FE'd Up Covenants

Jeremy Rubin*

May 29, 2024

Abstract

Covenants are a way of expressing restrictions on Bitcoin. Covenants, while possible to implement as an extension to Bitcoin, do not exist natively. To enable them requires the Bitcoin community to agree upon upgrades such as CTV, CAT, CSFS, and more. This paper serves to demonstrate at a high level how covenants could be introduced to Bitcoin without a soft fork using Functional Encryption and Zero Knowledge Proofs.

1. Introduction

1.1. Preliminaries

1.1.1. Functional Encryption

Functional Encryption (FE) is well explained in the literature[CI].

At a high level, FE is an algorithm for a given function F:

(I) $KeyGen() \to (k, \mathcal{K})$
(II) $EncryptFunc(k, F : A \to B) \to q$
(III) $Encrypt(\mathcal{K}, A) \to c$
(IV) $Decrypt(q, c) \to B$

This allows computing over encrypted data to a decrypted result.

We will particularly focus on two equivalent varieties, function-hiding and multi-argument (which are theoretically equivalent). In multi-argument, you can submit multiple independently encrypted arguments to F. In function hiding, the function F cannot be learned from q.

With fully general function-hiding, you can implement multi-argument by currying. E.g., let $F : A \to G, G : B \to H, H : C \to D$. These functions work all in plaintext to compose $F \circ G \circ H : A \to D$. If we wrap each output in $EncryptFunc$ with k, then we are able to get a curried function that accepts multiple arguments.

*jeremy [dot] L [dot] rubin [at] gmail [dot] com

Alternatively, if you have multi-argument FE (MAFE), you can have a parameter W that is encrypted and accepted as the first argument, e.g. $F : (W, A) \rightarrow B$. The function can check that the W parameter is correct via hash comparison $sha256(W) = h$. W serves as the functions hidden state. If the function is a generic CPU, and the W hidden state is a program, one can see how multi-argument enables full function hiding.

The approach of multi-argument reduction to function-hiding seems simpler, so if multi-argument is available it may not be worth developing function hiding separately.

1.1.2. ZKP/Stark

It is out of scope of this paper to deliver a full treatment to ZKP. There exists ample literature, but the short version is being able to prove a predicate P via a certificate C where the certificate is compact, efficient to verify, and/or not revealing information. For example, one could prove they know all branches of a Merkle Tree without revealing the size or contents of the Merkle Tree.

1.1.3. Covenants

Covenants are restrictions on where and how Bitcoin can move. The simplest covenant is something like CTV, which is a commitment to an exact next transaction. That is, it is a predicate $P(tx) : sha256(tx) = v$. More sophisticated covenants can be built out of things like OP_CAT, or other generalized computation predicates.

2. FE'd Up Covenants

Covenants can be built from MAFE. We will demonstrate with CTV.

First, generate a new functional encryption key and a EC key[1].

Let a trusted party generate the following parameters:

$$(m, \mathcal{M}) \longleftarrow KeyGen()$$

$$(p, \mathcal{P}) \longleftarrow Secp256k1KeyGen()$$

Next, encrypt m.

$$C_p \longleftarrow Encrypt(\mathcal{M}, p)$$

Next, let

$$UniqueSecKey(p, TX) = Add(m, CTVHash(TX))$$

$$UniquePubKey(\mathcal{P}, TX) = TweakAdd(P, CTVHash(TX))$$

Next, let $F(p, TX) = ECSDASign(UniqueSecKey(p, TX), TX)$

Now let $u \longleftarrow UniquePubKey(\mathcal{P}, TX)$.

Next,

[1]We'll use ECDSA for simplicity here, to avoid Taproot. But both should work

$$C_F = EncryptFunc(m, F)$$

Now the trusted party deletes m, p.

We can now do:

$$\sigma \longleftarrow C_F(C_P, Encrypt(\mathcal{M}, TX))$$

σ can be used as a signature of u over TX.

If m, p have been deleted, C_F can only be used to sign in accordance with CTV.

2.1. Program Keys

In CTV, the program is the key. In general, we want

$$UniquePubKey(\mathcal{P}, E_I) = TweakAdd(\mathcal{P}, Sha256(E_I))$$

where E is the generic covenant script and E_I is the concrete parametrized instance that is executing that is e.g. equivalent to the Extended Bitcoin Script you'd use to create the covenant.

2.2. Extending to Extra Data

Intuitively we can replace TX in the example with a TX and some additional argument data. The TX needs to go on chain, the extra arguments are only for the blackbox function. $F(p, TX, V)$ E.g., in CTV, the parameter V could tell the compiler which input index the covenant should be generated for. This is only ever dual-argument MAFE as the (TX, V) are one logical parameter and can be encrypted at the same time and de-structured in F.

2.3. Public Parameters

An unknown area of inquiry (as far as I can turn up in the literature) is FE with partial private parameters. In the covenant application, the only private parameter is the private key used for signing. It is strictly less powerful than MAFE if all parameters other than one are public unencrypted data.

2.4. ZKP Nesting

First, let us format our programs as follows. This can be read in plain language as: if the TX satisifies E, with the additional argument V, then sign the TX with the program specific key otherwise don't.

```
def F(p, I, TX, V)
    If E_I(TX, V):
        return Sign(Tweaked(p, E_I), TX)
    Else:
        return null
```

We can translate this to:

```
def F(p, I, TX, ZKP):
    If ZKPVerify(E_I)(TX, ZKP):
        return Sign(Tweaked(p, E_I), TX)
    Else:
        return null
```

and a separate ZKProve

```
def ZKProve(E, TX, V):
   ... elided
```

Which reads, if the provided ZKP of the ZK Circuit E is satisfied, sign.

This trick is handy to compress the size of the circuit needed to exist inside of FE. The ZKP is also "public parameters" as described above. The ZKP compresses the internal verification inside of F to another layer.

We could also further run the ZKP to compute things like hashes of transactions for the signature algorithm outside of the FE.

2.5. Using FE Covenants

Sending to a covenant:

(I) Find an encrypted opcode binary you trust with covenant E and PK P.
(II) Select the covenant instance parameters I you need.
(III) Tweak P by E_I
(IV) Send funds to E_I

Redeeming to a covenant:

(I) Plan to do a state transition with a specific TX and collect all verification data needed
(II) Pass the data and tx to the encrypted opcode binary
(III) use the resulting signature

3. Discussion

We've demonstrated that with single private argument dual input functional encryption general enough to verify a ZKP and make and ECDSA signature are sufficient to allow a trusted ceremony setup covenant opcode with no soft fork to Bitcoin.

These covenants are superior to most existing covenant techniques for a number of reasons, to name a few:

(I) All verification off chain.

(II) Fixed witness size 64 bytes signature, script size 32 bytes PK
(III) Composeable – signature can function as a literal opcode in a script with other conditions
(IV) Privacy – all covenants can look indistinguishable from normal key spends

They are superior to interactive oracle servers since after the trusted setup the binary can be used fully offline (and master keys deleted).

The downsides are threefold:

(I) Under-developed crypto make it impractical to use presently
(II) Trust assumption on setup
(III) Simple primitives like CTV remain bytewise and verification wise more efficient

References

[CI] M. Carla Mascia and Irene Villa. *A SURVEY ON FUNCTIONAL ENCRYPTION*.