjecture 1, evaluating $\mathcal{D}.\mathsf{Gen}_{\mathcal{S}}$ requires at most $\sim 160,000 \cdot ||t||$ opcodes;*

[12] Even though we write $\omega \in \mathcal{S}$, in our actual construction we will be able to utilize $||\omega|| = 33$, which is no longer a Small Script element. This is because, while 33-bit elements are not directly readable in Small Script, they can nevertheless be generated via `OP_ADD` and other arithmetic opcodes, as these are allowed to output values with a carry.

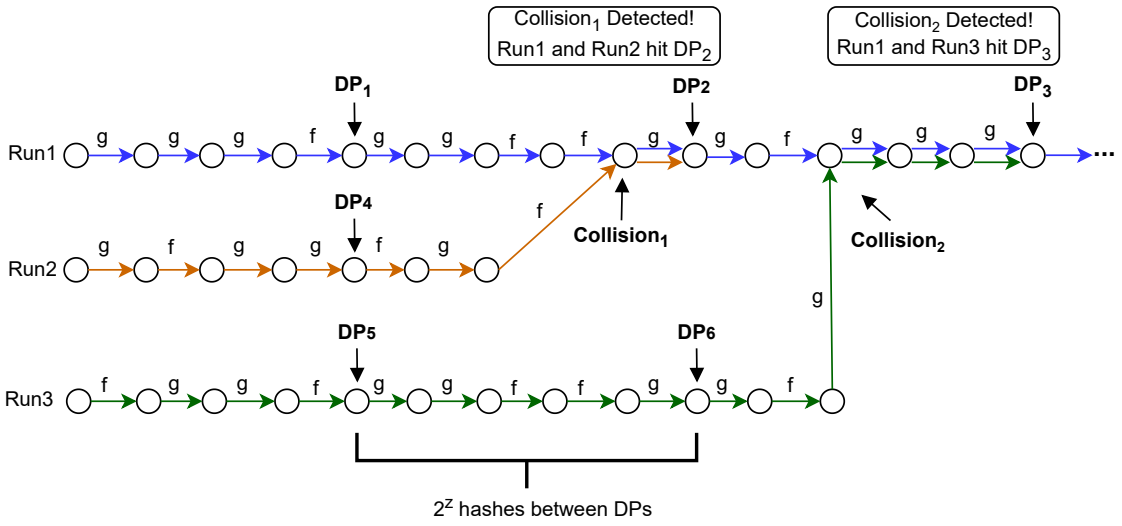

Figure 7: An example execution of our parallel collision algorithm. The circles are 160-bit hash values ρ, the values that we store as distinguished points we denote with $\mathbf{DP}_i$. The lines represent a query to either f or g. Collisions are detected when two runs generate the same distinguished point. $\mathbf{Collision}_1$ is useless collision because the collision is between the same function $f(x) = f(x')$. $\mathbf{Collision}_2$ can be used to spend a covenant because the collisions $f(x) = g(x')$.

- *Suppose that* $||\omega|| = 33$ *and* $35 \leq ||t|| \leq 75$. *Then, in the random oracle model,*

$$|\mathcal{D}| \geq (1 - 2^{-15})2^{||\omega||+||t||}$$

with probability at least $1 - 2^{-19}$.

We prove Proposition 3 in Appendix D. In the following sections, for simplicity of analysis, we assume that $|\mathcal{D}| = 2^{||\omega||+||t||}$ *exactly*, as the deviations of $|\mathcal{D}|$ from this size are negligible.

5 Finding 160-bit collisions

Here we describe our algorithm for finding a collision between a signature and a value in $\mathcal{D}$ used in our equivalence check. We pattern our algorithm on the parallel search distinguished points collision finding approach from [49]. The main challenge we face in designing this algorithm is dealing with the cost of *pseudo-collisions* that arise from the constraints on number of bits used for our input to $\mathcal{D}.\mathsf{Gen}$.

Our equivalence check requires finding a collision between two different 160-bit hash functions $f(x) = g(x')$. By $f(x)$ we denote the hash function that performs $\mathcal{D}.\mathsf{Gen}$. By $g(x)$ we denote the hash function that hashes a Bitcoin signature on a spending transaction. We are only interested in collisions of the form $f(x) = g(x')$. We refer to collisions of the form $f(x) = f(x')$ or $g(x) = g(x')$ as *useless collisions* as we can't use them in our covenants. We combine f and g into a single function h which calls f if the input is prefixed with at least c zeros and calls g otherwise.

$$\mathsf{h}(x) = \begin{cases} \mathsf{f}(x), & x\,|_0^c = \{0\}^c \\ \mathsf{g}(x), & x\,|_0^c \neq \{0\}^c \end{cases}$$

This parameter c allows us to adjust the probability of $f(x)$ or $g(x)$ being called on a random input to $h(x)$. Let's look at these functions $f(x)$ and $g(x)$ in more detail:

$$\begin{aligned} &\mathsf{f}(x) : \{0,1\}^{||w||+||t||} \to \{0,1\}^{160}, && x \mapsto \mathcal{D}.\mathsf{Gen}(x\,|_c^{||\omega||}, x\,|_{c+||\omega||+1}^{c+||\omega||+||t||+1}) \\ &\mathsf{g}(x) : \{0,1\}^{160} \to \{0,1\}^{160}, && x \mapsto \mathrm{SHA1}(G||(\mathrm{SchnorrHash}(\mathrm{TxGrind}(tx, x)) + 1)) \end{aligned}$$

Algorithm 3 Our algorithm for finding collisions needed for our Bitcoin equivalence check

Notation: Let $\mathcal{D}$ be the Bitcoin equivalence tester set from Algorithm 2. We write
$\mathsf{f}(x) = \mathcal{D}.\mathsf{Gen}(x\,|_{c}^{c+||\omega||}, x\,|_{c+||\omega||+1}^{c+||\omega||+||t||+1})$
$\mathsf{g}(x) = \mathrm{SHA1}(G||(\mathrm{SchnorrHash}(\mathrm{TxGrind}(tx, x)) + 1))$
$\mathsf{h}(x) = \begin{cases} \mathsf{f}(x), & x\,|_{0}^{c} = \{0\}^c \\ \mathsf{g}(x), & x\,|_{0}^{c} \neq \{0\}^c \end{cases}$

where we denote the size of w by $||\omega||$, the size of t by $||t||$, and the number of zeros to be a distinguished point by z. Here, the variable c determines frequency of queries to f to g. At $c = 0$ they are both called the an equal number of times, each time c is increased by one, the number of queries to g doubles and the number of queries to f is reduced by half.

$\mathsf{ProveRun}(tx, \mathrm{DP})$:

1. Draw random ρ from $\mathcal{S}^5$
2. Initialize empty prev values table: PT
3. PT.Start $\leftarrow \rho$
4. $y \leftarrow \mathsf{h}(\rho)$
5. While not ($y\,|_{160-z}^{160} = \{0\}^z$ and $y \in \mathrm{DP}$): ▷ While no colliding DP found
   - PT$[y] \leftarrow \rho$
   - If $y\,|_{160-z}^{160} = \{0\}^z$ and $y \notin \mathrm{DP}$: ▷ New DP found
     - DP$[y] \leftarrow$ PT.Start
     - PT $\leftarrow$ Empty
     - PT.Start $\leftarrow \rho$
   - $\rho \leftarrow y$
   - $y \leftarrow \mathsf{h}(\rho)$
6. $y' \leftarrow \mathrm{DP}[y]$
7. While $y' \notin$ PT :
   - $\rho' \leftarrow y'$
   - $y' \leftarrow \mathsf{h}(\rho')$
8. return $\rho', \mathrm{PT}[y']$ ▷ Collision Found

$\mathsf{Prove}(tx)$:

1. Initialize empty dist. point table: DP
2. While True: ▷ Loop can run in parallel
   - $x, x' \leftarrow \mathsf{ProveRun}(tx, \mathrm{DP})$
   - If $x\,|_{0}^{c} = \{0\}^c$ and $x'\,|_{0}^{c} \neq \{0\}^c$: ▷ Useful Collision
     - return $\pi = (\omega \leftarrow x\,|_{c}^{c+||\omega||}, t \leftarrow x\,|_{c+||\omega||+1}^{c+||\omega||+||t||+1})$ and $\rho \leftarrow x'$
   - If $x\,|_{0}^{c} \neq \{0\}^c$ and $x'\,|_{0}^{c} = \{0\}^c$: ▷ Useful Collision
     - return $\pi = (\omega \leftarrow x'\,|_{c}^{c+||\omega||}, t \leftarrow x'\,|_{c+||\omega||+1}^{c+||\omega||+||t||+1})$ and $\rho \leftarrow x$

The function $f(x)$ maps the input value x to (ω, t) and then returns $\mathcal{D}.\mathsf{Gen}(\omega, t)$ as its output. To keep the number of hash function calls low in our equivalence check we use a witness (ω, t) whose size is less than 160-bits. To map an 160-bit input x to a $||\omega|| + ||t|| < 160$-bit input (ω, t) we truncate the bits of x. This results in *pseudo-collisions* where $x \neq x'$ but a collision because the difference between x and x' was removed by truncation. Inputs to f will necessarily start with c zero bits. Since these bits will always be zero, we drop these c bits.

We will now walk through our collision finding algorithm shown in Figure 7. The algorithm starts r parallel runs. Each run is seeded with a different random value ρ. And recursively calls $h(\rho)$ supplying the output from the last call as the input to the next hash function call:

$$h(\dots h(h(\rho))\dots)$$

Each generated hash output is written to a hash table denoted, PT. This table, PT, is reset each time the instance hits a *distinguished point* so that it doesn't grow without bound. For our algorithm, we define a distinguished point as an output that ends with at least z zero bits.

$$y\,|_{160-z}^{160} = \{0\}^z$$

When an instance generates a distinguished point, it checks if the distinguished point *already exists* in a table of distinguished points which we denote DP. What we do next depends on if the output already in the distinguished points table or not.

If the output y does not exist in the distinguished points table, *i.e.,* $y \notin \text{DP}$, we add y in the Distinguished Points table using y as the key and PT.Start as the value. The variable PT.Start stores the last distinguished point this run encountered or the initial random seed ρ it started from. We need to set $\text{DP}[y] \leftarrow \text{PT.Start}$ so that if we hit this distinguished point again we can reconstruct the chain of calls to h allowing us to find the exact set of inputs which generated the collision. After adding the new distinguished point to the table we continue recursively calling h.

If on the other hand the output does exist in the distinguished points table, *i.e.,* $y \in \text{DP}$, we have detected that a collision occurred. To determine when the collision happened, we reconstructing the chain until we find some output that collides with an output in the previous values table (PT) of the current instance

$$h(\dots h(h(\text{DP}[y]))\dots)) \in \text{PT}$$

Once we have found this match, we have found the colliding inputs

$$x = h(\dots h(h(\text{DP}[y]))\dots)$$

$$x' = \text{PT}[h(\dots h(h(\text{DP}[y]))\dots)]$$

$$h(x) = h(x')$$

We know that $x \neq x'$ because the prior input in the chain would have been identified as the collision.

Once we have this collision we check if it is a useful $f(x) = g(x')$ collision. Remember that h chooses whether to call f or g based if the input is prefixed by c zeros. Hence we can determine if we have a useful collision by checking that one of the two returned input is prefixed by c zero bit and the other input is not prefixed by c zero bits. That is, we have a useful collision when:

$$(x\,|_0^c = \{0\}^c \text{ and } x'\,|_0^c \neq \{0\}^c) \text{ or } (x\,|_0^c \neq \{0\}^c \text{ and } x'\,|_0^c = \{0\}^c).$$

If the collision found is a useless collision, we restart the instance with a new random seed. If we have found a useful collision we stop our search return the colliding ρ and (ω, t):

$$\mathcal{D}.\mathsf{Gen}(\omega, t) = \text{SHA1}(G||(\text{SchnorrHash}(\text{TxGrind}(tx, \rho)) + 1))$$

Algorithm 3 presents a collision finding technique, which will be part of our Bitcoin equivalence check $\Pi = (\mathsf{Prove}, \mathsf{Check}_{\mathcal{B}}, \mathsf{Check}_{\mathcal{S}})$. The full construction is given in Algorithm 4, which, when combined with Algorithm 1, yields a covenant for Bitcoin.

To summarize the discussion above, we deduce the following

Proposition 4. *$\Pi = (\mathsf{Prove}, \mathsf{Check}_{\mathcal{B}}, \mathsf{Check}_{\mathcal{S}})$ as described in Algorithm 4, when instantiated with the correct parameters (such that the success probability $\geq 1/2$), satisfies the Completeness property of a Bitcoin equivalence check.*

Algorithm 4 Bitcoin equivalence check

Notation: Let $\mathcal{D}$ be the Bitcoin equivalence tester set from Algorithm 2.

$\mathsf{Prove}(tx)$:

1. Return the result of $\mathsf{Prove}(tx)$ from Algorithm 3.

$\mathsf{Check}_{\mathcal{B}}(R||s, \omega, \langle t \rangle_1)$:

1. If the input does not have the correct format (number of elements and element length): return Fail
2. Let $d = \mathcal{D}.\mathsf{Gen}_{\mathcal{B}}(\omega, \langle t \rangle_1)$
3. Return Success if and only if $\mathrm{SHA1}_{\mathcal{B}}(d) = \mathrm{SHA1}_{\mathcal{B}}(R||s)$.

$\mathsf{Check}_{\mathcal{S}}(\langle R||s \rangle_{32}, \omega, \langle t \rangle_1)$:

1. If the input does not have the correct format (number of elements and element length): return Fail
2. Let $\delta = \mathcal{D}.\mathsf{Gen}_{\mathcal{S}}(\omega, \langle t \rangle_1)$
3. Return Success if and only if $\mathrm{SHA1}_{\mathcal{S}}(\delta) = \mathrm{SHA1}_{\mathcal{S}}(\langle R||s \rangle_{32})$.

In regards to Algorithm 4, in the next sections, we'll analyze the space and time complexity of $\Pi.\mathsf{Prove}$, followed by the Small Script cost of $\Pi.\mathsf{Check}_{\mathcal{S}}$. After this, we'll instantiate Π with concrete parameters and analyze its soundness.

5.1 Time and space cost

In this section we summarize the time and space of our collision finding algorithm, namely $\Pi.\mathsf{Prove}$ from Algorithm 4. Appendix F provides the complete derivation of this analysis.

Our analysis depends on the following variables:

z:	The number of zeros that identifies a point as a distinguished point.
qf:	The number of unique queries to f is 2^{qf}
qg:	The number of unique queries to g is 2^{qg}

where, for a given parameter c, it holds that $qg - qf = c$. Note that Prove in Algorithm 3 does not have an explicit running time due to the "While True" in step 2. Instead, by choosing to terminate the loop early and controlling the value c, it is possible to freely control the values qf and qg.

The probability of a *useful collision i.e.,* a collision allows us to spend a covenant, is:

$$1 - e^{-2^{(qf+qg-160)}}$$

Thus when $qf+qg = 160$ we have a greater than $1/2$ probability of a useful collision because $1-e^{-2^{(0)}} \approx 0.632$. As we will show next this does not tell us the total work we need to do. This is because not every query to f or g is unique due to *useless collisions i.e.,* collisions not useful for spending a covenant.

Useless collisions: Each time we hit a collision it takes us 2^{z-1} additional hash queries before we reach a distinguished point in the distinguished point table. Then determining where the collision occurred requires an additional 2^{z-1} work. Thus, each collision we encounter costs an additional $2^{z-1} + 2^{z-1} = 2^z$ work. Most of the useless collisions we generate are pseudo-collisions *i.e.,* collisions that arise because we truncate the bits of the input inside of f. The expected number of $f(x) = f(x')$ collisions these including pseudo-collisions is:

$$2^{2qf-||w||-||t||-1}$$

As we don't truncate the input in g, g does not generate any pseudo-collisions. The expected number of $g(x) = g(x')$ useless collisions for 2^{qg} queries to g as:

$$2^{2qg-160-1}$$

The total number of useless collisions is given by:

$$2^{2qf-||w||-||t||-1} + 2^{2qg-160-1}$$

For a greater $1/2$ probability of finding a useful collision, $qf + qg \geq 160$ the expected work is:

$$2^z \times (2^{2qf-||w||-||t||-1} + 2^{2qg-160-1})$$

Additional hash queries in f: Notice that f costs more hash queries than g because f contains $\mathcal{D}.\mathsf{Gen}$ which performs one hash query per bit in t. We can count these additional queries by multiplying by $||t||$. For useless collisions:

$$2^{2qf+\log_2||t||+z-||w||-||t||-1} + 2^{2qg+z-160-1}$$

For the number of unique queries:

$$2^{qf} \times ||t|| + 2^{qg} = 2^{qf+\log_2||t||} + 2^{qg}$$

Time cost: As we have a greater than $1/2$ probability of finding a useful collision when $qf+qg = 160$, we set $qf + qg = 160$ and define qg as a function of $qg = 160 - qf$ giving us the *total time cost* of:

$$2^{qf+\log_2||t||} + 2^{160-qf} + 2^{2qf+\log_2||t||+z-||w||-||t||-1} + 2^{2(160-qf)+z-160-1}$$

Space cost: The space cost of our algorithm is simply storing our distinguished points table. We define our distinguished points as an output which has at least z zero bits at the end. This means that 2^{-z} is probability of a query to the hash function producing a distinguished point and 2^z is the number of queries between two distinguished points. A naive approach would be for each row on the distinguished point table to include the distinguished point (size $160/8 = 20$ bytes), and the previous distinguished point (or random seed of the run).

Observing the fact that distinguished points are added to the table as they are discovered, if we group distinguished point by the run which discovered them the previous distinguished point will just be the previous row in the table allowing us to reference it implicitly using only negligible space. Since all distinguished points end in z we don't need to store the z all zero bits. Thus a row in our distinguished point table is $(160-z)/8$ bytes. If the total number of unique queries we make is $Q = 2^q$, then distinguished point table will have 2^{q-z} rows and the table will use space $2^{q-z} \times (20 - z/8)$.

Each parallel run stores the table PT of the previous hash outputs. When a run hits a distinguished point, that runs PT is reset to empty. The space need for PT is the number of parallel runs, r, times the expected number of outputs between distinguished points 2^z. This results in $20 \times r \times 2^z$ space. We employ a time space trade-off here by only storing every $2^{z/2}$-th output in PT and then reconstruct the matching region of the PT for an increase in $2^z/2^{z/2} = 2^{z/2}$ time and $r \times 20 \times 2^{z/2}$ space. This value is small enough relative to the other values in time and space that we can ignore it.

Thus the *total space cost* is:

$$(20 - z/8) \times (2^{qf-z} + 2^{(160-qf)-z})$$

To summarize,

Proposition 5. *Let $qf, ||\omega||, ||t||, z$ be positive integers such that $qf \leq 160$ and $||\omega|| + ||t|| \leq 160$. Then, the total running time of* Prove *in Algorithm 4 is given by at most*

$$2^{qf+\log_2||t||} + 2^{160-qf} + 2^{2qf+\log_2||t||+z-||w||-||t||-1} + 2^{2(160-qf)+z-160-1}$$

hash function evaluations and its total memory usage in bytes is given by

$$(20 - z/8) \times (2^{qf-z} + 2^{(160-qf)-z}).$$

While Proposition 5 estimates the off-chain complexity of Prove, we are also interested in the Small Script cost of $\mathsf{Check}_{\mathcal{S}}$. As it is, the cost is too prohibitive to fit into 4 million opcodes due to the complexity of $\mathcal{D}.\mathsf{Gen}_{\mathcal{S}}$, as given in Proposition 3. In the following section, we improve the Small Script cost of $\mathsf{Check}_{\mathcal{S}}$ via precomputing part of $\mathcal{D}.\mathsf{Gen}$ and storing it in a Merkle tree, which can be implemented using *any* hash function. We choose to use BLAKE3 due to its cheap Small Script cost.

5.2 Reducing Small Script costs with BLAKE3

The essential idea here is to replace many of hash function queries needed to compute $\mathcal{D}.\mathsf{Gen}$ in Small Script with a lookup table that maps the first u-bits of the equivalence witness $\pi = (\omega, t)$ to output of $\mathcal{D}.\mathsf{Gen}$ on that truncated witness. Rather than storing this enormous lookup table in Small Script we use a Merkle tree to commit to it and the spending transaction simply proves that a particular input, output pair exist in the Merkle tree. This allows us to compress the bytes we need for our Small Script RIPEMD and SHA1 queries.

Using a Merkle tree has several space savings advantages. First, the hash function used to compute this Merkle tree can be *any* hash function. We use BLAKE3 as it is currently the most compact available implementation of hash function in Small Script. Second, we only need *one* BLAKE3 query per level of the tree, whereas each RIPEMD and SHA1 query requires having Small Script code for *both* hash functions for each bit of t. Third, we can truncate the output of BLAKE3 to 128-bits and have a radix 4 Merkle tree, further reducing the number of hash function calls we need in Small Script.

Below, we look at how to construct this Merkle tree, why it is safe to truncate the BLAKE3 output from 512-bits to 128-bits in the Merkle tree, and how we can get even more Small Script savings by replacing the first 5 levels of the Merkle tree with a lookup table.

5.2.1 Constructing the Merkle tree

We parameterize our Merkle tree with the value u, which is number of bits of the equivalence witness, $\pi = (\omega, t)$ that the Merkle tree will compress. That is, u layers of the binary tree of $\mathcal{D}$ will be stored in the Merkle tree and the rest $||\omega|| + ||t|| - u$ layers will be computed using RIPEMD and SHA1.

Each *leaf* of the Merkle tree consists of three outputs from $\mathcal{D}.\mathsf{Gen}$: $\text{BLAKE3}(\mathcal{D}.\mathsf{Gen}(\pi)||\mathcal{D}.\mathsf{Gen}(\pi + 1)||\mathcal{D}.\mathsf{Gen}(\pi + 2))$. As BLAKE3 has a message block size of 512-bits, it allows us to put three 160-bit $\mathcal{D}.\mathsf{Gen}(\pi)$ values in a single BLAKE3 message block. All other nodes in the Merkle tree are the hash of four nodes rather than three.

The leaves of Merkle are sorted by the from low to high by the equivalence witness, with left most leaf being

$$\mathcal{D}.\mathsf{Gen}(\omega = 0, t = 0)||\mathcal{D}.\mathsf{Gen}(\omega = 0, t = 1))||\mathcal{D}.\mathsf{Gen}(\omega = 0, t = 2)$$

and the right most leaf being

$$\mathcal{D}.\mathsf{Gen}(\omega = 2^{||\omega||} - 1, t = 2^{||t||} - 3)||\mathcal{D}.\mathsf{Gen}(\omega = 2^{||\omega||} - 1, t = 2^{||t||} - 2))||\mathcal{D}.\mathsf{Gen}(\omega = 2^{||\omega||} - 1, t = 2^{||t||} - 1)$$

In this way, the Merkle tree commits to the equivalence witness, the input to $\mathcal{D}.\mathsf{Gen}$, as well as committing to the output of $\mathcal{D}.\mathsf{Gen}$ for that particular input. Thus, the Merkle path allows us enforce the relationship between input (ω, t) and outputs $\mathcal{D}.\mathsf{Gen}(\omega, t)$ by checking if the leaf position in the Merkle tree matches the equivalence witness supplied by the spending transaction.

Throughout the Merkle tree computation, we truncate the output of BLAKE3 to 128 bits to save space by having one one BLAKE3 compression function call per *inner node*. Hence, each *inner node* of the Merkle tree takes four BLAKE3 outputs truncated to 128 bits, x_0, x_1, x_2, x_3, and computes another (truncated) hash over them as follows: $\text{BLAKE3}(x_0||x_1||x_2||x_3) \mid_0^{128-1}$. The Merkle tree is radix 4, but the leaves contain three outputs. Thus, the depth of the Merkle tree is $\log_4(n/3)$, where $n = 2^u$ is in the number of values committed to.

Cutting the top off of the tree. We can gain additional savings in bytes by replacing the top five levels of the tree with a lookup table. Storing a lookup table of size uses $16 \times 4^5 = 16,384$ bytes allowing us to save five BLAKE3 calls and each BLAKE3 call costs $\sim 45,000$ bytes. Spaces savings $5 \times 45,000 - 16,384 = 208,616$ bytes. If we store 2^{90} leaves in the Merkle tree, we get a depth of 45 in the Merkle tree, removing the top 5 levels we require 40 only BLAKE3 evaluations.

Is 128-bit truncation safe? Considering the size of this Merkle tree and the truncated output of 128-bits there are bound to be a large number of collisions within the Merkle tree itself. To address this, the membership test checks both that the output exists in the Merkle tree and also that it is the correct output for equivalence witness $\pi = (\omega, t)$ by checking the leafs position in the Merkle tree. The intuition here is that because the Merkle root was created in a trusted manner, for any particular leaf of the Merkle tree, an attacker needs to find the 2nd preimage of the root or any node on the path to the root, as opposed to just finding a collision between nodes (which is easier).

As given in [26], the probability of a collision in a Merkle tree of output size of m, and path depth of k is, where the root is fixed and known is:

$$2^{-m} + e^{-2^{-m}} - e^{-(k+1)2^{-m}}$$

Thus, for example, for $m = 128$ and $k = 90$ we get an adversary advantage of

$$2^{-128} + e^{-2^{-128}} - e^{-(90+1)2^{-128}} = 91 \times 2^{-128} \approx 2^{121.5}.$$

Thus, our Merkle tree provides ≈ 121.5 bits of security.

To summarize, we get the following result

Proposition 6. *It is possible to precompute 2^u hashes, modifying $\mathsf{Check}_{\mathcal{S}}$ from Algorithm 4 such that*

- *its Small Script cost is*

  $$\sim 17,000 + (\lceil \log_4(2^u/3) \rceil + 1 - 5) \cdot ||BLAKE3_{\mathcal{S}}|| + (||\omega|| + ||t|| - u) \cdot (||SHA1_{\mathcal{S}}|| + ||RIPEMD_{\mathcal{S}}||),$$

  as opposed to the old cost of

  $$\sim ||t|| \cdot (||SHA1_{\mathcal{S}}|| + ||RIPEMD_{\mathcal{S}}||).$$

- *Moreover, for any integer $0 \leq \ell \leq \lceil \log_4(2^u/3) \rceil$, Prove from Algorithm 4 is modified accordingly as follows:*
  - *there is an additional off-chain memory overhead of $4^{\ell+2}/3$ bytes;*
  - *there is an additional off-chain computational overhead of $\approx 2^{u-2\cdot\ell}$ hashes.*

Proof. The $\mathsf{Check}_{\mathcal{S}}$ Small Script cost claim follows from the discussion above and Proposition 3.

The additional off-chain overhead for Prove follows from the fact that we only the store the Merkle tree root in the locking script. To spend the locking script, decommiting the Merkle tree is required. Our solution is to keep the first ℓ layers of the radix 4 tree in storage while using $\approx 2^{u-2\cdot\ell}$ hashes to compute the last layers of the Merkle tree. Since we store 128-bit hashes per leaf, they require 4 bytes each. □

As an example instantiation for Proposition 6, let $||\omega|| = 33$, $||t|| = 67$, $u = 90$, and assume Conjecture 1 holds. This implies a set $\mathcal{D}$ of size 2^{100}, which, when using 2^{90} precomputation, reduces the cost of $\mathsf{Check}_{\mathcal{S}}$ to about 3.3 million opcodes, versus 16 million using the naive approach. Choosing $\ell = 20$, the off-chain overhead of Proposition 6 is ~ 5.86 terabytes of memory and computing $\sim 2^{50}$ hashes per spend, which is negligible compared to the cost of other parts of Prove. In 8 we show how given a Small Script implementation size of SHA1 and RIPEMD we can choose a u that allows our covenant to fit in a 4 MB Bitcoin transaction

6 Concrete parameters and costs

One challenge we face is how to perform the equivalence check in Small Script while (a) being feasible to execute off-chain and (b) staying under the maximum Bitcoin script size of roughly 4 million opcodes.

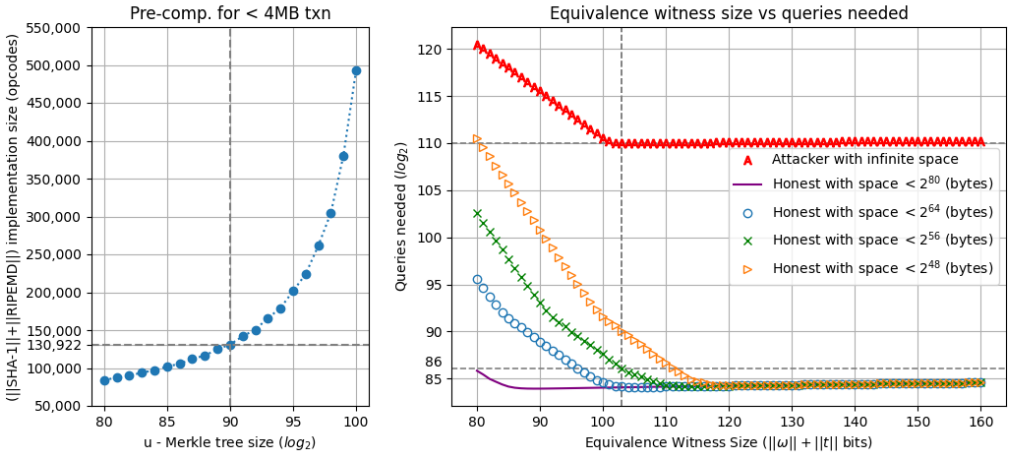

Figure 8: (Left) Plots the maximum allowable size that the small script implementation of SHA1 and RIPEMD can be for a particular size of Merkle tree before the covenant transaction will be greater than 4 Megabytes, *e.g.,* for a Merkle tree containing 2^{90} elements ($u = 90$), the summed small script implementation of SHA1 and RIPEMD can not be larger than 130,922 opcodes. (Right) Graphs the cost in number of hash queries needed to honestly spend a covenant for different equivalence witness sizes, compared with the cost to break the covenant.

6.1 Likely parameters

To minimize $||t||$, we choose the largest value allowed by Small Script $||w|| = 33$ bits[13]. As $||w|| + ||t||$ is likely be much smaller than 160, the number of f to f collisions will be much larger than the number of g to g useless collisions when $qf = qg$. We can reduce the overall number of useless collisions by making g more frequent than f by adjusting c.

The space requirements depend on size of the distinguished points table which is a function of both qf and z. Assuming no more than 4096 16-TB hard drives for the distinguished points gives us a limit of 2^{56} bytes. We note that this space cost is not unreasonable, especially as cloud services can provide petabytes of storage for a monthly fee, and as long as multiple covenants are computed sequentially, this cost is *amortized* over all the covenants evaluated during that period of time.

Our choice of parameters places the following restrictions on qf and z:

$$2^{56} \geq (20 - z/8) \times (2^{qf-z} + 2^{(160-qf)-z})$$

We find that[14] for $z = 33$, $qf = 75$, $qg = 85$, $||t|| = 70$, $||\omega|| = 33$ we get time $2^{86.11}$ and space $2^{55.99}$.

Small Script cost. We rely on Proposition 6, together with the implementations from [5, 14] and Conjecture 1. We'll make a few additional comments:

1. When computing the last layers of $\mathcal{D}$, it is possible to replace the last layer cost of $||\text{SHA1}|| + ||\text{RIPEMD}||$ by just $\min\{||\text{SHA1}||, ||\text{RIPEMD}||\}$. This is because the last layer can either compute a hash function if $t_j = 1$ or use the identity function if $t_j = 0$.

2. Extracting the transaction data from the SchnorrHash requires at least two $\text{SHA256}_{\mathcal{S}}$ evaluations. If the size of the transaction is small, each $\text{SHA256}_{\mathcal{S}}$ evaluation will be on a single input block ($\text{SHA256}_{\mathcal{S}}$'s cost grows linearly with the number of input blocks).

[13] While Small Script opcodes can not take a 33-bit element as an input, the Small Script `OP_ADD` can generate a 33 bit output. This allows us to verify the equivalence between a 33-bit element and that element represented in Small Script as a 32-bit and 1-bit element by adding the smaller elements to together and checking equality.

[14] A less conservative choice of 2^{58} distinguished points table size would reduce the hash queries to $2^{84.86}$.

3. To compute $\mathsf{Check}_{\mathcal{S}}$, a Schnorr signature needs to be evaluated on with $\text{SHA1}_{\mathcal{S}}$. We assume this requires $70K$ additional opcodes.

Recall that $|\mathcal{D}| = 2^{103}$, where $||\omega|| = 33$ and $||t|| = 70$. Thus, choosing to perform 2^u precomputation in Proposition 6, where $u = 90$, we can make the following estimation on the covenant's Small Script cost:

$$2 \cdot 211K + (17K + (\lceil \log_4(2^{90}/3) \rceil + 1 - 5) \cdot 45K) + (12 \cdot 160K + 70K) + 70K,$$

which totals to about 4.34 million opcodes, *just above* Bitcoin's 4 million opcode limit.

We mention multiple ways to go below this limit. One of them is to instead choose $u = 93$ in Proposition 6, resulting in the following cost

$$2 \cdot 211K + (17K + (\log_4(2^{93}/3) + 1 - 5) \cdot 45K) + (9 \cdot 160K + 70K) + 70K,$$

which totals to 3.91 million opcodes. Another possible choice is to note that, as mentioned in Section 5.2, BLAKE3 can be replaced with any other hash function in our Merkle tree construction. Thus, we make the following conjecture

Conjecture 2. *There exists a Bitcoin-friendly hash function implementable in Small Script, that, for a single block input of* 512 *bit, requires* $\sim 35k$ *opcodes.*

This brings our previous calculation to

$$2 \cdot 211K + (17K + (\lceil \log_4(2^{90}/3) \rceil + 1 - 5) \cdot 35K) + (12 \cdot 160K + 70K) + 70K,$$

which totals to just about 3.93 million opcodes.

As another example of parameter instantiation, suppose we still use BLAKE3, but assume a more aggressive version of Conjecture 1, where SHA1 and RIPEMD are also as efficient as BLAKE3, each requiring $\sim 45K$ opcodes. This allows us to get away with only 2^{86} precomputation cost, *i.e.,* $u = 86$. In this scenario, the total cost is

$$2 \cdot 211K + (17K + (\lceil \log_4(2^{86}/3) \rceil + 1 - 5) \cdot 45K) + (16 \cdot 90K + 70K) + 45K,$$

bringing us to 3.72 million opcodes.

Finally, in Figure 8 (left) we assume SHA1 and RIPEMD are equal in size and then we plot the precompuation cost 2^u needed to keep the transaction under the 4MB size limit. For $u = 90$, both hash functions must sum to $130,922$, costing

$$2 \cdot 211K + (17K + (\lceil \log_4(2^{90}/3) \rceil + 1 - 5) \cdot 45K) + (12 \cdot 130K + 65K) + 65K,$$

which sums up to 3.98 million opcodes[15].

7 Security discussion

We now turn our attention to the security of our equivalence check. The job of an attacker is break soundness, *i.e.,* construct an input $(\langle tx \rangle_{32}, \pi, R_1||s_1, \langle R_2||s_2 \rangle_{32}; spender\_tx)$ to our equivalence check which passes the equivalence check but the Big Script input, $R_1||s_1$, and the Small Script input, $R_2||s_2$ are not in fact equivalent, $R_1||s_1 \neq R_2||s_2$. This necessarily requires finding a triple collision:

$$\text{SHA1}(R_1||s_1) = \mathcal{D}.\mathsf{Gen}(\omega, t) = \text{SHA1}(R_2||s_2)$$

We present two potential attacks that aim to find a triple collision and break soundness. We treat SHA1 and RIPEMD as 160-bit ideal hash functions. In the next Section 7.1 we outline our arguments for why we consider SHA1 reasonable to model as an ideal hash function given known weaknesses [45] leaving a more detailed analysis for future work.

[15] We provide our code for this at github.com/EthanHeilman/collision-covenants/settings

Approach 1 (Fix and Search):

1. Find $R_1||s_1$ and (ω, t) such that $\text{SHA1}(R_1||s_1) = \mathcal{D}.\mathsf{Gen}(\omega, t)$ using our collision finding algorithm.
2. Then attempt to transform this into a triple collision by finding a value $R_2||s_2$ such that $\text{SHA1}(R_2||s_2) = \mathcal{D}.\mathsf{Gen}(\pi)$ and $R_1||s_1 \neq R_2||s_2$.

Since (ω, t) is fixed, finding $\text{SHA1}(R_2||s_2) = \mathcal{D}.\mathsf{Gen}(\pi)$ is requires computing the second preimage of $R_1||s_1$ which needs 2^{160} queries (assuming a secure hash function).

Approach 2 (Meet-in-the-Middle):

1. Generate $P = 2^p$ unique pairs of signatures $(R_1||s_1, R_2||s_2), (R_3||s_4, R_4||s_4), \ldots$ such that each pair collides $\text{SHA1}(R_i||s_i) = \text{SHA1}(R_j||s_j)$. The signature on the left will be accepted by the covenant, the signature on the right would be rejected by the covenant.
2. Then try to find a π that collides with one of these pairs such that

$$\text{SHA1}(R_i||s_i) = \text{SHA1}(R_j||s_j) = \mathcal{D}.\mathsf{Gen}(\omega, t)$$

We consider an attacker with infinite space that is concerned only with the time cost. This allows the attacker to find collisions by simply by constructing a giant lookup table. In Appendix G we include an analysis in which the attacker can not use more than 2^{64} bytes and uses a distinguished points parallel collision algorithm. We do not include it here as it does not alter the time cost.

We denote the number of queries we make in step 1 as 2^{q_1} and the number of queries we make in step 2 as $2^{q_2+\log_2(||t||)}$. The $\log_2(||t||)$ in step 2 is the result of the $||t||$ additional queries that are made to perform a query to $\mathcal{D}.\mathsf{Gen}$. To succeed with greater than $1/2$ probability the attacker must make:

$$2^{q_1} + 2^{q_2+\log_2(||t||)}$$

queries, such that $q_2 \leq ||\omega|| + ||t||$ and the following inequality holds:

$$\frac{2^{2q_1-161} \cdot 2^{q_2}}{2^{160}} \geq 1/2$$

Simplifying the inequality we get:

$$2q_1 + q_2 \geq 320.$$

Thus to find the work the attacker needs to do, we find a q_1, q_2 which minimizes the number of queries, which satisfying the two conditions.

The above gives us the time cost of our best attack on our equivalence-check, but in the context of covenants the number of queries in step 1 doubles. This is because to cheat a covenant, you need one of the colliding signatures to pass the covenant's rules and the other colliding signatures to be rejected by the covenant. Since only half of these collisions generated in step 1 can be used to break our covenant, this requires twice as many unique queries for step one increasing the threshold for the attacker to:

$$2q_1 + q_2 \geq 321.$$

Plugging in our proposed witness size parameters of $||\omega|| = 33$ and $||t|| = 70$ we get $q_1 = 109.3764$ and $q_2 = 102.2471$ which requires

$$2^{109.3764} + 2^{102.2471+6.12} = 2^{109.3764} + 2^{108.3671} \approx 2^{109.9}$$

queries and satisfies the conditions $q_2 = 102.2471 \leq 103$ and

$$2 \cdot 109.3764 + 102.2471 \approx 321.$$

The time cost of creating a transaction that honestly spends the covenant is $2^{86.11}$. The time cost of constructing a transaction that cheats the covenant is $2^{109.9}$. The means that for our proposed parameters an attacker must do $2^{23.79} = 14,504,500$ times more work than an honest spender. Figure 8 (Right) shows how the cost in queries to honestly spend or attack our covenant changes for different sizes of equivalence witness sizes.

The Bitcoin hash rate is roughly 700 exahashes/second. This is $2^{69.25}$ hashes/second, $2^{85.64}$ hash/day, $2^{91.12}$ hashes/year. Honestly spending a covenant has work roughly in terms of number of hash queries to roughly 33 hours of Bitcoin mining. Cheating such a covenant based on the best attack we have developed costs $450,136$ years of the Bitcoin mining network.

7.1 SHA1 known weaknesses

Our above security analysis models SHA1 as an ideal hash function despite SHA1's known weaknesses that place the cost of finding collisions at $2^{63.4}$ [29], well below the generic collision resistance, 2^{80} of a 160-bit hash function that we assume. We leave a full analysis for future work, but we present three brief arguments why this is a reasonable assumption.

All known attacks on SHA1's collision resistance that rely on colliding messages that are many message blocks long and with the attacker determining some bytes of the colliding messages. An attacker attempting to break the soundness of our covenant construction is much more constrained. The attacker neither has more than one or two message blocks to work with, nor can the attacker determine bits of the message blocks.

If an attacker were to achieve collisions under these much more restrictive conditions, we could deploy the hardened SHA1 countermeasure [45] in Small Script. Small Script would detect messages with bit patterns that could result in a collision and reject that input. While not explored in this paper, this level of control over when to allow or not allow a SHA1 collision in Small Script could enable a reduction in the computational cost of our protocol without damaging soundness.

Finally we could make SHA1 collisions between Big Script and Small Script signatures impossible by using RIPEMD to hash the signature rather than SHA1.

7.2 Quantum attacks

Unlike approaches based on functional encryption [39, 25], the security of our approach depends only on the difficulty of colliding ideal hash functions. Even though we use elliptic curve signatures in our construction, we only require they be correct, not that they are sound. Similarly, while Taproot signatures would be forgeable by a quantum computer, commitments to Tapleaves would likely remain strong commitments, meaning that Taproot scripts would continue to be secure.

While giving concrete bounds is outside of the scope of this paper (and likely to depend on the real-world parameters of a quantum computer), we can expect our approach to remain practically secure against quantum attacks.

We also observe that the heavy computational cost in our construction occurs in two places: in a globally one-time precomputation, and at the time that a covenant construction is used. It is therefore very cheap for users to write and deploy covenant scripts if they do not expect to actually use them.

We therefore suggest that users of Bitcoin could use our covenant construction to produce scripts that require a Lamport signature over transaction data in order to be satisfied. While the resulting script will be very large, they can hide it in a Taproot leaf where it will never be serialized on-chain unless it is used. In the case of a "surprise quantum computer" which forces Bitcoin to suddenly disable all elliptic curve cryptography including taproot key spends, such users will still be able to spend their coins (though at enormous cost). If a large quantity of users do this, it may be possible for the Bitcoin chain to survive such an event, even if no better solution is found or deployed.

8 Conclusion

In this paper we showed how to construct a covenant in Bitcoin, without the need of a soft fork. Our construction relied on existing opcodes, such as `OP_SHA1` and `OP_RIPEMD160`, to build an equivalence check between Bitcoin's Big Script (which has access to transaction information) and Small Script (which can perform arbitrary computation on said information).

To achieve this, we built a Bitcoin equivalence tester set $\mathcal{D}$, whose elements can be generated in both Big Script and Small Script. Thus, showing the equivalence of a Big Script element and its Small Script representation amounts to finding hash collisions with elements of the set $\mathcal{D}$. To find collisions, we design a parallel collision finding algorithm based on distinguished points. Fooling our covenant requires finding a 3-way collision with short inputs, which we assume to be hard.

8.1 Future work

We list directions for future work, some of which were mentioned in the introduction.

Small Script hash implementations. In Conjecture 1 we make assumptions on the Small Script cost of SHA1 and RIPEMD. To fully realize our scheme requires these hash functions to be efficiently implemented in Small Script. Moreover, while in Section 5.2 we utilized the somewhat efficient BLAKE3, an even cheaper "Bitcoin-friendly" hash function would greatly reduce the opcode cost of our covenant. Finally, our covenants uses SHA256 in Small Script – reducing the cost of these expensive SHA256 calls greatly reduce transaction size.

Improved cryptographic primitives for Bitcoin. Beyond finding "Bitcoin-friendly" hash functions, as mentioned in the previous paragraph, other improvements to our scheme are likely applicable. A major component in our scheme is the computation of Merkle trees. While we use the standard construction, the improvements shown in [13, 12] could be useful to gain more efficiency in either the off-chain cost or Small Script cost of our covenant. Moreover, considering other techniques to reduce the Small Script cost of generating $\mathcal{D}$'s elements, such as utilizing "Bitcoin-friendly" cryptographic accumulators, is another avenue for future research.

Cryptanalysis of SHA1. This paper treats SHA1 and RIPEMD as ideal 160-bit hash function despite practical attacks on SHA1 [45] that are well below the 2^{80} cost of a generic collision on an ideal 160-bit function. If such attacks were shown to be applicable to our equivalence check, depending on the details they could help or hurt our equivalence check. If we could exploit SHA1 cryptanalytic weaknesses in our collision finding algorithm, we could reduce the work to find needed collisions. In fact, the idea of exploiting SHA1 weaknesses to perform an equivalence check was part of the inspiration that resulted in this paper. On the other hand, if these weaknesses could be used to reduce the cost of finding triple collisions for inputs accepted by our equivalence they could weaken our soundness.

Better time-space trade offs. For our covenant construction, we use a parallel collision finding algorithm based on distinguished points, in similar vein to [49]. Our algorithm exhibits a time-memory trade-off [19], improving upon which is an important future research direction.

The most expensive aspect of our construction is the $> 2^{80}$ off-chain compute cost. We expect possible improvements can come from two directions:

- Non-trivial collision attacks on SHA1 (or RIPEMD), which will lower our covenant off-chain cost (while still keeping finding 3-way collisions infeasible).
- Precomputation methods to reduce the off-chain per-spend cost of the covenant. This is not trivial because we find collisions of the form $f(\pi) = g_{tx}(\rho)$, where f is always the same hash function, while g_{tx} depends on the transaction data (hence only known during runtime).

8.2 Acknowledgements

The authors are indebted to Gil Segev for invaluable discussions in the early stages of this work. We also thank Weikeng Chen, Robin Linus, Neha Narula, Noam Nisan, Jeremy Rubin, and Madars Virza for helpful discussions and feedback.

References

[1] Ark. https://arkdev.info/.

[2] Bitcoin Layers. https://www.bitcoinlayers.org/.

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

[4] BitVM. https://bitvm.org/.

[5] BitVM Github. https://github.com/BitVM/BitVM.

[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. CAT and Schnorr tricks. https://github.com/Bitcoin-Wildlife-Sanctuary/covenants-gadgets.

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

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

[11] D. Christian and J. Rubin. BIP-118 - SIGHASH_ANYPREVOUT, Feb. 2017. github.com/ajtowns/bips/blob/bip-anyprevout/bip-0118.mediawiki.

[12] C. Dhar, Y. Dodis, and M. Nandi. Revisiting collision and local opening analysis of ABR hash. In D. Dachman-Soled, editor, *3rd Conference on Information-Theoretic Cryptography, ITC 2022, July 5-7, 2022, Cambridge, MA, USA*, volume 230 of *LIPIcs*, pages 11:1–11:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[13] Y. Dodis, D. Khovratovich, N. Mouha, and M. Nandi. T5: hashing five inputs with three compression calls. *IACR Cryptol. ePrint Arch.*, page 373, 2021.

[14] T. Giladi. https://github.com/TomerStarkware/BitVM/tree/tomer/main.

[15] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions (extended abstract). In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 464–479. IEEE Computer Society, 1984.

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

[17] E. Heilman. Signing a bitcoin transaction with lamport signatures (no changes needed), 2024. groups.google.com/g/bitcoindev/c/mR53go5gHIk.

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

[19] M. Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980.

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

[21] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In M. K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[22] 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.

[23] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

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

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

[26] O. Kuznetsov, A. Rusnak, A. Yezhov, K. Kuznetsova, D. Kanonik, and O. Domin. Merkle trees in blockchain: A study of collision probability and security implications. *Internet of Things*, page 101193, 2024. arxiv.org/pdf/2402.04367.

[27] j. Lau. Op_pushtxdata, 2017, 2017. github.com/jl2012/bips/blob/vault/bip-0ZZZ.mediawiki.

[28] 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.

[29] G. Leurent and T. Peyrin. {SHA-1} is a shambles: First {Chosen-Prefix} collision on {SHA-1} and application to the {PGP} web of trust. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1839–1856, 2020. www.usenix.org/conference/usenixsecurity20/presentation/leurent.

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

[31] 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.

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

[33] 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.

[34] J. OBeirne, G. Sanders, and A. Towns. BIP-345 - OP_VAULT, Feb. 2023. github.com/bitcoin/bips/blob/master/bip-0345.mediawiki.

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

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

[37] A. Poelstra. Script State from Lamport Signatures, May 2024. https://bitcoinmagazine.com/technical/script-state-from-lamport-signatures-.

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

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

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

[41] R. Russell. The Great Script Restoration Project, May 2024. www.youtube.com/watch?v=rSp8918HLnA.

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

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

[44] 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.

[45] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. In *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part I 37*, pages 570–596. Springer, 2017.

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

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

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

[49] P. C. Van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of cryptology*, 12:1–28, 1999. people.scs.carleton.ca/ paulv/papers/JoC97.pdf.

[50] N. van Saberhagen. CryptoNote v2.0, Oct 2013. https://diyhpl.us/ bryan/papers2/bitcoin/cryptonote.pdf.

[51] P. Wuille, J. Nick, and T. Ruffing. BIP-340 - Schnorr Signatures for secp256k1, Janary 2020. github.com/bitcoin/bips/blob/master/bip-0340.mediawiki.

[52] 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.

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

[54] 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.

A Big Script and Small Script Opcodes

Throughout this paper we have divided Bitcoin Script into two subsets, Small Script and Big Script, with the distinction that the former is able only to operate on 32-bit (4-byte) values. In this Appendix we make this distinction precise, by listing every opcode of Script and describing the size of its inputs and outputs.

First, each opcode is one byte in size. Of the 256 possible values for a byte, 100 are "failure" or "no-op" opcodes. If present in a script, they have no effect (except possibly to invalidate the transaction), so for our purposes we can ignore them.

Of the remaining 156,

- 96 are "push" opcodes which simply push data onto the stack. In Small Script, these can be used to push data of up to 4 bytes in size.
- 17 are "stack manipulation opcodes" (e.g. `OP_2DROP` or `OP_ROT`). These simply rearrange stack elements regardless of size, and therefore can be executed either in Big Script or Small Script.
- The `OP_IFDUP` deserves a category of its own, since it takes an arbitrarily-sized input, interprets it as a boolean (which means that every bit except the most-significant one is 0), and duplicates its input if so.
- 5 are "control flow opcodes" (`OP_IF`, `OP_NOTIF`, `OP_ELSE`, `OP_ENDIF`, `OP_VERIFY`) and can be executed in either context.

  (Strictly speaking, `OP_IF` and `OP_NOTIF` are legal only with inputs of 0 or 1. But it is easy to transform an input of any size into this form, e.g. by `OP_SIZE OP_ONOTEQUAL`.)

For the remaining 38 opcodes, we look more carefully at their allowed input sizes and possible output sizes. We list them in Table 1.

In the table, a dash indicates a 0-sized input or output, and "Small*" indicates an opcode that may output a 5-byte value but typically outputs a Small Script value. (`OP_CLTV` and `OP_CSV` simply output their input unchanged). It is always possible to detect these cases in Script and handle them separately, so these opcodes can be considered to have Small Script outputs.

For a given script, we say that it is a Small Script while the elements on the stack during its execution are 4 bytes or smaller, and call it a Big Script otherwise. In general, we cannot say that an

entire script is a "Big Script" or "Small Script", but we can say whether the execution is currently "in Big Script" or "in Small Script", and say which opcodes are permissible in this context.

As we see from the table, the 5 signature-checking opcodes cannot be used with Small Script inputs (and in fact, cannot be used with the outputs of *any* opcodes except push opcodes, since the required input sizes do not match the output sizes of any other opcodes). We further see that the 5 hashing opcodes, whose outputs are marked "Big,", can be used in any context, but once used, will leave the script in a Big Script context.

Finally, we see that the opcodes `OP_SIZE`, `OP_EQUAL`, `OP_EQUALVERIFY`, and `OP_NOT` can be used to move from Big Script back to Small Script. (In Taproot it is impermissible to use `OP_NOT` in this way, but it can be simulated with `OP_IFDUP` and `OP_SIZE`; see above.) However, these opcodes all lose all or most of the information available from the Big Script stack elements, and therefore cannot be used to bridge Big Script and Small Script.

B Schnorr SigHash

The data to be signed in Bitcoin's Taproot Schnorr signature implementation [52, 53] takes as input the contents of the surrounding transaction as well as a *sighash flag*, which is appended to the serialized signature and determines which parts of the transaction are to be signed.

There are two variants of the signature hash: one used in "key spends" which is used when no script is provided, and one used by "script spends" which are signature checks invoked from within a script. Only the latter is relevant for our purposes, so we will disregard keyspends from now on.

The data to be signed is hashed using a BIP-0340 tagged hash [51] defined as

$$\text{Hash}_{\text{TapSighash}}(x) = \text{SHA256}(\text{SHA256}(\texttt{TapSighash})\|\text{SHA256}(\texttt{TapSighash})\|x)$$

where here $\|$ indicates concatenation and `TapSighash` is the literal ASCII encoding of that word.

The exact data which is signed is then

$$M = \text{Hash}_{\text{TapSighash}}(\texttt{0x00}\|\text{SigMsg}(\text{hash\_type}, 1)\|\text{ext})$$

where the constant `0x00` is fixed and the constant 1 indicates that this is a script spend. The function SigMsg(hash_type, 1) is in turn defined as the serialization of:

- hash_type (1 byte);
- Transaction version (4 bytes, little endian);
- Transaction locktime (4 bytes, little endian);
- If the high bit of hash_type is **unset**,
  - sha_prevouts, the SHA256 hash of all input outpoints;
  - sha_amounts, the SHA256 hash of all input amounts;
  - sha_scriptpubkeys, the SHA256 hash of all input scriptPubKeys (serialized as script with length prefix);
  - sha_sequences, the serialization of all input sequence numbers (4 bytes each, little endian);
- If the lowest 2 bits of hash_type are 0 or 1,
  - sha_outputs, the serialization of all transaction outputs;
- `0x03` if an annex (auxiliary data to be signed is present; otherwise `0x02`;
- If the high bit of hash_type is **set**,
- sha_outputs, the serialization of all transaction outputs;
  - The outpoint of the input being signed for;
  - The amount of the input being signed for;

Table 1: Bitcoin Script opcodes and their input and output sizes. A dash indicates a 0-sized input or output, and "Small*" indicates an opcode that may output a 5-byte value but typically outputs a Small Script value. (`OP_CLTV` and `OP_CSV` simply output their input unchanged).

Opcode	Input Size	Output Size
`OP_DEPTH`	–	Small
`OP_SIZE`	Any	Small
`OP_EQUAL`	Any	Small
`OP_EQUALVERIFY`	Any	–
`OP_1ADD`	Small	Small*
`OP_1SUB`	Small	Small*
`OP_NEGATE`	Small	Small*
`OP_ABS`	Small	Small
`OP_NOT`	Any	Small
`OP_ONOTEQUAL`	Small	Small
`OP_ADD`	Small	Small*
`OP_SUB`	Small	Small*
`OP_BOOLAND`	Small	Small
`OP_BOOLOR`	Small	Small
`OP_NUMEQUAL`	Small	Small
`OP_NUMEQUALVERIFY`	Small	–
`OP_NUMNOTEQUAL`	Small	Small
`OP_LESSTHAN`	Small	Small
`OP_GREATERTHAN`	Small	Small
`OP_LESSTHANOREQUAL`	Small	Small
`OP_GREATERTHANOREQUAL`	Small	Small
`OP_MIN`	Small	Small
`OP_MAX`	Small	Small
`OP_WITHIN`	Small	Small
`OP_CODESEPARATOR`	–	–
`OP_CLTV`	Small*	Small*
`OP_CSV`	Small*	Small*
`OP_RIPEMD160`	Any	Big (20)
`OP_SHA1`	Any	Big (20)
`OP_SHA256`	Any	Big (32)
`OP_HASH160`	Any	Big (20)
`OP_HASH256`	Any	Big (32)
`OP_CHECKSIG`	Big (32, 64)	Small
`OP_CHECKSIGVERIFY`	Big (32, 64)	–
`OP_CHECKSIGADD`	Big (32, 64)	Small
`OP_CHECKMULTISIG`	Big	Small
`OP_CHECKMULTISIGVERIFY`	Big	Small

– The scriptPubKey of the input being signed for;
– The sequence number of the input being signed for;

We observe that the fields sha_amounts and sha_scriptpubkeys, while present in the signature hash, are not present in the transaction itself. (Conversely, the witness data of the transaction is not present in the signature hash.) Therefore, when we refer to the signature hash as the "transaction data" this is actually a slight abuse of terminology.

Definition 7. In the main body, we'll write, for a transaction tx,

$$\text{MsgHash}(tx) = M = \text{Hash}_{\text{TapSighash}}(\texttt{0x00} \| \text{SigMsg}(\text{hash\_type}, 1) \| \text{ext})$$

to refer to the transaction serialization and hashing procedure defined above.

With the signature hash M defined as above, we can describe the BIP-0340 Schnorr signature algorithm. First, we need another tagged hash:

$$\text{Hash}_{\text{BIP0340/challenge}}(x) = \text{SHA256}(\text{SHA256}(\texttt{BIP0340/challenge}) \| \text{SHA256}(\texttt{BIP0340/challenge}) \| x)$$

where as above, the text is ASCII-encoded.

Then a BIP-0340 signature with public key $P = xG$ is defined by:

$$\begin{aligned} k &\leftarrow_{\text{uniform}} \{0,1\}^{256} \\ R &\leftarrow kG \\ e &\leftarrow \text{Hash}_{\text{BIP0340/challenge}}(R \| P \| M) \\ s &\leftarrow k + ex \\ \sigma &\leftarrow R \| s \end{aligned}$$

We assuming no sighash flag is attached to the signature. This causes Bitcoin to use the flag `SIGHASH_DEFAULT`, which sets hash_type to 0, signing all inputs and all outputs of the transaction.

Definition 8. In the Schnorr signature above, we define $\text{SchnorrHash}^{R,P}$ as the transaction serialization and hashing to produce the message for the signature algorithm. That is, we define, for a transaction tx,

$$\text{SchnorrHash}^{R,P}(tx) = \text{Hash}_{\text{BIP0340/challenge}}(R \| P \| M)$$

as above.

B.1 Schnorr Trick

The Schnorr trick [35, 36] uses the above construction, but chooses the secret key x and secret nonce k both to be 1, rather than uniformly random. The above equation then reduces to

$$\begin{aligned} s &\leftarrow 1 + \text{Hash}_{\text{BIP0340/challenge}}(R \| P \| M) \\ \sigma &\leftarrow R \| s \end{aligned}$$

Which can be verified by using `OP_CAT` and `OP_EQUALVERIFY` to ensure that $R = G$, directly setting $P = G$, then using `OP_CHECKSIGVERIFY` to verify that the above equation holds.

Then, by using `OP_CAT` and `OP_SHA256` to recompute M and $\text{Hash}_{\text{BIP0340/challenge}}(R \| P \| M) + 1$, transaction data can be extracted from M.

In the original Schnorr trick, the addition by 1 required special consideration. In particular, it assumed that only the last byte of the signature would be affected by the addition, and used `OP_CAT` to separate and ignore this last byte (since when comparing 256-bit hashes for equality, ignoring a single byte will not meaningfully affect security).

However, this assumption fails to hold for 1/256 of possible transaction hashes. The original Schnorr trick addressed this by asking the user to grind their signatures to ensure this was not the case.

This amount of grinding is very small (requiring no extra signatures at all 255/256 of the time), and can be made smaller by ignoring more suffix bytes. But for our purposes, any amount of grinding would complicate our algorithm and potentially make an already-expensive process much more expensive. We therefore address the addition by 1 differently: rather than assuming it affects only a single byte, and ignoring it, we address it head-on by implementing a 256-bit addition in Small Script.

C Ring Signatures Without Blake3 Merkle Trees

In Section 2.2.5 we described the "Tapleaf tragedy," which is that we cannot use Taproot trees as lookup tables for transaction signatures, since transaction signatures change uncontrollably when we change which branch we reveal. This means that to implement covenants via a small-to-big equivalence check on signatures, we must implement Merkle tree lookups directly in Script. We used BLAKE3 for this purpose since it is the most efficiently implementable in Script.

In Remark 2 we observed that nearly all of the on-chain cost of our covenant construction could be avoided, if not for the Tapleaf tragedy.

However, while signatures are the only type of object visible to Big Script which we might want to manipulate using Small Script. There are also public keys, which are a fixed size (32 bytes in Taproot script).

To illustrate how this might be useful, we describe a ring signature construction. We assume we have access to a zero-knowledge proof construction for set membership which can be implemented for secret signing keys in Small Script. For example, one could use a linkable ring signature [50] for which the keypairs were of the form (x, xH) with H a fixed nonstantand generator for the secp256k1 curve, and whose key images were of the form xG, where G is the standard generator. By ring-signing the empty string, a user identified by a ring signature key in the set $\{x_1H, \ldots, x_nH\}$ can prove that a transaction signing key x_iG uses some x_i from the set without revealing each one.

Now, even assuming a ring signature primitive, it is not possible to directly ring-sign a Bitcoin transaction. The reason, as always, is that our ring signature is implemented in Small Script, which does not have access to transaction data. We could provide transaction data to Small Script by using the covenant construction of this paper, but there is a more efficient way: rather than using the small-to-big equivalence check to feed transaction data from Big Script to Small Script, we use the equivalence check to feed a *public key* from Small Script to Big Script. By applying the equivalence check to a public key rather than signature, we can do (part of) our Merkle tree lookup using Taproot trees rather than expensive Small Script construction.

In more detail, our construction is as follows:

Let G be the the standard generator of the secp256k1 elliptic curve, and H an alternate nothing-up-my-sleeve (NUMS) generator. Then, a set of users $\{U_i\}_i$ generate random secret keys x_i and publish public keys x_iH. Using the set of keys $\{x_iH\}_i$ they construct a Bitcoin script which:

- First, checks a linkable ring signature $\langle\sigma\rangle_{32}$ on an empty message against key image $\langle I\rangle_{32}$, where the ring signature passes if and only if $I = x_iG$ for some x_i.
- Then, does a small-to-big equivalence check between a proposed public key I and $\langle I\rangle_{32}$
- Then calls `OP_CHECKSIG` using the public key I to verify a signature on the transaction itself.

Essentially, we are using the linkable ring signature "backward," exchanging the roles of key images and public keys. By validating the ring signature in Small Script, we obtain a key image whose discrete logarithm is equal to that of one of the ring signature keys. We then use the small-to-big equivalence check to move this key image into Big Script, where it is interpreted as an ordinary signing key for a Bitcoin transaction.

The above approach is a rough sketch; it is unlikely to be the most efficient construction, and does not attempt to preserve privacy across transactions (e.g. by making H be unique per transaction). But it illustrates the potential of this approach.

D Proof of Proposition 3

To prove the last part of Proposition 3, we'll need the following lemmas.

Lemma 1 (Poisson approximation of balls and bins)**.** *Suppose you throw m balls into n bins, each ball independently landing on a bin uniformly at random. Let $\mathcal{E}_m$ be a* monotone event *whose indicator random variable is a function of $f(L_1, \ldots, L_n)$, where L_i is the number of balls in bin i. Consider a second experiment where we choose n* independent *random variables $Z_1, \ldots, Z_n$, where each $Z_i \sim Pois(\lambda)$ follows a Poisson distribution with parameter $\lambda = m/n$. Then,*

$$\Pr[f(L_1, \ldots, L_n)] \leq 2 \cdot \Pr[f(Z_1, \ldots, Z_n)].$$

Here, we say that $\mathcal{E}_m$ is a *monotone event* if it always holds that either $\Pr[\mathcal{E}_m] \leq \Pr[\mathcal{E}_{m'}]$ or $\Pr[\mathcal{E}_m] \geq \Pr[\mathcal{E}_{m'}]$ for $m' < m$. The events we are interested in will be monotone.

Lemma 2 (Chernoff's inequality)**.** *Suppose that $X_1, \ldots, X_n$ are independent random variables taking values in $\{0,1\}$. In addition, let $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbb{E}[X]$. Then for any $\delta > 0$ it holds that*

$$\Pr\left[X \leq (1-\delta)\mu\right] < \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu}.$$

In particular, for $\delta = 2^{-20}$, we have that

$$\Pr\left[X \leq (1-\delta)\mu\right] < \left(1 - 2^{-41}\right)^{\mu}.$$

Proof of Proposition 3. The claim about evaluating $\mathcal{D}.\mathsf{Gen}_{\mathcal{B}}$ is trivial, and the claim about evaluating $\mathcal{D}.\mathsf{Gen}_{\mathcal{S}}$ follows from Conjecture 1.

Now, for a function f and a set S, define $f(S) = \{f(s) | s \in S\}$. To bound $|\mathcal{D}|$, we make the following definitions:

$$\begin{aligned}
L_0 &= \{0,1\}^{33} \\
L_1 &= \mathrm{SHA1}(L_0) \cup \mathrm{RIPEMD}(L_0) \\
L_2 &= (\mathrm{SHA1}(L_1) \cup \mathrm{RIPEMD}(L_1)) \setminus L_1 \\
L_3 &= (\mathrm{SHA1}(L_2) \cup \mathrm{RIPEMD}(L_2)) \setminus (L_2 \cup L_1) \\
&\vdots \\
L_{||t||} &= \left(\mathrm{SHA1}(L_{||t||-1}) \cup \mathrm{RIPEMD}(L_{||t||-1})\right) \setminus \bigcup_{j=1}^{||t||-1} L_j
\end{aligned}$$

where, by definition, $|\mathcal{D}| \geq |L_{||t||}|$. Our strategy will be to prove that $|L_i|$ is sufficiently large for each i. A subtlety arises in that we work in the random oracle model. Thus, for simplicity of analysis we'd like to require that all L_i and L_j are sampled *independently*, whenever $i \neq j$, as otherwise, we might visit values that were already queried to the random oracle. This is why in the construction we required that L_i is *disjoint* from $L_{i-1}, L_{i-2}, \ldots, L_1$.

To this end, for some $\{\alpha_i\}_i$ to be specified later, we'll define a sequence of events

$$\begin{aligned}
\mathcal{B}_1 &: |L_1| = 2^{34} \\
\mathcal{B}_2 &: |L_2| = 2^{35} \\
&\vdots \\
\mathcal{B}_{34} &: |L_{34}| = 2^{77} \\
\mathcal{B}_{35} &: |L_{35}| \geq 2^{78}\alpha_{35} \\
&\vdots \\
\mathcal{B}_{||t||} &: |L_{||t||}| \geq 2^{33+||t||}\alpha_{||t||}.
\end{aligned}$$

Since it holds that

$$\Pr\left[|\mathcal{D}| \geq 2^{33+||t||}\alpha_{||t||}\right] \geq \Pr\left[B_{||t||}\right],$$

we'll bound the probability of the latter from below. Starting by bounding $\mathcal{B}_i$ for $i = 1$, it holds that

$$\Pr\left[\mathcal{B}_1\right] \geq e^{-2^{2\cdot 34-160}} \geq 1 - 2^{-92},$$

where the inequality follows by the usual birthday bound. For $i = 2$, define

$$\widetilde{L_2} = \mathrm{SHA1}(L_1) \cup \mathrm{RIPEMD}(L_1).$$

By using the union bound, we get that

$$\Pr\left[\mathcal{B}_2 | \mathcal{B}_1\right] \geq 1 - \Pr\left[|\widetilde{L_2}| < 2^{35} \middle| \mathcal{B}_1\right] - \Pr\left[\widetilde{L_2} \cap L_1 \neq \emptyset \middle| \mathcal{B}_1\right].$$

Bounding these quantities we get that

$$\Pr\left[|\widetilde{L_2}| < 2^{35} \middle| \mathcal{B}_1\right] \leq 2^{2\cdot 35-160} = 2^{-90},$$
$$\Pr\left[\widetilde{L_2} \cap L_1 \neq \emptyset | \mathcal{B}_1\right] \leq \left(1 - 2^{34-160}\right)^{2^{35}} \leq 2^{-90},$$

where the latter inequality uses the fact that $|L_1|$ is at most 2^{34}, so in the random oracle model, we're essentially throwing 2^{35} balls into 2^{160} bins, hoping that they all miss the $|L_1| \leq 2^{34}$ bins. Hence

$$\Pr\left[\mathcal{B}_2 | \mathcal{B}_1\right] \geq 1 - 2^{-89}.$$

Continuing with a similar analysis for $i = 3, \ldots, 34$ (where we define $\widetilde{L_i}$ similarly to how we defined $\widetilde{L_2}$), we get that

$$\Pr\left[|\widetilde{L_i}| < 2^{33+i} \middle| \mathcal{B}_{i-1}\right] \leq 2^{2\cdot(33+i)-160} \leq 2^{-26},$$
$$\Pr\left[\widetilde{L_i} \cap \bigcup_{j=1}^{i-1} L_j \neq \emptyset | \mathcal{B}_{i-1}\right] \leq \left(1 - 2^{33+i-160}\right)^{2^{34+i}} \leq 2^{-26},$$

Hence, for $i = 3, \ldots, 34$, it holds that

$$\Pr\left[\mathcal{B}_i | \mathcal{B}_{i-1}\right] \geq 1 - 2^{-25}.$$

For $i \geq 35$ a different strategy is needed. To bound $\Pr\left[|L_i| < 2^{33+i}\alpha_i \middle| \mathcal{B}_{i-1}\right]$, we'll apply Lemma 1. This implies there are independent Bernoulli random variables $X_j \sim Ber\left(1 - e^{-2^{33+i-160}}\right)$ such that

$$\Pr\left[|L_i| < 2^{33+i}\alpha_i \middle| \mathcal{B}_{i-1}\right] \leq 2 \cdot \Pr\left[\sum_{j=1}^{2^{160}-2^{33+i}} X_i < 2^{33+i}\alpha_i\right],$$

where X_j equals 1 if bin j is nonempty in round i. In total there are 2^{160} bins, but we exclude 2^{33+i} of them, which are potentially occupied by values from $L_1, \ldots, L_{i-1}$. Applying Lemma 2, we get that

$$\Pr\left[\sum_{j=1}^{2^{160}-2^{33+i}} X_j \leq (1-\delta)\mu\right] < \left(1 - 2^{-41}\right)^{\mu},$$

where $\mu = (2^{160} - 2^{33+i})\left(1 - e^{-2^{33+i-160}}\right)$. To make it applicable to the previous bound we choose

$$\alpha_i = \frac{(1-\delta)\mu}{2^{33+i}},$$

which satisfies $|\alpha_i - 1| \leq 2^{-15}$. Since $\mu \geq 2^{32+i} \geq 2^{67}$, it holds that

$$\Pr\left[\sum_{j=1}^{2^{160}-2^{33+i}} X_j \leq (1-\delta)\mu\right] < \left(1 - 2^{-41}\right)^{2^{67}} \leq 2^{-1,000,000},$$

meaning we can mostly neglect it. In total we get that

$$\Pr[|\mathcal{D}| > (1 - 2^{-15})2^{||\omega||+||t||}] \geq \Pr[\mathcal{B}_1] \cdot \prod_{j=2}^{||t||} \Pr[\mathcal{B}_j | \mathcal{B}_{j-1}] \geq \left(1 - 2^{-25}\right)^{35} \geq 1 - 2^{-19}.$$

□

E Hash function collisions

Here we list some basic results on hash function collisions.

E.1 Expected collisions per Number of queries

Here we give the equation for the expected number of collisions, $E[C]$, after $Q = 2^q$ queries to a hash function whose output size is $N = 2^n$, *i.e.,* , n bit length output. The probability of any two queries colliding is: 2^{-n}. The number of possible collisions for Q queries is $\binom{Q}{2} = \frac{Q!}{2!(Q-2)!} = \frac{Q^2-Q}{2} = \frac{2^{2q}-2^q}{2}$. Thus, the expected number of collisions is the probability of a collision times the number of possible ways a collision count occur:

$$E[C] = 2^{-n}\binom{Q}{2} = \frac{2^{2q} - 2^q}{2^{n+1}} = 2^{2q-n-1} - 2^{q-n-1}$$

We can drop the last part of this equation 2^{q-n-1} as it is insignificant for size of numbers we are using.:

$$E[C] \approx 2^{2q-n-1}$$

E.2 Collisions between two hash functions

Here we derive the expected collisions and probability of one or more collisions between two n bit hash functions, $\mathtt{H}_a, \mathtt{H}_b$ after 2^a queries to $\mathtt{H}_a$ and 2^b queries to $\mathtt{H}_b$.

We are only concerned with collisions between the two hash functions, that is $\mathtt{H}_a(x) = \mathtt{H}_b(x')$ where $x \neq x'$. We do not count collisions that occur within the same hash function $\mathtt{H}_a(x) = \mathtt{H}_n(x')$.

The expected number of collisions between them $\mathtt{H}_a(x) = \mathtt{H}_b(x')$ after 2^a queries to $\mathtt{H}_a$ and and 2^b queries to $\mathtt{H}_b$ is:

$$2^a \times 2^b \times \frac{1}{2^n} = \frac{2^{a+b}}{2^n} = 2^{a+b-n}$$

The probability of at least one collision between $\mathtt{H}_a$ and $\mathtt{H}_b$ is:

$$1 - (1 - \frac{1}{2^n})^{2^{a+b}} \approx 1 - e^{-(2^{a+b-n})}$$

F Time and space cost of our distinguished points collision algorithm

We define the hash functions f and g take $\rho \in \{0,1\}^{160}$ as input:

$$\begin{array}{ll} f(\rho) : \{0,1\}^{||w||+||t||} \rightarrow \{0,1\}^{160} & \text{is } \mathcal{D}.\mathsf{Gen}(\rho\,|_1^{||\omega||}, \rho\,|_{||\omega||+1}^{||\omega||+||t||+1}) \\ g(\rho) : \{0,1\}^{160} \rightarrow \{0,1\}^{160} & \text{SHA1}(G||(\text{SchnorrHash}(\text{TxGrind}(tx, \rho)) + 1)) \end{array}$$

We define the following variables as:

z: The number of zeros that identifies a point as a distinguished point.
qf: The number of unique queries to f is 2^{qf}
qg: The number of unique queries to g is 2^{qg}

The probability of a *useful collision i.e.,* a collision allows us to spend a covenant, is:

$$1 - e^{-2^{qf+qg-160}}$$

We have greater than $1/2$ probability of a useful collision when $qf + qg \geq 160$. As we will show below this does not tell us the total work we need to do. This is because not every query to f or g is unique due to *useless collisions i.e.,* collisions that are not useful for finding a covenant.

F.1 Distinguished Point Table Size

We define our distinguished points as an output which has at least z zero bits at the end. This means that 2^{-z} is probability of a query to the hash function producing a distinguished point and 2^z is the number of queries between two distinguished points.

A naive approach would be for each row on the distinguished point table to include the distinguished point (size $160/8 = 20$ bytes), and the prev distinguished point. As the distinguished points are added to the table in as they are discovered, the previous distinguished point can simply be the previous address in memory, letting us reference it implicitly without using any space. Since all distinguished points end in z we can truncate the z all zero bits. Thus a row in our distinguished point table is $(160 - z)/8$ bytes.

If the total number of unique queries we make is $Q = 2^q$, then distinguished point table will have 2^{q-z} rows, the table will use space:

$$2^{q-z} \times (20 - z/8)$$

F.2 The Useless Collisions Problem

Collisions such as $f(\rho) = f(\rho')$ or $g(\rho) = g(\rho')$ do not help us show equivalence. We term such collisions, useless collisions. Only collisions between f and g *e.g.,* $f(\rho) = g(\rho')$ are useful for showing equivalence.

Each time we hit a collision it takes us 2^{z-1} additional hash queries before we hit reach a distinguished point in the distinguished point table. Then determining where the collision occurred to determine if the collision is a useless collision or not requires an additional 2^{z-1} work. Thus, each useless collision we encounter costs an additional $2^{z-1} + 2^{z-1} = 2^z$ work.

Pseudo-collisions are collisions that arise we not all the bits of the input to a hash function influence the output. In our case pseudo-collisions arise when $||w|| + ||t||$ is less than ρ, as f can not fit all the bits of ρ into f causing a truncation of ρ. As the size of ρ is 160 then we encounter pseudo-collisions in f when $||w|| + ||t|| < (||rho|| = 160)$ *i.e.,* the domain of f, $\{0,1\}^{||w||+||t||}$, is smaller than $\rho \in \{0,1\}^{160}$.

$$f(\rho) = f(\rho'); \rho \neq \rho'; \rho \mid_0^{||w||+||t||} = \rho' \mid_0^{||w||+||t||}$$

Pseudo-collisions presents a serious problem of wasted work for our collision finding algorithm because the size constraints of Small Script forces us to significantly truncate ρ in f. The expected number of f to f collisions, including pseudo-collisions, for 2^{qf} queries to f is (See Appendix E.1):

$$2^{2qf-||w||-||t||-1}$$

We can also compute the expected number of g to g useless collisions for 2^{qf} queries to g as:

$$2^{2qg-160-1}$$

As $||w|| + ||t||$ is likely be much small than 160 the number of f to f collisions is likely to much larger than the number of g to g useless collisions if $qf = qg$. We can reduce the overall number of useless collisions by making g more frequent than f. The total number of useless collisions is given by:

$$2^{2qf-||w||-||t||-1} + 2^{2qg-160-1}$$

As shown earlier, for a greater $1/2$ probability of finding a useful collision, $qf + qg \geq 160$. Whatever we subtract from qf, we must add to qg.

Now we look at much work is wasted for each useless collision. Each time we hit a collision it takes us 2^{z-1} additional hash queries before we hit reach a distinguished point in the distinguished point table. Then we need to determining the preimage of the collision to determine if the collision is a useless collision or not requires an additional 2^{z-1} work. Thus, each useless collision we encounter costs an additional $2^{z-1} + 2^{z-1} = 2^z$ work.

The total wasted work, is:

$$2^z \times (2^{2qf-||w||-||t||-1} + 2^{2qg-160-1}) = 2^{2qf+z-||w||-||t||-1} + 2^{2qg+z-160-1})$$

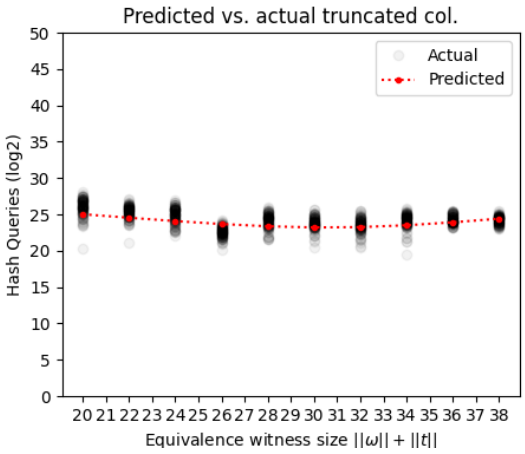

Figure 9: Our equation for the number of queries need to find a collision for our collision finding algorithm. Rather than a 160-bit hash functions, we use 40-bit hash functions and compare this against an implementation of our collision finding algorithm. We use $||\omega|| = 10$ and $z = 3$. We run the attack 100 times per position.

F.3 Additional Work in f

The hash function f calls the function $\mathcal{D}.\mathsf{Gen}$ which will make an additional hash function query for each bit in t for $||t||$ additional hash function calls. It is not unreasonable to view $||t||$ as a constant and treat $\mathcal{D}.\mathsf{Gen}$ a single hash function call for the purpose of computing work. In our setting we are considered with practical collisions, and want to choose parameters so we include the number of hash function calls in $\mathcal{D}.\mathsf{Gen}$ as additional queries. Thus, the additional cost of a call to f vs g is $||t||$.

F.4 Time and Space for a Useful Collision

The number of unique queries is including the additional work in f is:

$$2^{qf} \times (||t||) + 2^{qg} = 2^{qf+\log_2 ||t||} + 2^{qg}$$

The amount of wasted work we have to do is:

$$2^{2qf+\log_2 ||t||+z-||w||-||t||-1} + 2^{2qg+z-160-1}$$

The total amount of time work to find a useful collision is $2^{qf+\log_2 ||t||}+2^{qg}+2^{2qf+\log_2 ||t||+z-||w||-||t||-1}+2^{2qg+z-160-1}$ where $qf + qg \geq 160$. For $qf + qg = 160$ we can simplify this by defining qg as $qg = 160 - qf$:

$$\text{Work}(qf, ||t||, ||w||, z) = 2^{qf+\log_2 ||t||} + 2^{160-qf} + 2^{2qf+\log_2 ||t||+z-||w||-||t||-1} + 2^{2(160-qf)+z-160-1}$$

The space requirements are:

$$(20 - z/8) \times (2^{qf-z} + 2^{(160-qf)-z})$$

F.5 Tests on truncated outputs

To validate our math we implemented our collision attack against hash functions truncated to an 40-bit output. In Figure 9 we graph the result of our experimental runs of this collision attack against our work equation.

This requires generalizing our work equation above for n-bit outputs

$$\text{Work}(qf, ||t||, ||w||, z) = 2^{qf+\log_2 ||t||} + 2^{n-qf} + 2^{2qf+\log_2 ||t||+z-||w||-||t||-1} + 2^{2(n-qf)+z-n-1}.$$

For two 48-bit hash functions, $z = 6$, $||\omega|| = 10$, $||t|| = 20$, $qf = 2^{-7}$, $qg = 1$

$$2^{qf+\log_2 20} + 2^{40-qf} + 2^{2qf+\log_2 20+z-10-20-1} + 2^{2(40-qf)+6-40-1}$$

G Attacks on our Equivalence Checks

In this appendix we provide a detailed walk through of the best attacks we have developed against our equivalence check. To break our equivalence check an attacker must break soundness, *i.e.,* construct an input

$$(\langle tx \rangle_{32}, \pi, R_1||s_1, \langle R_2||s_2 \rangle_{32}; spender\_tx)$$

to our Tapscript equivalence check which passes the equivalence check but the Big Script input, $R_1||s_1$, and the Small Script input, $R_2||s_2$ are not in fact equivalent, $R_1||s_1 \neq R_2||s_2$. This necessarily requires a finding a triple collision:

$$\text{SHA1}(R_1||s_1) = \mathcal{D}.\mathsf{Gen}(\pi) = \text{SHA1}(R_2||s_2)$$

We present two potential attacks that aim to find a triple collision and break soundness. We treat SHA1 and RIPEMD as 160-bit ideal hash functions. For a discussion of the known weaknesses of SHA1 in see Section 8.

Approach 1 (fix and search):

1. Find $R_1||s_1$ and π such that $\text{SHA1}(R_1||s_1) = \mathcal{D}.\mathsf{Gen}(\pi)$ using our collision finding algorithm.
2. Then attempt to transform this into a triple collision by finding a value $R_2||s_2$ such that $\text{SHA1}(R_2||s_2) = \mathcal{D}.\mathsf{Gen}(\pi)$ and $R_1||s_1 \neq R_2||s_2$.

Since π is fixed, finding $\text{SHA1}(R_2||s_2) = \mathcal{D}.\mathsf{Gen}(\pi)$ is finding the second preimage of $R_1||s_1$ which requires 2^{160} queries assuming the underlying hash function is secure.

Approach 2 (Meet-in-the-Middle):

1. Generate $P = 2^p$ unique pairs of signatures $(R_1||s_1, R_2||s_2), (R_3||s_4, R_4||s_4), \ldots$ such that each pair collides $\text{SHA1}(R_i||s_i) = \text{SHA1}(R_j||s_j)$. The signature on the left will be accepted by the covenant, the signature on the right would be rejected by the covenant.
2. Then try to find a π that collides with one of these pairs such that

$$\text{SHA1}(R_i||s_i) = \text{SHA1}(R_j||s_j) = \mathcal{D}.\mathsf{Gen}(\omega, t)$$

We first analyze the cost of this attack against an attacker that has infinite space. This allows the attacker to use a lookup table to find collisions in 160-bit hash functions in only 2^{80} queries. We follow this analysis with a more realistic analysis in which the attacker has space constraints and must use distinguished points collision algorithm.

Attacker with infinite space. We consider an attacker with infinite space that is concerned only with the time cost. In step one finding $2^p = 2^{2q-160-1}$ pairs of colliding signatures requires $2^q = 2^{(p+161)/2}$ queries.

$$2^p = 2^{2q-160-1}$$

$$p = 2q - 161$$

$$(p + 161)/2 = q$$

We denote the number of queries we make in step 1 as 2^{q_1} and the number of queries we make in step 2 as $2^{q_2+\log_2(||t||)}$. The $\log_2(||t||)$ in step 2 is the result of the $||t||$ additional queries that are made to perform a query to $\mathcal{D}.\mathsf{Gen}$. To succeed with greater than 1/2 probability the attacker must make:

$$2^{q_1} + 2^{q_2+\log_2(||t||)}$$

queries, such that $q_2 \leq ||\omega|| + ||t||$ and the following inequality holds:

$$\frac{2^{2q_1-161} \cdot 2^{q_2}}{2^{160}} \geq 1/2$$

$$2^{2q_1-161+q_2-160} \geq 1/2$$

Simplifying the inequality we get:

$$2q_1 - 161 + q_2 \geq 159$$

$$2q_1 + q_2 \geq 320$$

Thus to find the work the attacker needs to do, we find a q_1, q_2 which minimizes the number of queries, which satisfying the two conditions.

The above gives us the time cost of our best attack on our equivalence-check, but in the context of covenants the number of queries in step 1 doubles. This is because to cheat a covenant, you need one of the colliding signatures to pass the covenant's rules and the other colliding signatures to be rejected by the covenant. Since only half of these collisions generated in step 1 can be used to break our covenant, this requires twice as many unique queries for step one increasing the threshold for the attacker to:

$$2q_1 + q_2 \geq 321.$$

Plugging in our proposed witness size parameters of $||\omega|| = 33$ and $||t|| = 70$ we get $q_1 = 109.3764$ and $q_2 = 102.2471$ which requires

$$2^{109.3764} + 2^{102.2471+6.12} = 2^{109.3764} + 2^{108.3671} \approx 2^{109.9}$$

queries and satisfies the conditions $q_2 = 102.2471 \leq 103$ and

$$2 \cdot 109.3764 + 102.2471 \approx 321.$$

The time cost of creating a transaction that honestly spends the covenant is $2^{86.11}$. The time cost of constructing a transaction that cheats the covenant is $2^{109.9}$. The means that for our proposed parameters an attacker must do $2^{23.79} = 14,504,500$ times more work than an honest spender. Figure 8 (Right) shows how the cost in queries to honestly spend or attack our covenant changes for different sizes of equivalence witness sizes.

The Bitcoin hash rate is roughly 700 exahashes/second. This is $2^{69.25}$ hashes/second, $2^{85.64}$ hash/day, $2^{91.12}$ hashes/year. Honestly spending a covenant has work roughly in terms of number of hash queries to roughly 33 hours of Bitcoin mining. Cheating such a covenant based on the best attack we have developed costs $450,136$ years of the Bitcoin mining network.

Attacker with bounded space. Now we consider an attacker who does not have infinite space. We assume they use the distinguished points approach to find 2^p collisions. This only impacts step 1, as we assume the attacker can store the 2^p colliding pairs needed in step 2 because given the likely values of $||\omega|| + ||t||$ the value 2^p is will to be less than 2^{64}. We shown above in step 1 we need to make $q = 2^{161-(||\omega||+||t||)/2}$ queries to generate $2^p = 2^{(159+1)-(||\omega||+||t||)}$ collisions. The $+1$ comes from the fact that half the collisions are useless, so we need to double the number of collisions we are finding. Using the distinguished points approach each collision costs an additional 2^z time. This results in a step 1 time cost of:

$$2^{161-(||\omega||+||t||)/2} + 2^{z+160-(||\omega||+||t||)}$$

We need to store $2^{161-(||\omega||+||t||)/2-z}$ distinguished points and each distinguished point costs $20 - (z/8)$ bytes for a total space cost of $(20 - z/8) \times 2^{161-(||\omega||+||t||)/2-z}$ bytes. We limit the attacker to 2^{64} bytes, that is roughly 1 million 16-TB hard drives.

$$2^{64} = (20 - z/8) \times 2^{161-(||\omega||+||t||)/2-z}$$

Plugging in $(||\omega|| + ||t||) = 103$ then:

$$2^{64} = (20 - z/8) \times 2^{161-103/2-z} = (20 - z/8) \times 2^{109.5-z}$$

solving for z we get $z = 49.29$ which gives us a time cost for step 1

$$2^{(159-103)/2+81.5} + 2^{49.29+160-103} = 2^{109.5} + 2^{106.29}$$

These means that attackers with $\geq 2^{64}$ space has roughly the same attack time cost as an attacker with unlimited space. Not this only holds for a attacker than wants to break one covenant. An attacker that has infinite space can reuse the work from step two in additional attacks by constructing a $\mathcal{D}.\mathsf{Gen}$ lookup table.

Our analysis assumes that none of the pairs collide with other pairs as $p < 80$ this is unlikely. The $q = 2^{103}$ queries to $\mathcal{D}.\mathsf{Gen}$ will generate $2^{2\times 103+160-1} = 2^{206-160-1} = 2^{45}$ colliding outputs. These colliding outputs from $\mathcal{D}.\mathsf{Gen}$ are wasted work, but 2^{47} is so small we can ignore it.