Blockstream

2015 08 24

# Tree Signatures

Pieter Wuille

Multisig on steroids using tree signatures

Introduction

Technical details

Scaling up

Honeypots

Combining with Schnorr signatures

Implementation

Conclusion

## Introduction

When we started off determining what consensus improvements we would incorporate in our Alpha sidechain, we had a relatively long list of ideas. However, several things didn't make the cut due to time pressure or other reasons.

One of the changes that was included is the addition of a few simple opcodes[1] that used to exist in Bitcoin. One of them is OP_CAT, which concatenates two stack variables.

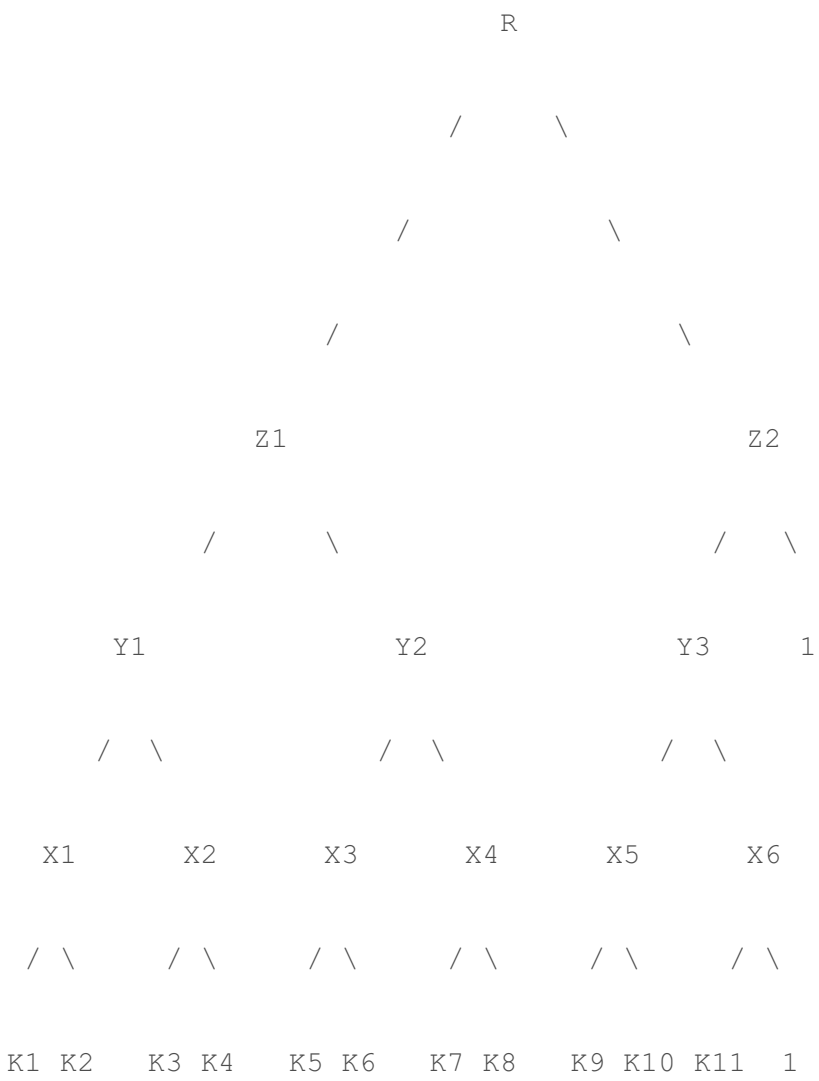
One that wasn't included is Merkleized Abstract Syntax Trees (MAST) for scripting. This is an idea that has been around for several years, but was also never implemented in Bitcoin. The basic idea is to rearrange the script's conditions into a tree and to only reveal the part that is actually used by the spender. It would enable many large, complex scripts, but requires a deep redesign of the script language. We've been thinking about this for a while, but this will not be ready soon.

However, briefly after the release of Alpha we realized that its script language, thanks to the addition of OP_CAT, enables a simple form of Merkle tree verification. Therefore, we can now emulate a subset of MAST without any changes to the consensus rules. All we need to do is use the script language in a way that wasn't anticipated. As you will see in the next sections, this enables some very large multisig[2] constructions with very low cost (in terms of opcodes executed and size on the chain).

The rest of this post covers the technical details, use cases and thoughts on implementations. Feel free to skip the tech if that's not your thing.

## Technical details

Imagine you want to create a 1-of-11 multisig (with 11 known public keys). You compute the SHA256 hashes of the public keys involved and put them in a Merkle tree. In the graph below, all symbol names represent 32-byte hashes, except the 1. The 1 is used as a simple constant instead of a right branch when there is none.




Using the script in our Alpha sidechain, we can build ascript that takes as input a public key, a signature, and a Merkle path. At verification time, it would use the Merkle path to prove that the public key belongs to a tree with root R, and that the signature checks out with that public key.

8 PICK SHA256 (SWAP IF SWAP ENDIF CAT SHA256)*4 <R> EQUALVERIFY CHECKSIG

Note that the inner 6 operations are repeated 4 times: once for each level in the tree.

To see why this works, follow the execution steps with the input below. It represents a

signature using the tenth key (whose hash is K10). The 8 last entries constitute the Merkle path. For each of the levels, from bottom to top, they contain the entry the running hash is combined with, plus a 0 or 1, indicating whether that is respectively a left or a right node.

| | |
|---|---|
| (scriptSig) | **<sig> <key 10> Z1 0 1 1 X6 1 K9 0** |
| 8 PICK | <sig> <key 10> Z1 0 1 1 X6 1 K9 0 **<key 10>** |
| SHA256 | <sig> <key 10> Z1 0 1 1 X6 1 K9 0 **K10** |
| SWAP | <sig> <key 10> Z1 0 1 1 X6 1 K9 **K10 0** |
| IF SWAP ENDIF | <sig> <key 10> Z1 0 1 1 X6 1 K9 K10 |
| CAT SHA256 | <sig> <key 10> Z1 0 1 1 X6 1 **X5** |
| SWAP | <sig> <key 10> Z1 0 1 1 X6 **X5 1** |
| IF SWAP END | <sig> <key 10> Z1 0 1 1 **X5 X6** |
| CAT SHA256 | <sig> <key 10> Z1 0 1 1 **Y3** |
| SWAP | <sig> <key 10> Z1 0 1 **Y3 1** |
| IF SWAP END | <sig> <key 10> Z1 0 **Y3 1** |
| CAT SHA256 | <sig> <key 10> Z1 0 **Z2** |
| SWAP | <sig> <key 10> Z1 **Z2 0** |

| | |
|---|---|
| IF SWAP END | <sig> <key 10> Z1 Z2 |
| CAT SHA256 | <sig> <key 10> **R** |
| <R> EQUALVERIFY | <sig> <key 10> |
| CHECKSIG | **1** |

## Scaling up

In the previous section, an example was given that shows how a 1-of-11 multisig can be implemented in Alpha script. However, Bitcoin script can already do 1-of-11 multisig just fine.

The advantage of this approach is that it scales logarithmically. Doubling the number of valid public keys only adds a constant 40 bytes to the transactions involved, and the number of (expensive) signature validations remains constant. Compare this with Bitcoin, where adding one single public key already adds 34 bytes.

Due to script's operation count limitations, this approach works in theory up to a 1-of-4,294,967,296 multisig, although the tree becomes annoyingly large to construct at that point. Up to 1-of-1,000,000 should be doable, though.

1-of-250,000 using this approach requires 886 bytes of Alpha Script. That's over 9,000 times smaller than the equivalent in Bitcoin Script, if it were allowed at all.

## Honeypots

One use case of such large multisig constructs is honeypots. Let's say you operate a 10,000-server cluster and want to leave some amount of BTC on each, hoping that an attacker would

steal the coins, and alert you of the breach.

If you can afford to lose up to 10 BTC for this purpose, and need to put a separate Bitcoin wallet on each, you would be limited to 1 mBTC on each machine, which is not a particularly attractive amount.

With a 10,000-wide multisig, however, you could assign the entire 10 BTC to such a large multisig script, while still leaving only a single key on each machine. Because all the keys are still separate, and the signature reveals which one was used, it means you can still detect which machine was compromised.

## Combining with Schnorr signatures

There are other use cases of large multisig constructs, but the approach described so far only supports 1-of-N, and not M-of-N.

However, this approach can be combined with an interesting property of Schnorr signatures[3] to obtain something more powerful. If you have a number of signatures for the same message (transaction), the signatures can be 'added' together to obtain a signature for the "sum" of their public keys.

This means that instead of using a 3-of-3 multisig script, we can simply use the sum of the 3 public keys involved as the "1-of-1" public key for a normal CHECKSIG. The signers then can all sign a spending transaction individually and add their signatures together, in order to produce a signature that is valid for the sum of the public keys. In other words: using Schnorr signatures, M-of-M multisig with arbitrarily high M, are reduced to a simple 1-of-1 as far as the blockchain is concerned. The downside is that two rounds are necessary, because the participants need to agree on a signing nonce first.

Perhaps by now you see how to combine them. Merkle tree keys support very large 1-of-N.

Schnorr signatures support very large M-of-M. This means that if we can write our spending conditions as a 1-of-(N possible M-of-M's), we can build a Merkle tree consisting of Schnorr combined public keys. Every leaf in the Merkle tree becomes a set of keys that all have to sign.

For example: 4-of-5 multisig with keys A,B,C,D,E can be written as:

| |
|---|
| (A and B and C and D) or<br><br>(A and B and C and E) or<br><br>(A and B and D and E) or<br><br>(A and C and D and E) or<br><br>(B and C and D and E) |

This becomes a Merkle tree with 5 leafs, each consisting of the sum of 4 public keys. In Alpha, this would require 286 Script bytes. In Bitcoin, a 4-of-5 multisig needs 490 bytes.

Also note that it always only requires a single (expensive) CHECKSIG operation. In Bitcoin Script, you always need one CHECKSIG per required signer (M, for an M-of-N multisig).

Some more examples:

| **Multisig** | **Bitcoin Script bytes** | **Alpha Script bytes** |
|---|---|---|
| 5-of-8 | 665 | 446 |
| 1-of-10 | 441 | 326 |
| | | |

| | | |
|---|---|---|
| 9-of-17 | 1263 | 766 |
| 3-of-20 | 927 | 606 |
| 12-of-23 | (1,686) | 1,006 |
| 98-of-100 | (10,582) | 886 |
| 2-of-1000 | (34,174) | 926 |
| 999-of-1000 | (106,955) | 926 |
| 1-of-10000 | (340,101) | 726 |
| 10000-of-10000 | (1,070,028) | 125 |

## Generalized threshold trees

The approach of creating a Merkle tree with leaves representing combinations of keys is not even limited to M-of-N conditions. For anything that can be expressed as a relatively small number of combinations of sets of keys that all have to sign, this approach is superior.

We have designed a simple language that describes such conditions for combinations of keys. A description is either:

- A public key in hex format, requiring a signature by that key.
- OR(a,b,c,...) where a, b, c... are other descriptions, requiring one of those to be satisfied.
- AND(a,b,c,...) where a, b, c… are other descriptions, requiring all of those to be satisfied.
- THRESHOLD(n,a,b,c,...) where n is a number and a, b, c… are other descriptions, requiring n of those to be satisfied.

This language can easily express conditions like “Either A and B sign, or 2 out of C, D and E sign”

## Properties

Greg Maxwell came up with a list of 5 interesting properties ($ACE^{UP}$) for signatures schemes in this context.[4]

- Accountability: Both OP_CHECKMULTISIG and these tree signatures allow all participants in the scheme to know who signed.
- Composability: Given two participants that each request signing by a complex description, can you construct a description that describes signing by one of those, and still have software that knows how to deal with it? Tree signatures are very good at this.
- Efficiency: Tree signatures have always better or equal transaction size and validation performance than OP_CHECKMULTISIG. However, this comes with higher signing time.
- Usability: Tree signatures require 2 rounds across all participants due to first needing a Schnorr signing nonce.
- Privacy: The participants in a tree signature and their number are much easier to hide, as adding dummies has very low cost.

## Implementation

See https://github.com/ElementsProject/elements/pull/48 for a patch for Alpha’s wallet to support tree signatures. New RPCs addtreesigaddress and createtreesig create P2SH addresses for scripts implementing public key trees, and signrawtransaction takes an extra argument to pass in the public key tree (which must be known at signing time).

## Conclusion

It is interesting to see how only a few changes to the scripting language can enable unexpected advantages. By combining Merkle trees of public keys with Schnorr combined signatures, we can support some combinations of very large M-of-N multisig in an efficient way. As part of our work with Sidechain Elements, this will be available in the Alpha codebase soon, and support

for it will be improved in potential successors.

---

[1] See https://github.com/ElementsProject/elementsproject.github.io#new-opcodes

[2] A transaction that locks coins such that it requires multiple signatures from multiple signers. An M-of-N multisig construction requires M signatures from N different signers whose public keys are known.

[3] See the “Schnorr Signature validation” element on http://elementsproject.org/

[4] see https://www.youtube.com/watch?v=TYQ-3VvNCHE