Bitcoin Covenants

Malte Möser[1], Ittay Eyal[2], and Emin Gün Sirer[2]

[1] Department of Information Systems, University of Münster, Germany
[2] Initiative for Cryptocurrencies & Contracts (IC3),
Computer Science Department, Cornell University, USA

Abstract. This paper presents an extension to Bitcoin's script language enabling *covenants*, a primitive that allows transactions to restrict how the value they transfer is used in the future. Covenants expand the set of financial instruments expressible in Bitcoin, and enable new powerful and novel use cases. We illustrate two novel security constructs built using covenants.

The first, *vaults*, focuses on improving the security of private cryptographic keys. Historically, maintaining these keys securely and reliably has been a critical vulnerability for Bitcoin users. We show how covenants enable vaults, which disincentivize key theft by preventing an attacker from gaining full access to stolen funds.

The second construct, *poison transactions*, is a generally useful mechanism for penalizing double-spending attacks. Bitcoin-NG, a protocol that has been recently proposed to improve Bitcoin's throughput, latency and overall scalability, requires this feature. We show how covenants enable poison transactions, and detail how Bitcoin-NG can be implemented progressively as an overlay on top of the Bitcoin blockchain.

1 Introduction

Bitcoin is an innovative payment system built to enable a wide variety of financial contracts that are executed in a decentralized manner. Part of its power and expressiveness derives from the way transactions use a flexible script language to specify redemption criteria. The system ensures that subsequent transactions must fulfill the redemption criteria in order to unlock the embedded value. While traditional financial contracts rely on trust and after-the-fact enforcement, Bitcoin's scripting mechanism allows to enforce contracts within the currency system itself.

Yet, the functionality provided by the scripting language is characterized by an inherent trade-off between security, efficiency, and expressiveness. Currently, the expressiveness of the script language is limited, not only by the constricted operations of the language, but also by the information that can be accessed in or checked by a script program.

To extend the capabilities of the system, we propose an extension to Bitcoin's script language that enables *covenants*[3]: transactions that are able to enforce re-

[3] A covenant is a special contract in property law that restricts the use of an object, typically restricting the use of land for certain purposes. We adopt the term from earlier discussions on related ideas [22], which are discussed in Section 6.

strictions on the composition of subsequent transactions (cf. Section 3). Covenants enable multiple novel and powerful use cases. We first illustrate the power of covenants by describing how colored coins, a well-established but ill-supported idea to attach meaning beyond nominal value to bitcoins, would benefit from the ability to prevent such coins from accidentally mixing into general circulation. We then focus on two new use cases.

First, we use covenants to implement secure vaults, which addresses one of the biggest problems of cryptocurrency security: the difficulty of secure key management. Vaults improve end-user security by disincentivizing theft of coins using a mechanism that prevents an attacker from gaining full control over funds despite stealing the private keys used to secure them (cf. Section 4).

Then, we describe how covenant functionality enables new overlays to be placed on top of the Bitcoin blockchain. Making changes to the consensus protocol of a cryptocurrency is a difficult process as it requires agreement by participants and stakeholders. Bitcoin-NG [16] is an alternative blockchain protocol that promises significant improvement in transaction throughput and confirmation delay. However, changing Bitcoin's blockchain protocol would require a change to Bitcoin's consensus protocol, a daunting task.

We use covenants to implement *poison transactions*, which invalidate a deposit using fraud proof. With poison transactions, we detail the implementation of Bitcoin-NG as an overlay on top of Bitcoin. This implementation can be progressively adopted, not requiring a change of the consensus rules (cf. Section 5) beyond the general functionality of covenants.

In summary, this paper makes the following contributions:

1. Covenants, a new script operation that enables novel security constructs,
2. Vaults, a construct that reduces private key theft incentives by prohibiting an attacker from gaining control of funds, and
3. An implementation of *Bitcoin-NG* as an overlay using covenant-enabled poison transactions.

We review related work in Section 6 and conclude in Section 7.

2 Preliminaries

Bitcoin is a distributed, decentralized cryptocurrency [23] that uses a probabilistic consensus protocol to serialize transactions of the currency among its users. We describe the elements of Bitcoin's design relevant to this work, a detailed description of the system can be found in [4, 26].

The novel data structure used in Bitcoin, and many other derived altcoins [14, 21], is the *blockchain*, an append-only log used to track and store currency transactions. To serialize new transactions, *miners* aggregate transactions in a block and append the block to the ledger by solving a proof of work crypto puzzle. This process is financially rewarded by allowing the successful miner to mint new coins in a special *coinbase* transaction.

Rather than having a notion of accounts and transactions among accounts, as in earlier cryptocurrency systems [9, 27], Bitcoin tracks the individual coins, or, more accurately, fractions of coins. Each transaction in Bitcoin describes the movement of coins from one logical location to another. Cryptographic tools allow only designated principals to move coins out of a location.

Transaction structure The logical locations are called *transaction outputs*. Each transaction contains an array of such outputs and specifies the amount of currency it places in each. A location is uniquely defined by the unique *transaction identifier* and the index of the output. The coins placed are moved, or *spent*, from their previous locations, namely transaction outputs of previous transactions. The sources are listed in the transaction in an array of *transaction inputs*. We note that there is no notion of individual coin tracking — there is no meaningful way to connect specific inputs to specific outputs.

The sum of values in the outputs referenced by a transaction's input array is the total input value of the transaction. The total output value, given by the sum of values in the transaction's output array, cannot be larger than the total input value. Any value not accounted for is transacted to an output specified by the miner who generated the block in the block's coinbase transaction.

Each transaction furthermore has a locktime field that determines the minimal time (block number or unix time) after which it can be placed.

Script To make sure that funds can only be spent by designated principals, spending an output requires satisfying a predicate. Such a predicate is included in each output as a program written in a stack-based language called *Script* [23]. Inputs redeeming an output have to provide data to the output's program. A transaction is valid if its inputs yield true for all corresponding outputs.

In the common case, transactions are secured with public-key cryptography. The logical location of a coin is defined by the public key supplied in the output's script program. The owner, and only this owner, can move the coins by proving her control of the matching private key in the input.

Typical output script programs require the ownership of one or more private keys for successful validation by including public keys or hashes thereof. To validate signatures corresponding to the keys listed, the script language contains a `CheckSig` operation that accepts a signature and a public key, and then verifies the validity of the signature computed over the spending transaction. A detailed step-by-step execution of a script program can be found in [26].

The Bitcoin script language is intentionally restricted to a small set of opcodes, prioritizing security and efficiency over expressiveness and feature-completeness. A key limitation is that the scope of Bitcoin's script operations is restricted to the data provided in the output program and the data provided in the input script. This rule will, however, soon have an exception in the form of two new opcodes (which we will use) called `CheckLockTimeVerify` (CLTV) [25] and `CheckSequenceVerify` (CSV) [6]. These allow to make an output unspendable until a certain point in time is reached. This extends the awareness of the script to the current position of the transaction in the blockchain.

Algorithm 1: Specification of `CheckOutputVerify`

```
On CheckOutputVerify($index$, $value$, $pattern$)
    if not exists output at output $index$ then
        return False
    if $value \neq 0$ then                                        (Check value)
        if (value at output $index$) $\neq$ $value$ then
            return False
    if $pattern \neq 0$ then                                      (Check pattern)
        $sanitizedPattern \leftarrow pattern$, replacing pattern-placeholders with $pattern$,
          then replacing key placeholders with 0's
        $map \leftarrow$ 1's of length $sanitizedPattern$, but 0's at key placeholders
        if (script at output $index$ bitwise-and $map$) $\neq$ $sanitizedPattern$ then
            return False
    return True
```

3 Covenants

Our main contribution is an extension of Bitcoin's script language to enable covenants: restrictions on future use of coins. Covenants enable a transaction output to restrict the outputs in its spending transaction. Using a form of reflection, a covenant can be specified recursively. This enables the enforcement of covenants across a potentially unlimited number of subsequent transactions.

In this section, we describe the operation of single-use covenants (Section 3.1) and show how to extend them into the future by applying them recursively (Section 3.2). As a running example, we use distinguished coins (Section 3.3). Inspired by colored coins [11], distinguished coins are tokens that correspond to real-world assets that should not be mixed or merged with others.

3.1 Basic Covenants

Each transaction output consists of an amount and an output script program. We enable covenants by adding a new operation to the scripting language that restricts both of these fields. Specifically, the operation takes an output index, an amount and a pattern. It verifies that the output at the given index exists, that it carries the required amount and that its script matches a given pattern. Algorithm 1 shows the formal specification.

We implement this operation as a patch of Bitcoin Core, the standard Bitcoin client, as an opcode, `CheckOutputVerify`. The opcode expects the index as the first parameter, allowing to place it in the input of the spending transaction. The creator of the spending transaction can therefore determine the output's position.

The script pattern is simply a script program with placeholders for variable parts. We make use of two placeholder opcodes that are already used internally by the client, namely `PubKey` and `PubKeyHash`, to represent arbitrary public keys or hashes of public keys within the pattern.

Both placeholders represent fields of static size. The script interpreter replaces each placeholder with the appropriate number of zero-bits. Separately, it creates a bitmask with the program's size that masks out these placeholder locations. A bitwise comparison of both programs sanitized with the bitmask then yields the verification results.

There may exist scenarios in which it is only necessary to check the value or the script program, but not both. In this case, either of those values can be set to 0. This prevents one from requiring that an output script is actually equal to `False`, which prevents any future spending of the output. It also prevents one from requiring that an output carries 0 value, which is not a useful notion either.

A toy example will lead us to the construction of the distinguished coins covenant. Here, we will let a specific 1 BTC stand in for the ownership of a real-world asset. The transaction output requires that the subsequent output sends exactly 1 BTC to an arbitrary public key. We supply `CheckOutputVerify` with (1) an output index of 0, (2) an amount of 1 BTC (specified as 100,000,000 units of 10^{-8} Bitcoin, called Satoshis) and (3) a pattern that contains a placeholder for a public key (*PubKey*) followed by the `CheckSig` opcode:

```
0 <100000000> <PubKey CheckSig> CheckOutputVerify.
```

This covenant ensures that the bitcoin corresponding to the asset can only be transferred in whole and cannot be mixed with other coins. This particular covenant, however, only holds for one transaction.

3.2 Recursive Covenants

It is critical to be able to apply covenants to an entire chain of transactions that derive from a covenant-bearing transfer. This section describes how this can be accomplished with recursive covenants.

We start by enforcing our example covenant over two subsequent transactions by including the covenant for the second output in the covenant for the first output. Modifying our example, the following script program not only enforces the first output in the next transaction to have a value of 1 BTC, but also puts the same restriction on the first output in the subsequent transaction (we omit the output's index hereinafter as they can be supplied in the input script).

```
<100000000> <<100000000> <PubKey CheckSig> CheckOutputVerify
    PubKey CheckSig> CheckOutputVerify <keyDest> CheckSig
```

To extend the sequence of outputs further, we could again include another `CheckOutputVerify` command in the innermost script pattern. Since we cannot repeat this infinitely, and instead of creating a self-reproducing script (a Quine), we use the interpreter to replace a dedicated keyword with the pattern itself.

We therefore add a new placeholder opcode called `Pattern` that allows to specify the occurrence of the pattern within itself. When evaluating a pattern, the `Pattern` opcode will be replaced by the pattern itself, thereby resolving the recursion one step at a time. The following example demonstrates the basic use of the `Pattern` opcode.

```
<100000000> <<100000000> Pattern CheckOutputVerify PubKey
    CheckSig> CheckOutputVerify <keyDest> CheckSig
```

When evaluating the pattern in this script program, the `Pattern` opcode will be replaced by the full pattern itself, yielding the exact same script as a pattern for comparison with the script program of the spending output.

3.3 Distinguished Coins

Many (most notably [11]) have noted that Bitcoin can be used as a digital asset exchange mechanism by associating a physical asset to a certain coin. For example, one could attach an arbitrary amount of bitcoins to a certain amount of gold, deposited with a trusted party. This coin could then be used to represent ownership of the gold and be easily traded.

However, as we noted above, bitcoin amounts are not separately tracked by the system, as every Bitcoin transaction inherently mixes all of its inputs' values. In contrast, covenants are carried from one input to a distinct set of outputs, thereby making it possible to meaningfully link currency flows. Our running example constructs distinguished coins by enforcing the coin to retain its distinguished status in all subsequent transactions.

A small security issue inherent to the script used so far is that a user can have multiple distinguished coins with the same value. When the output index is explicitly provided in the input, a user could reuse the same index for multiple covenants and thereby invalidate all but one of the distinguished coins. Solving this problem is straightforward as each distinguished coin can include a unique identifier in the script that prevents mapping multiple inputs to the same output, as in the following example. First, we define `patternDistinguishedWithId` as

```
<assetId> Drop <value> Pattern CheckOutputVerify PubKey
    CheckSig ,
```

and the distinguished coin covenant is

```
<assetId> Drop <value> <patternDistinguishedWithId>
    CheckOutputVerify <keyDest> CheckSig .
```

3.4 Overhead

The `CheckOutputVerify` opcode maintains Script's simplicity, and does not introduce excessive overhead. While `CheckOutputVerify` does not enable loops, a naïve implementation (verbatim following the specification of Algorithm 1) could cause the interpreter to form an excessively large final script program with repeated use of the `Pattern` opcode. To mitigate this concern, instead of first replacing all `Pattern` placeholder with the script itself, the interpreter incrementally expands the pattern and compares the prefix, thereby bounding the overhead to the parsing time of the output to be matched.

3.5 Discussion

Variants. There are a few design choices in our implementation of covenants. For instance, the covenant opcode could be made more flexible than a simple match. One option is that the opcode can return `true` or `false` on the stack, indicating the verification result. This would allow for elaborate combinations of validity conditions. Similarly, an opcode can return the output value, allowing arithmetic operations for validation.

Our current implementation allows covenants in different transaction inputs to refer to the same outputs, which requires some diligence of covenant programmers. This behavior can be removed by enforcing a one-to-one mapping of output checks to transaction outputs. Each transaction input can check any number of the outputs and map these outputs in order, according to the transaction input order and the access order with `CheckOutputVerify`. This alternative implementation introduces slightly more complexity, but is also more resilient to human error.

Covenant termination. As part of the covenant programmer's responsibility, we note that in various cases covenants should enable an exit strategy, allowing to repurpose a coin, for example after a set time or with a specific private key.

4 Vault Transactions

Bitcoin funds are, by and large, protected by sets of cryptographic secret keys. Whoever knows those keys can instantly, anonymously, and irrevocably move the funds by spending the transaction outputs in which they are represented.

This makes Bitcoin private key theft an attractive target for thieves. A long list of thefts has been curated by the Bitcoin community [20], illustrating that attackers have been able to steal bitcoins worth millions of USD. Even major Bitcoin companies frequently fall prey to such attacks (e. g., [19]).

Correctly managing private keys is therefore one of the central challenges for client-side security in Bitcoin in order to protect users from both accidental loss and deliberate theft. A recent study by Eskandari et al. evaluated different methods of key management and concluded that each of them is vulnerable to a range of attacks [15].

A common approach for storing bitcoins in a secure manner is to put them into "cold storage", which means that their keys are stored on a device not connected to the Internet. However, in order to retrieve the coins, one must regularly (albeit infrequently) interact with those keys, which makes them vulnerable as well.

We now introduce recoverable vaults, which reduce the incentive for Bitcoin private key theft. Vaults provide two mechanisms to increase security. First, funds stored in a vault can be recovered using a recovery key in case the vault key is compromised. Second, in case the recovery key is compromised as well, the owner can still prevent the attacker from moving the coins; the worst the attacker can do is to prevent the owner from regaining full control over the funds. As this reduces the motivation for key theft, users can be more permissive in their storage of the private keys, reducing the chances of key loss.

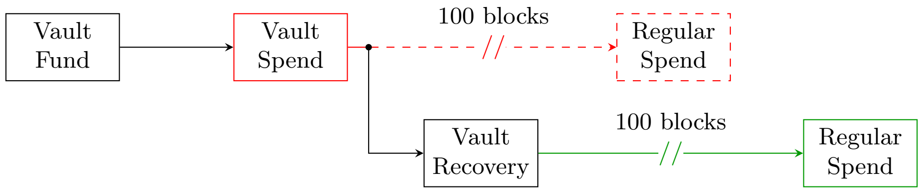

Fig. 1. The attackers spending attempt (red) is interrupted by a vault recovery issued by the legitimate owner, followed by a regular spend of the legitimate owner (green)

Note As explained below, vault transactions use a delay mechanism. We note that vault transactions cannot be implemented with existing timing mechanisms such as the `CheckLockTimeVerify` opcode or transaction locktime.

4.1 Overview

Vault transactions prevent an attacker from instantly moving funds from a victim's wallet by enforcing a delay for the transfer of those bitcoins. Coins placed in a vault transaction cannot be released immediately. The key idea of vaults is that the spending transaction has to be placed publicly on the blockchain, with its output locked for a specified amount of time. During this period, the owner of the coins can abort the release of the coins using a recovery key (preferably placed in cold storage) to send them to a different address in a new spending transaction, thereby denying the payout from the attacker. When the attacker also gains access to the recovery key, she can use the same recovery mechanism to again try to send the funds to her address. However, as the covenant is enforced recursively, the legitimate owner can again cancel the payout. Using a long locktime, it is cheap for the legitimate owner to maintain the block.

While ultimately the attacker can blackmail an owner, promising a share of the funds once they are released, this increases both the cost and the exposure of the attacker due to the need to communicate with the victim and provides a lead for criminal investigations.

4.2 Architecture

To secure an amount with a vault, a user sends it to a *vault fund* transaction. The output of this transaction requires a signature corresponding to the vault key and contains a covenant script program that enforces that the output cannot be spent directly, but must be spent through a *vault spend* transaction. The vault spend offers two possibilities to redeem its funds. First, the funds can be spent to a standard output, but only after a certain time has passed, e. g., 100 blocks using a locktime. This is the timer on the vault. When there is no attack, this simply delays the payout from a vault.

Alternatively, the funds can be spent at any time (without having to wait for the locktime to expire) using the recovery key in another vault spend, identical to the first one (cf. Figure 1). This effectively resets the locktime of the funds.

4.3 Script Programs

In the following we provide the script programs implementing vault transactions. Beside our `CheckOutputVerify`, we assume the availability of another opcode that is currently being developed called `CheckSequenceVerify` (CSV) [6], which allows outputs to specify a locktime relative to the block height (or timestamp) when their containing transaction is committed to the blockchain.[4]

Vault Spend Assume that 1 BTC has been locked in a vault. To spend this bitcoin, the payout transaction has to specify a relative locktime (using CSV) after which the coin can be redeemed with the signature belonging to a certain public key. Before the locktime expires, the funds can be moved to an output that retains the value and adheres to the vault pattern (which we describe below). This new output must be accompanied with a signature corresponding to the recovery key.

```
If
  <100> CheckSequenceVerify <keyDest> CheckSig
Else
  <100000000> <patternVault> CheckOutputVerify <keyRecovery>
      CheckSig
EndIf
```

Pattern To enforce the above script program structure, we use the following pattern. Coins can be spent with an arbitrary public key (specified by the `PubKey` placeholder) after a relative locktime of 100 blocks has passed, or immediately respent in an output that adheres to the same pattern (enforced through the `Pattern` placeholder) and provides a valid signature with the recovery key.

```
If
  <100> CheckSequenceVerify PubKey CheckSig
Else
  <100000000> Pattern CheckOutputVerify <keyRecovery>
      CheckSig
EndIf
```

Vault Fund To initially lock funds in a vault, we use a script program which specifies that an output in the redeeming transaction must adhere to the `patternVault`, have the same value (otherwise it would be possible to retrieve the money through another output) and that the transaction has to provide a valid signature corresponding to the vault key.

```
<100000000> <patternVault> CheckOutputVerify <keyVault>
    CheckSig
```

[4] We abstract from opcode behavior specific to Bitcoin's soft-fork upgrade mechanism, namely the need to drop items from the stack afterwards.

5 Bitcoin-NG Overlay

Bitcoin-NG is a blockchain protocol that offers major improvements of transaction bandwidth and latency in comparison to Bitcoin [16]. We explain how we can use covenants to deploy Bitcoin-NG's core features on top of Bitcoin, so users can benefit from improved efficiency. This allows to gradually deploy Bitcoin-NG, with nodes gradually adopting it.

In the following, we overview the Bitcoin-NG protocol (Section 5.1) demonstrate how such a deployment can be carried out (Section 5.2) and how covenants allow to implement Bitcoin-NG on top of Bitcoin by providing a mechanism to realize *poison transactions* (Section 5.3).

5.1 Preliminaries: Bitcoin-NG Operation

Bitcoin-NG's blockchain has two types of blocks. Key-blocks are generated with proof of work, like in Bitcoin, but contain no transactions. They serve as a leader election mechanism and contain a public key that identifies the chosen leader. Once a leader is elected, she publishes microblocks that contain transactions.

In order to motivate participants to follow the protocol, Bitcoin-NG uses the following mechanisms. As in Bitcoin, proof-of-work is motivated by a subsidy — a prize for mining. As in Bitcoin, each transaction pays a fee to the system, but unlike Bitcoin, this fee is distributed, with 40 % to the leader, and 60 % to the subsequent leader. Finally, if a leader forks the chain by generating two microblocks with the same parent, she is punished by revoking her subsidy revenue; whoever detects the fraud wins a nominal fee.

5.2 Overlaying Bitcoin-NG on Top of Bitcoin

Our goal is to have Bitcoin-NG nodes use the Bitcoin protocol when communicating with other Bitcoin nodes, but use the Bitcoin-NG protocol when talking to Bitcoin-NG nodes.

To achieve this, all the information in Bitcoin-NG blocks — both key-blocks and microblocks — must be translated into standard Bitcoin blocks for compatibility. Specifically, key-blocks are mapped to standard Bitcoin blocks. We start with the case where consecutive key-blocks are found by Bitcoin-NG miners. The second miner puts all transactions placed in microblocks by the previous miner into the mapped Bitcoin block. When Bitcoin-NG nodes communicate with each other, they exchange key-block and microblock data structures. A Bitcoin-NG node can reconstruct the standard Bitcoin blocks on demand. However, if a Bitcoin-NG node communicates with a standard Bitcoin node, it sends the standard blocks, which contain all the transactions. For both Bitcoin and Bitcoin-NG nodes, mining is performed on the standard Bitcoin blocks, again, for backward compatibility.

Recall that in Bitcoin-NG the transaction fees are distributed among the current and next leaders. In the overlay implementation, the microblock transactions are actually placed in the subsequent key-block, and their fees go to the

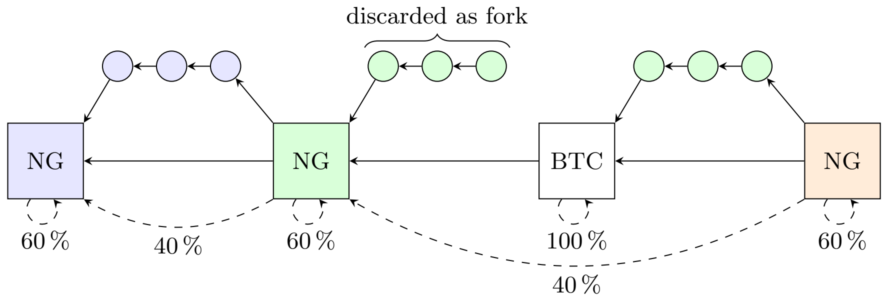

Fig. 2. Structure of a mixed Bitcoin and Bitcoin-NG blockchain

subsequent leader. This key-block must therefore redistribute those fees. If the fees are not distributed correctly, the block is not a valid Bitcoin-NG key-block, and it is considered a standard block by the protocol.

However, if not all miners are running the Bitcoin-NG client, some blocks are found by non-NG miners. These do not respect the microblock chain of the current leader, and do not distribute the fees correctly. If a Bitcoin-NG key-block (and its microblocks) is followed by a standard block, Bitcoin-NG-miners discard the current microblock chain; the leader, previously chosen, remains leader and starts a new microblock chain on top of the standard block. This is illustrated in Figure 2. A new leader will thus pay 40 % of the fees in the latest microblock chain to the previous leader, no matter how many standard block separate its key-block from the previous leader's key-block. Keeping the leader across standard blocks has two objectives. First, the Bitcoin-NG fast transaction commitment can be used even after standard blocks. Second, it increases the incentive to run a Bitcoin-NG node, as the leader is guaranteed to win 40 % of a subsequent epoch, even if not the immediate next one.

5.3 Poison Transactions

The missing piece towards deploying Bitcoin-NG on top of Bitcoin are poison transactions. Leaders commit to destroying a large share of their own coinbase reward if they produce a fork in their microblock chain. Destroying the value of the coinbase is enforced by a covenant. Without this commitment, the blocks are not accepted as valid Bitcoin-NG blocks.

Poison Structure Bitcoin-NG's coinbase transactions need a time frame in which they are unspendable by the miner, but destroyable by a poison transaction. In Bitcoin, a consensus rule enforces that normal coinbase outputs can be spent after 100 confirmations [3]. Bitcoin-NG's coinbase transactions must therefore delay the ability to spend the coinbase output by an additional number of blocks t. We implement this using CLTV:

```
If
    <height+100+t> CheckLockTimeVerify <pkLeader> CheckSig
Else
    <90% of value> <Return> CheckOutputVerify
    <10% of value> 0 CheckOutputVerify
    <pkPoison> CheckSig
EndIf
```

Within these t blocks a poison transaction can destroy a significant share of the coinbase's value and reward the reporting user with the remainder of the funds. This mechanism is enforced by a covenant that ensures that most of the value is destroyed in an unspendable `Return` output and the rest of the funds can be claimed by the user reporting the misbehavior, who can choose her own output script program.

Fraud Detection The crux of the fraud detection mechanism is to construct every microblock such that if the leader creates a fork with more than one microblock succeeding any block it is possible to extract the private poison key.

To achieve this, we use a property of the ECDSA signature scheme used for Bitcoin transaction signing. Each ECDSA signature created with a secret key d requires the signer to select an ephemeral key k, that is, a secret random number used in the signing process. This ephemeral key must not be reused with the same private key to sign another message as this allows to calculate d from the two signatures [18]. In fact, such operational security mistakes have led to theft of bitcoins [5].

Every ECDSA signature contains a value r that is computed based on k and otherwise fixed parameters. Computing k from r is believed to computationally infeasible [18]. We utilize this fact as follows.

Each key-block and microblock are published with a bundled value r, thereby committing to a certain ephemeral key for the *next* microblock. Each microblock is signed with the leader's poison key using the ephemeral key previously selected; if the r value of a microblock does not match the commitment in the previous block, it is considered invalid.

If the leader creates a microblock fork, she is forced to reuse the ephemeral key to sign the microblocks. This allows any party with access to both messages to calculate the private poison key. The leader can only profit from such forking by making one microblock public, as part of the main chain, and one microblock known to some defrauded party. Once this defrauded party learns about the fork, she can find the poison private key and expose the fraud.

Note that we use two different keys in the scheme as the leaked key should only enable the poison mechanism, but not to be useful to produce arbitrary microblocks on behalf of the leader.

6 Related Work

Covenants The first mention of *covenants* in Bitcoin is due to Maxwell [22], who coined the term. Maxwell proposed using zero-knowledge succinct noninteractive arguments of knowledge (SNARKs) to place, and discharge, arbitrarily complicated constraints on any data in the blockchain. However, even assuming SNARKs can be implemented efficiently, the generality of this approach makes it difficult to reason about the system's security. Consequently, this idea was immediately dismissed by Maxwell himself. Ethereum [7] is a blockchain-based protocol that provides a Turing-complete programming language for writing arbitrary programs. The power of the scripts is limited in Ethereum through utility pricing to limit malicious use and to maintain fairness. While the programming language is universally expressive and can thus implement covenants, it offers no formal security guarantees. In contrast, the covenants implementation we propose requires only limited changes to Bitcoin's limited script language, accesses only designated outputs, and incurs nominal overhead.

Further discussion on the concept of covenants [2] focuses mostly on risks, such as potential impact on fungibility and the possibility to use covenants to enforce anti-money laundering (AML) regulation upon Bitcoin. Covenants do not necessarily impact fungibility if programmed properly, and it is the responsibility of the covenant programmer to lift a covenant when it makes sense to do so. In general, most general-purpose extensions, including the presence of unconfirmed transactions [13] as well as extensions such as CLTV, can pose problems for fungibility if not properly used. Overall, the political consequences of general-purpose technical features are beyond the scope of this paper.

Vault transactions In [10], a Bitcoin forum participant outlines a 4-line proposal to deter theft using restrictions on expenditures. This scheme uses a recursive covenant that allows the owner of funds to abort a theft transaction within a bounded time, and send the funds to a new output, secured by a different private key. While this scheme may be useful in certain scenarios, in essence, it simply secures the funds by an additional key. Multisignature transactions, which require m out of n keys to be used to sign a valid transaction, provide similar protections. Eskandari et al. [15] evaluate the usability of different key management schemes, and conclude that there is no silver bullet for private key storage. And while others have suggested more efficient threshold signature schemes based on ECDSA [17], with better privacy and smaller transaction size in comparison to standard multisignature transactions, these schemes all rely on the secrecy of the signing keys. In contrast, vaults prohibit a thief from taking possession of the funds even if she learns *all* the secret keys.

Fraud proof Fraud-proofs have gained attention with the introduction of cryptocurrencies due to the ability to use them against a security deposit. Fraud proofs similar to the one we use were suggested in the context of generic covenants by d'aniel and Todd [12]. Other instances are in the context of pegged sidechains [1], and proof-of-stake [8]. Ruffing, Kate, and Schröder [24] propose a general scheme for double-attestation proof, using a cryptocurrency as a primitive.

7 Conclusions

We showed how Bitcoin covenants can be added to the existing scripting language with a single simple opcode with nominal overhead. Overall, covenants introduce a novel functionality that opens the door to a wide range of security constructs and financial contracts. We demonstrate this with two novel and useful constructs.

The first, vault transactions, tackles cryptocurrency key security. Vault transactions significantly reduce theft in Bitcoin by removing the ability of a thief to keep the proceeds.

The second, poison transactions, enable automatic fraud-proof-based penalizing, a generally useful construct. We showed how covenant-enabled fraud-proofs can be used to progressively deploy Bitcoin-NG as an overlay on top of the Bitcoin blockchain, thereby enabling significant improvements in throughput, confirmation time and scalability.

Acknowledgments The authors thank Glenn Willen for useful conversations, Tim Ruffing and Dominique Schröder for their advice on cryptographic primitives, and the anonymous reviewers for their valuable feedback.

This material is based upon work supported by a fellowship within the FITweltweit programme of the German Academic Exchange Service (DAAD), the German Bundesministerium für Bildung und Forschung (BMBF) under grant agreement No. 13N13505, and the National Science Foundation under Grant No. CNS-1518779 and Grant No. CNS-1561209. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

Bibliography

[1] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. *Enabling Blockchain Innovations with Pegged Sidechains.* URL: `https://blockstream.com/sidechains.pdf` (visited on 2015-11-03).

[2] *#Bitcoin-Wizard IRC log.* URL: `https://download.wpsoftware.net/bitcoin/wizards/2014/01/14-01-15.log` (visited on 2015-10-28).

[3] *Block chain.* URL: `https://en.bitcoin.it/w/index.php?title=Block_chain&oldid=59033` (visited on 2015-10-19).

[4] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. "Research Perspectives on Bitcoin and Second-Generation Cryptocurrencies". In: *IEEE Symposium on Security and Privacy.* San Jose: IEEE, 2015.

[5] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. "Elliptic Curve Cryptography in Practice". In: *Financial Cryptography and Data Security.* Vol. 8437. Barbados: Springer Berlin Heidelberg, 2014, pp. 157–175.

[6] BtcDrak, Mark Friedenbach, and Eric Lombrozo. *BIP 112: CHECKSEQUENCEVERIFY.* 2015. URL: `https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki` (visited on 2015-10-08).

[7] Vitalik Buterin. *A Next Generation Smart Contract & Decentralized Application Platform.* `https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf/`, retrieved Feb. 2015. 2013.

[8] Vitalik Buterin. *Slasher: A Punitive Proof-of-Stake Algorithm.* `https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/`. January 2015.

[9] David Chaum, Amos Fiat, and Moni Naor. “Untraceable Electronic Cash”. In: *Advances in Cryptology — CRYPTO’ 88.* Ed. by Shafi Goldwasser. Vol. 403. Lecture Notes in Computer Science. New York: Springer, 1990, pp. 319–327.

[10] coastermonger. *Thief’s downfall covenant.* 2013. URL: `https://bitcointalk.org/index.php?topic=278122.msg3164726#msg3164726` (visited on 2013-09-16).

[11] Colored Coins Project. *Colored Coins.* `http://coloredcoins.org/`, retrieved Sep. 2015.

[12] d’aniel and Peter Todd. *Security deposits.* 2013. URL: `https://bitcointalk.org/index.php?topic=278122.msg2973895#msg2973895` (visited on 2013-08-20).

[13] Christian Decker. *[bitcoin-dev] [BIP] Normalized transaction IDs.* 2015. URL: `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-November/011657.html` (visited on 2015-11-03).

[14] Dogecoin Project. *Dogecoin.* `https://dogecoin.org`, retrieved Nov. 2014.

[15] Shayan Eskandari, David Barrera, Elizabeth Stobert, and Jeremy Clark. “A First Look at the Usability of Bitcoin Key Management”. In: *NDSS Workshop on Usable Security (USEC).* 2015.

[16] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert van Renesse. “Bitcoin-NG: A Scalable Blockchain Protocol”. In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, March 16-18, 2016, Santa Clara, CA, USA.* 2016.

[17] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A. Kroll, Edward W. Felten, and Arvind Narayanan. “Securing Bitcoin Wallets Via a New DSA/ECDSA Threshold Signature Scheme”. 2015.

[18] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer New York, 2004.

[19] Stan Higgins. *Bitstamp Claims $5 Million Lost in Hot Wallet Hack.* 2015. URL: `http://www.coindesk.com/bitstamp-claims-roughly-19000-btc-lost-hot-wallet-hack/` (visited on 2015-10-16).

[20] *List of Major Bitcoin Heists, Thefts, Hacks, Scams, and Losses.* URL: `https://bitcointalk.org/index.php?topic=576337` (visited on 2015-10-16).

[21] Litecoin Project. *Litecoin, open source P2P digital currency.* `https://litecoin.org`, retrieved Nov. 2014.

[22] Gregory Maxwell. *CoinCovenants Using SCIP Signatures, an Amusingly Bad Idea.* 2013. URL: https://bitcointalk.org/index.php?topic=278122.0 (visited on 2015-10-25).

[23] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* http://www.bitcoin.org/bitcoin.pdf. 2008.

[24] Tim Ruffing, Aniket Kate, and Dominique Schröder. "Liar, Liar, Coins on Fire! — Penalizing Equivocation By Loss of Bitcoins". In: *CCS'15.* Proceedings of the 22nd Conference on Computer and Communications Security. (Denver, CO, USA). CCS'15. New York, NY, USA: ACM.

[25] Peter Todd. *BIP 65: OP_CHECKLOCKTIMEVERIFY.* 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki (visited on 2015-10-08).

[26] Florian Tschorsch and Björn Scheuermann. *Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies.* Cryptology ePrint Archive, Report 2015/464. 2015.

[27] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gün Sirer. "Karma: A Secure Economic Framework for Peer-to-Peer Resource Sharing". In: *Workshop on the Economics of Peer-to-Peer Systems.* Vol. 35. Berkeley, California, 2003.