Enhancing Bitcoin Transactions with Covenants

Russell O'Connor and Marta Piekarska

Blockstream
{roconnor, marta}@blockstream.com

Abstract. Covenants are Bitcoin Script programs that restrict how funds are allowed to be spent. In previous work [9], Möser et. al. implemented covenants with a new Script operation that allows one to programmatically query the transaction. In this paper, we show that covenants can be implemented with a new CHECKSIGFROMSTACK operation that verifies a signature for a message passed as an argument. When the same public key and signature is used together with CHECKSIG, one can recover transaction data, which then allows one to enforce a covenant. To illustrate our technique, we reimplement Möser et. al.'s vault construction for securing funds against key compromise. We use Elements Alpha, a sidechain whose Script language has the needed operations.

1 Introduction

To spend funds in Bitcoin, one has to provide an input to satisfy a predicate that is associated with the funds. This predicate is programmed in a language called Script [11]. A typical predicate requires a digital signature for a public key that is fixed by the particular program. However, more complex predicates are possible.

Predicates restrict who is authorized to make a transaction. Recent extensions to the Script language, such as CHECKSEQUENCEVERIFY [3], allow predicates to restrict when a transaction is authorized. However, there is no way to restrict what transactions are authorized. Once someone has the authorization to spend funds, they may send the funds anywhere they wish.

A way to limit how funds may be spent, including specifying how much must be spent and to what addresses, is by the introduction of covenants. Covenants may be recursive by requiring transactions to be spent to outputs that contain the same covenant. State can be stored and updated in these scripts allowing one to build smart contracts that execute a state machine through a series of transactions.

It is believed that it is impossible to introduce covenants in Bitcoin as Script does not contain operations that allow reading of the transaction data. The only way to interact with the transaction data is by use of CHECKSIG that verifies a digital signature for a message built from the transaction data. Thus, some have proposed to extend Script to support covenants by adding new operations to interact directly with the transaction data [9].

This paper introduces a novel approach to the problem. We show that it is possible to implement covenants in Bitcoin by adding purely computational operations that do not access the transaction data. Instead, we leverage the existing CHECKSIG operation to recover the signed message data that is built from the transaction data. Then it is easy to add conditions that restrict what transaction data is acceptable for one's particular covenant.

To illustrate how this works, we implement Möser et. al.'s covenant for vault transactions [9] in Elements Alpha, a Bitcoin sidechain that includes our needed extensions to Script.

The rest of the paper is organized as follows. In the next section, we introduce the basics required to understand how covenants work. Next, in Section 3, we talk about the Elements Alpha sidechain that we used for implementation of our solution. Section 4 describes the covenants and how they work in Elements Alpha. In Section 5, a classical use case is presented: Möser et. al.'s vault covenant. Related work is discussed in Section 6, and we conclude in Section 7.

2 Background

In this section we discuss how Bitcoin Transactions work and what Script is.

2.1 Bitcoin Transactions

A Bitcoin transaction contains a series of inputs and outputs. The data for each output contains its Bitcoin value and a predicate written in Script called the *scriptPubKey*. The data for each input contains an *outpoint*, which references a previous transaction's output, and a *scriptSig*, which is the input for that output's predicate. The sum of the output values must be equal to no more than the sum of the values of the outpoints referenced by the inputs. Any difference between the two sums counts as fee that Bitcoin miners may collect for adding the transaction to the blockchain.

2.2 Script

Script is a Forth-like, stack based language for defining predicates. Its operations manipulate a stack of byte arrays. Input for a Script program, the scriptSig, gives the initial state of the stack. Execution is successful when all of the program's operations complete and the resulting stack has a non-zero value on top.

Script is a deliberately limited language; it has conditionals but no looping (or recursion) operations. This means that the language is not Turing complete. These limitations facilitate static analysis of its programs. For example, only a limited number of expensive, digital signature verification operations are allowed per transaction, and this limit is statically checked by counting the number of the operations appearing in the program.

The CHECKSIG and CHECKSIGVERIFY operations perform a digital signature verification of a *signed hash*. A signed hash is a double SHA-256 hash of *signed data* generated from the transaction data. The signed data is determined by a *SigHash type* which is specified by a byte that is appended to the digital signature. For the most common SigHash type, `SIGHASH_ALL`, the signed data consists of the transaction data with the scriptSigs replaced by the byte `0x00`. The exception being the input corresponding program being executed. There the scriptSig is replaced with the scriptPubKey of its outpoint, which is the scriptPubKey being executed. Later we use this exception to implement recursive covenants.

Other SigHash types produce variants of this signed data. We will be using one that allows us to discard all but the first input and output of the transaction data.

3 Elements Sidechain

Elements Alpha [1] is a fork of the Bitcoin codebase that implements a sidechain on Bitcoin's testnet [2]. Instead of mining, a federation of signers produces one block per minute. Coins are not minted by new blocks; instead they enter the sidechain through a pegging process. The user sends testnet coins to a multisignature scriptPubKey controlled by the federation. Once confirmed, the same value of coins are unlocked on the Elements Alpha sidechain. Elements Alpha's coins can be redeemed on Bitcoin's testnet later by locking the coins on the sidechain. Once locked, the federation will release the same value of coins on Bitcoin's testnet.

Elements Alpha lets us explore new features for Bitcoin without putting the main network at risk. For example, Segregated Witness [6,7], Confidential Transactions [8] as well as the new Script operations were all developed in Elements Alpha. In particular, the inputs in Element Alpha's signed data include their Bitcoin value. We will now show how these new Script operations let us implement covenants.

4 Covenants in Elements Alpha

New operations in Elements Alpha's Script include: CHECKSIGFROMSTACK, CHECKSIGFROMSTACKVERIFY and CAT. Interestingly, the CAT operation used to exist in Bitcoin Script, but it was disabled [10].

The two CHECKSIGFROMSTACK operations are similar to the CHECKSIG operations except they perform a digital signature verification of the SHA-256 hash of a messaged passed on the stack. These operations have several applications including secure multi-party computation [5]. For the purposes of covenants, CHECKSIGFROMSTACK is used in conjunction with CHECKSIG to recover the signed data.

After a successful CHECKSIG operation, the digital signature and public key together form a commitment to the signed data. If the same public key and signature are used in a successful CHECKSIGFROMSTACK operation, it provides a cryptographic guarantee that the message passed to the CHECKSIGFROMSTACK operation is identical to the signed data.

4.1 Recovering Signed Data

In order to present a clear example of how our solution is realized, let us take the following stack:

signature
pubkey
message

In the Script program presented in Listing 1.1, the first line duplicates the signature and public key, and then it appends the byte `0x01` to the end of the signature, which is the flag for `SIGHASH_ALL` type. In the next line, a CHECKSIGVERIFY is executed using the signature and the public key. If it is successful, it means that the signature and public key form a commitment to the signed data for the `SIGHASH_ALL` SigHash type. This leaves the original three items on the stack. Next, the program computes the SHA-256 hash of message data. Finally, a successful CHECKSIGFROMSTACKVERIFY ensures that the message is identical to the signed data from the CHECKSIGVERIFY operation on the second line. This leaves only the message on the stack which has been proven identical to the signed data.

```
2DUP 1 CAT
SWAP CHECKSIGVERIFY
2 PICK SHA256
ROT CHECKSIGFROMSTACKVERIFY
```

Listing 1.1. Elements Alpha Script to verify that *message* is the signed data

Further operations can be added to enforce that the transaction data contained in the signed data satisfies whatever policy the user desires to enforce.

The signed data recovery process only relies on the *integrity* property of digital signatures. In essence, we are treating the *signature*-*pubkey* pair as a cryptographic hash of message data. The three inputs, *signature*, *pubkey*, and *message*, can be provided in the scriptSig by the person creating the transaction. In the next section, we will show how to apply this technique to build a practical covenant.

5 The Möser-Eyal-Sirer Vault

In this section, we recreate the Möser-Eyal-Sirer Vault in Elements Alpha. Möser, Eyal, and Sirer described an implementation of covenants using a new operation, CHECKOUTPUTVERIFY, that directly verifies if a transaction output matches a given pattern [9]. Using this operation, they developed covenants to implement a smart contract for a vault to help secure funds against malicious transfers.

In their scheme, funds held in the vault can only be withdrawn through a two transaction process. The first transaction's output has a time-lock. This introduces a fixed delay, called the *unvaulting period*, before the second transaction can send the funds to the destination. The purpose of the time-lock is to provide an opportunity for the fund's owner to detect transfers made by a malicious party who may have obtained the vault's private keys. During the unvaulting period, the user has an option to use a rescue private key, kept offline, to create a transaction that overrides the destination address of the withdrawal. This override starts another unvaulting period, during which further overrides can be made. This design ensures that even if the malicious party gets the rescue private key, they still won't be able to profit because the owner and the malicious party end up locked in an endless battle of repeatedly resetting the target script. Given the no-win scenario, hopefully the malicious party realizes that there is no point in attacking in the first place.

Our implementation of the vault smart contract is composed of two Script programs [12]. The first program is the *main vault script*. It holds funds in the vault and its covenant forces that the funds are sent to a scriptPubKey containing a vault loop script, which is the second half of the smart contract. The *vault loop script* uses CHECKSEQUENCEVERIFY to enforce the unvaulting period after which its covenant forces that the funds are sent to a destination that was set by the main vault script's input. Alternatively, the vault loop script allows the funds to be sent, at any time, to another instance of vault loop script containing a new destination address when authorized by a rescue key. In this sense, the vault loop script is recursive.

5.1 Main Vault Script

input 1	*outpoint*
	value
script	*main-vault-script*
output 1	*value*
scriptPubKey	PUSH *target*
	vault-loop-script
SigHash type	SIGHASH_SINGLE

Table 1. Summary of recovered signed data for the main vault script. Items in italics are data provided by the scriptSig input.

The main vault script reconstructs the signed data from pieces provided by the scriptSig and from fixed constants. Table 1 summarizes the reconstructed signed data. The items in italics are provided by the scriptSig while the other items are fixed by the main vault script. The CHECKSIG / CHECKSIGFROMSTACK technique described in Section 4 verifies that the reconstructed signed data matches the actual signed data. For the public key, we require either the wallet or rescue key to be used. This way, the CHECKSIG is used for both covenant enforcement and verifying the transaction is authorized.

The same *value* parameter is used in both the input and output of the signed data to ensure the entire vault's funds are moved together. We use the SIGHASH_SINGLE type to generate signed data that excludes all but the first input and first output. This allows other inputs to cover the transaction fees.

The output's scriptPubKey begins by pushing a *target* value, and it is followed by the vault loop script. The *target* value is the initial "state" for the vault loop script. It determines the scriptPubKey of the fund's destination. The next section will describe how vault loop script works.

5.2 Vault Loop Script

input 1	*outpoint*
	value
script	PUSH target
	vault-loop-script
output 1	*value*
scriptPubKey	target
SigHash type	SIGHASH_SINGLE

Table 2. Summary of signed data for the standard redemption of funds for the vault loop script.

input 1	*outpoint*
	value
script	PUSH target
	vault-loop-script
output 1	*value*
scriptPubKey	PUSH *new-target*
	vault-loop-script
SigHash type	SIGHASH_SINGLE

Table 3. Summary of signed data for rescue of funds for the vault loop script.

There are two different ways to redeem the vault loop script. The primary method is to wait out the unvaulting period and then send the funds to the target destination. The secondary method is to use the rescue key to send the funds to another copy of the vault loop script with a new target script.

Table 2 summarizes the standard redemption's signed data. We allow any public key to be used for the covenant enforcement because redemption doesn't require authorization. Instead, we rely on the covenant to restrict the transaction's output to the target script that is fixed by the script's "state", and we use a time-lock to enforce the unvaulting period.

At any time, the owner may change the destination of the vault loop by redeeming it with a rescue transaction that replaces the "state" with a *new-target*. Table 3 summarizes the signed data for the rescue transaction. In this case, we require that the rescue public key is use to enforce covenant as this also verifies that the transaction is authorized.

Because the signed data includes the script being executed, we can enforce the input and output scripts are the same. It is an example of building a recursive smart contract composed of Scripts, even though the Script language itself does not allow loops or recursion.

6 Related Work

In this section we compare our solution for covenants with Möser et. al.'s solution [9]. Their solution proposes adding a CHECKOUTPUTVERIFY operation to Script. Given an output index, a value, and a script pattern, CHECKOUTPUTVERIFY verifies that the transaction's output at the given index has the given value and its scriptPubKey matches the given pattern. Their script pattern relies on a few ad hoc placeholders including PUBKEY, PUBKEYHASH, and PATTERN. The PUBKEY and PUBKEYHASH placeholders provide places where "state" variables can be changed. The PATTERN placeholder is replaced with an instance of the script itself, allowing one to construct recursive covenants without resorting to building Quines.

Our solution does not require patterns. Using CAT, we can assemble arbitrary scripts from some parts taken from inputs and other parts that are fixed. Instead of having a PATTERN placeholder or using Quines, we take advantage of the fact that the input script is part of the signed data to build recursive covenants. We can copy only part of the input script to the output script, leaving the rest of the script to store the updateable "state" of a smart contract.

That said, our solution comes at significant cost. The CHECKSIGFROMSTACK operation is as expensive as CHECKSIG, which is by far the most expensive operation in the Script language. Also, CHECKOUTPUTVERIFY is designed to be easily soft-forked in, while our solution depends on CAT, which would require a new Segregated Witness Script version to enable it.

The next section will discuss how the implications of this work is more about the inevitability of covenants rather than about our solution being practical.

7 Conclusion

In this paper we presented a way to implement covenants, which can limit how funds may be spent, including specifying how much must be spent and to what addresses. To present a specific use case, we implemented the Möser-Eyal-Sirer vault in Elements Alpha. It would be possible to adapt it to create vaults and other covenants for similar blockchains. In particular, if CAT and CHECKSIGFROMSTACKVERIFY were added to Bitcoin's Script language then the implementation presented here could be introduced in Bitcoin.

It is important to observe that CAT and CHECKSIGFROMSTACK are pure functions in the sense that they are functions whose outputs are computable solely from their stack arguments. This paper demonstrates we can recover the signed data without the needing operations that access the signed data beyond the existing CHECKSIG operation. The fact is that the main thing stopping signed data recovery in Bitcoin's Script today is that it is *infeasible* to implement CHECKSIGFROMSTACK with the existing operations, rather that it being inexpressible. Any new operations that would make it feasible to implement

CHECKSIGFROMSTACK would enable covenants. For example, adding primitive elliptic curve and finite field operations for the Secp256-k1 curve [4] would likely be sufficient for implementing CHECKSIGFROMSTACK.

We see that Bitcoin's CHECKSIG operation fails to abstract away the signed data, even if abstraction was the intention. Rather than forcing users to go through an expensive CHECKSIGFROMSTACK to gain access to the transaction data embedded in the signed data, it would be better and cheaper for everyone involved to provide operations to directly access the transaction data.

References

1. Back, A.: Announcing sidechain elements: Open source code and developer sidechains for advancing bitcoin (2015), Blockstream blog post, `https://blockstream.com/2015/06/08/714/`
2. Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., Wuille, P.: Enabling blockchain innovations with pegged sidechains (2014), `https://www.blockstream.com/sidechains.pdf`
3. BtcDrak, Friedenbach, M., Lombrozo, E.: BIP112: Checksequenceverify. Bitcoin Improvement Proposal (2015), `https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki`
4. Certicom Research: Standards for Efficient Cryptography 2: Recommended Elliptic Curve Domain Parameters. Standard SEC2, Certicom Corp., Mississauga, ON, USA (Sep 2000)
5. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 418–429. CCS '16, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2976749.2978424`
6. Lombrozo, E., Lau, J., Wuille, P.: BIP141: Segregated witness (consensus layer). Bitcoin Improvement Proposal (2015), `https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki`
7. Lombrozo, E., Wuille, P.: BIP144: Segregated witness (peer services). Bitcoin Improvement Proposal (2016), `https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki`
8. Maxwell, G.: Confidential transactions (2015), plain text, `https://people.xiph.org/~greg/confidential_values.txt`
9. Möser, M., Eyal, I., Gün Sirer, E.: Bitcoin covenants. In: Clark, J., Meiklejohn, S., Ryan, P.Y., Wallach, D., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security: FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers, pp. 126–141. Springer Berlin Heidelberg, Berlin, Heidelberg (2016), `http://dx.doi.org/10.1007/978-3-662-53357-4_9`
10. Nakamoto, S.: misc changes. `https://github.com/bitcoin/bitcoin/commit/4bd188c4383d6e614e18f79dc337fbabe8464c82` (Aug 2010), `https://bitcoin.svn.sourceforge.net/svnroot/bitcoin/trunk@131`
11. Nakamoto, S.: Re: Transactions and Scripts: DUP HASH160 ... EQUALVERIFY CHECKSIG. `https://bitcointalk.org/index.php?topic=195.msg1611#msg1611` (Jun 2010)
12. O'Connor, R.: Covenants in Elements Alpha (Nov 2016), Blockstream blog post, `https://blockstream.com/2016/11/02/covenants-in-elements-alpha.html`