Strong Federations: Thresholds and Byzantine Security

•••

Mark Friedenbach
Blockstream

Thresholds Tradeoffs

When setting up a strong federation, the choice of threshold determines the availability and security properties of the network.

- Availability: how many functionaries can be offline and the network is still operational -- blocks signed, and withdraws processed.

- Byzantine security: how many functionaries need to be dishonest or compromised for there to be a chain split.

The smallest production threshold we use at Blockstream is 8 of 11. So I will use this as an example.

Threshold Tradeoffs: 8 of 11

Availability: 8 of 11 functionaries need to be online, so the network can tolerate up to 3 functionaries being offline at any time.

If 4 or more functionaries are offline, the federation stops signing blocks / processing withdraws.

Threshold Tradeoffs: 8 of 11

Byzantine security: How many dishonest functionaries are required to split the network? Requires:

8 of 11 on side A 8 of 11 on side B

Requires 5 dishonest or compromised functionaries, when all 11 are online.

3 honest on side A 3 honest on side B

5 dishonest on both sides

Advanced Elements Script

Mark Friedenbach
Blockstream

Feature: CHECK-SIG-FROM-STACK

<sig> <data> <pubkey> CHECK-SIG-FROM-STACK

Behaves exactly the same as CHECK-SIG, but verifies a signature against a message on the stack, rather than the spending transaction itself.

Used by Russell O'Connor to create "covenants" on Elements Alpha:

https://blockstream.com/2016/11/02/covenants-in-elements-alpha.html

Feature: DETERMINISTIC-RANDOM

<seed> <minimum> <maximum> DETERMINISTIC-RANDOM

Takes a seed value, a minimum, and a maximum off the stack. Computes a pseudorandom value with uniform distribution from the range of [minimum, maximum). Always computes the same value given the same inputs.

Example use case: probabilistic payments. Rather than paying Alice paying Bob 1¥ in a microtransaction, she pays 1,000¥ with a 0.001 chance of the transaction being valid.

Example: Probabilistic Payments

Alice wants to pay bob 1000x the requested payment, at 1/1000 the chance of being valid. She wants to do this because average this is the same as making normal payments, but with 1/1000th the fee.

Bob accepts the payment, even if it is one of the 99.9% that do not validate, because he knows that it had a 0.01% chance of being worth 1000x as much. Bob prefers this because on average Bob gets the same income, but without lots of small UTXOs.

Alice MUST commit to the payment without knowing whether it will validate.

Bob MUST not be able to modify the transaction to make it validate.

Example: Probabilistic Payments

- rAlice is a 256-bit number picked by Alice
- rBob is a 256-bit number picked by Bob
- hBob is HASH256(rBob)

1. rAlice|hBob signed by Alice
2. HASH256(rBob) equals hBob
3. random(HASH256(rAlice|rBob), 0, 1000) is zero
4. TX signed by Bob

Example: Probabilistic Payments

1. rAlice|hBob signed by Alice

```
sigAlice rAlice                ||  hBob CAT pubAlice CHECK-SIG-FROM-STACK-VERIFY
sigAlice rAlice hBob           ||  CAT pubAlice CHECK-SIG-FROM-STACK-VERIFY
sigAlice rAlice|hBob           ||  pubAlice CHECK-SIG-FROM-STACK-VERIFY
sigAlice rAlice|hBob pubAlice  ||  CHECK-SIG-FROM-STACK-VERIFY
                               ||
```

(continued)

Example: Probabilistic Payments

2. HASH256(rBob) equals hBob

```
rBob           ||  HASH256 hBob EQUALVERIFY
H(rBob)        ||  hBob EQUALVERIFY
H(rBob) hBob   ||  EQUALVERIFY
               ||
```

(continued)

Example: Probabilistic Payments

3. random(HASH256(rAlice|rBob), 0, 1000) is zero

```
rAlice rBob             ||  CAT HASH256 0 1000 DETERMINISTIC-RANDOM 0 1 WITHIN VERIFY
rAlice|rBob             ||  HASH256 0 1000 DETERMINISTIC-RANDOM 0 1 WITHIN VERIFY
H(rAlice|rBob)          ||  0 1000 DETERMINISTIC-RANDOM 0 1 WITHIN VERIFY
H(rAlice|rBob) 0        ||  1000 DETERMINISTIC-RANDOM 0 1 WITHIN VERIFY
H(rAlice|rBob) 0 1000   ||  DETERMINISTIC-RANDOM 0 1 WITHIN VERIFY
<random,0,1000>         ||  0 1 WITHIN VERIFY
<random,0,1000> 0       ||  1 WITHIN VERIFY
<random,0,1000> 0 1     ||  WITHIN VERIFY
<bool>                  ||  VERIFY
                        ||
```

(continued)

Example: Probabilistic Payments

4. TX signed by Bob

```
sigBob          ||  pubBob CHECK-SIG
sigBob pubBob   ||  CHECK-SIG
<bool>          ||
```

(done)

Feature: Merkle Branch Verification

Verify that a given element is contained within a committed Merkle tree.

```
[<leaf> <path> <root> CMB]      =>   <bool>
[<leaf> <path> <root> CMBV]     =>   # empty
```

The inputs are:

1. (pubkey) The root of the Merkle tree.

2. (witness) The hash of an element of the tree (usually a terminal node / leaf).

3. (witness) A 'proof' object containing the path through the tree and hash values for branches not taken.

Feature: Merkle Branch Verification

The opcode uses the element and its proof to recompute the root hash value, which is compared against the expected value given in the first argument. The result is either placed on the stack (CMB), or VERIFY semantics cause the script to terminate if they differ (CMBV).

> scriptPubKey => OVER HASH256 *root* CMBV ...
> scriptWitness => ... *leaf proof*

Feature: Tail-call recursion (A better *EVAL*)

Being able to execute sub-scripts enables general computation. Combined with existing stack manipulation opcodes, an EVAL operation allows for the construction of various loops or recursive subroutine calls.

A tail-call recursion is when the EVAL operation is the very last instruction in a script. No state needs to be maintained -- simply jump into the new script.

Not implemented via an opcode, but rather re-defining behaviour for non-clean stacks.

Feature: Tail-call recursion (A better *EVAL*)

If execution terminates with more than one element on the stack, treat the topmost element as a script and call it.

```
Script    => [] # empty/end-of-script
Stack     => <arg1> <arg2> ... <argN> [subscript]
```

Becomes:

```
Script    => [subscript]
Stack     => <arg1> <arg2> ... <argN>
```

Example: Mathematical Challenge

triangle(5) = 5 + 4 + 3 + 2 + 1

X

X X

X X X

X X X X

X X X X X

[NB: triangle(N) = (N^2 + N) / 2]

Example: Mathematical Challenge

C:

```
unsigned int
triangle(unsigned int n)
{
     if (n == 0)
          return 0;
     else
          return n + triangle(n-1);
}
```

Haskell:

```
tri x 0  = x

tri x n  = tri (x+n) (n-1)

triangle n = tri 0 n
```

Example: Mathematical Challenge

C:

```
unsigned int
triangleX(unsigned int x,
          unsigned int n)
{
     if (n == 0)
          return x;
     else
          return triangleX(x+n, n-1);
}
```

```
unsigned int
triangle(unsigned int n)
{
     return triangleX(0, n);
}
```

Example: Mathematical Challenge

```
[tri] x 0  ==  ROT 2DROP

[tri] x 0  ||  ROT 2DROP
x 0 [tri]  ||  2DROP
X
     (done)
```

```
[tri] x n          ==  SWAP OVER ADD SWAP 1SUB 2 PICK

[tri] x n          ||  SWAP OVER ADD SWAP 1SUB 2 PICK
[tri] n x          ||  OVER ADD SWAP 1SUB 2 PICK
[tri] n x n        ||  ADD SWAP 1SUB 2 PICK
[tri] n x+n        ||  SWAP 1SUB 2 PICK
[tri] x+n n        ||  1SUB 2 PICK
[tri] x+n n-1      ||  2 PICK
[tri] x+n n-1 2    ||  PICK
[tri] x+n n-1 [tri]
     (tail-call)
```

Example: Mathematical Challenge

```
[tri] x n = DUP 0 EQUAL
            IF
                ROT 2DROP [...finally...]       # [tri] x 0
            ELSE
                SWAP OVER MUL SWAP 1SUB 2 PICK  # [tri] x n
            ENDIF

[tri] x n       ||  DUP 0 EQUAL IF ...if... ELSE ...else... ENDIF
[tri] x n 0     ||  EQUAL IF ...if... ELSE ...else... ENDIF
[tri] x <bool>  ||  IF ...if... ELSE ...else... ENDIF
```

Example: Mathematical Challenge

Contract address commits to r, and code.
to claim, provide n such that triangle(n) = r.

```
Witness: <n>
Script:  <r> SWAP
         [
             DUP 0 EQUAL
             IF
                 ROT 2DROP EQUAL
             ELSE
                 SWAP OVER ADD SWAP 1SUB 2 PICK
             ENDIF
         ]
         0 ROT 2 PICK
```

Example: Mathematical Challenge

```
n                 ||  r SWAP [tri] 0 ROT 2 PICK
n r               ||  SWAP [tri] 0 ROT 2 PICK
r n               ||  [tri] 0 ROT 2 PICK
r n [tri]         ||  0 ROT 2 PICK
r n [tri] 0       ||  ROT 2 PICK
r [tri] 0 n       ||  2 PICK
r [tri] 0 n 2     ||  PICK
r [tri] 0 n [tri]
    (tail-call)
```

Example: Mathematical Challenge

```
# triangle(100) = 5050

$ e-cli compilescript '5050 SWAP [DUP 0 EQUAL IF ROT 2DROP EQUAL ELSE SWAP OVER ADD
SWAP 1SUB 2 PICK ENDIF] 0 ROT 2 PICK'

{
  "hex": "02ba137c10760087637b6d87677c78937c8c527968007b5279",
  "length": 25,
  "asm": "5050 OP_SWAP 760087637b6d87677c78937c8c527968 0 OP_ROT 2 OP_PICK",
  "type": "nonstandard"
}
```

Example: Mathematical Challenge

```
$ e-cli decodescript 02ba137c10760087637b6d87677c78937c8c527968007b5279

{
  "asm": "5050 OP_SWAP 760087637b6d87677c78937c8c527968 0 OP_ROT 2 OP_PICK",
  "type": "nonstandard",
  "p2sh": "XMFFTyUyewNGZQtkHeoDm3iX8wQtpos9CC"
}

$ TXID=$(e-cli sendtoaddress XMFFTyUyewNGZQtkHeoDm3iX8wQtpos9CC 10)
$ echo $TXID
2081d9366e5b0b1b6a31f50ac44913c5fffc0a6be4daa568dcc09f5799a45e1b
```

Example: Mathematical Challenge

```
$ e-cli getrawtransaction $TXID true
…
"vout": [
{
      "value": 10.00000000,
      "asset": "09f663de96be771f50cab5ded00256ffe63773e2eaa9a604092951cc3d7c6621",
      "n": 0,
      "scriptPubKey": {
            "asm": "OP_HASH160 6c8f471a263085dbc972f39fe1feaf72317cf778 OP_EQUAL",
            "hex": "a9146c8f471a263085dbc972f39fe1feaf72317cf77887",
            "reqSigs": 1,
            "type": "scripthash",
            "addresses": [
                  "XMFFTyUyewNGZQtkHeoDm3iX8wQtpos9CC"
            ]
      }
},
…
$ N=0
```

Example: Mathematical Challenge

```
$ CTADDR=$(e-cli getnewaddress)
$ echo $CTADDR
CTEmjrxU2H7rcG4uWGZWZDzUPNHyGDGHEfiP27gvWfVK2jPQ5mquaqvRs1qrP4iTA5Z8K8zxa68jiR2b

$ e-cli validateaddress $CTADDR
{
    "isvalid": true,
    "scriptPubKey": "76a91425f6e4074d7d67d0466a6e1df73802386f687bed88ac",
    "confidential_key":
"024be39a5ddb0f372a8bd587b6afa43a5524984387fbe02a6a517635cb4f45aba5",
    "unconfidential": "2dctV9tKiFQL5cqy3FSQkfB3vj6WPv6fTv6",
    "ismine": true,
…
}
$ ADDR=2dctV9tKiFQL5cqy3FSQkfB3vj6WPv6fTv6
```

Example: Mathematical Challenge

```
$ RAWTX=$(e-cli createrawtransaction
'[{"txid":"'$TXID'","vout":'$N',"sequence":4294967295}]'
'{"'$ADDR'":9.9999,"fee":0.0001}')

$ e-cli compilescript '100 [5050 SWAP [DUP 0 EQUAL IF ROT 2DROP EQUAL ELSE SWAP OVER
ADD SWAP 1SUB 2 PICK ENDIF] 0 ROT 2 PICK]'
{
  "hex": "01641902ba137c10760087637b6d87677c78937c8c527968007b5279",
  "length": 28,
  "asm": "100 02ba137c10760087637b6d87677c78937c8c527968007b5279",
  "type": "nonstandard"
}

$ RAWTX=$(e-cli insertscriptsig $RAWTX 0
01641902ba137c10760087637b6d87677c78937c8c527968007b5279)
```

Example: Mathematical Challenge

```
$ e-cli sendrawtransaction $RAWTX
972c2c1de0617b99bdc74963298377ad91b274d23947c7aacc9072d183e3280b

$ e-cli gettransaction 972c2c1de0617b99bdc74963298377ad91b274d23947c7aacc9072d183e3280b
...
  "details": [
    {
      "account": "",
      "address": "2dctV9tKiFQL5cqy3FSQkfB3vj6WPv6fTv6",
      "category": "receive",
      "amount": 9.99990000,
      "asset": "09f663de96be771f50cab5ded00256ffe63773e2eaa9a604092951cc3d7c6621",
      "label": "",
      "vout": 0
    }
  ],
...
```

Feature: Bitmask signature mode

Recall bitcoin's signature hash (sighash) modes:

SIGHASH_ALL: sign all outputs.

SIGHASH_SINGLE: sign only the output at the same position as the input

SIGHASH_NONE: sign none of the outputs

SIGHASH_ANYONECANPAY: sign only this input

(default): sign all inputs

The first three output modes can be optionally combined with the ANYONECANPAY flag for a total of six possible signing modes.

Feature: Bitmask signature mode

This *fixed function* signing pipeline allows for some interesting applications such as atomic swaps or assurance contracts. However restricting ourselves to these few signing modes is often inefficient and unnecessarily limiting. For example, the following use cases are difficult to support:

1. Signing both the desired output and a change address, but not others.
2. Having all pieces of an assurance contract sign the same output, allowing the non-interactive construction of a crowdfund transaction.
3. Paying multiple parties in a smart contract, while still allowing other outputs to be updated.

Feature: Bitmask signature mode

Bitmask signature mode allows each signature to specify two bitmasks: one for the inputs, one for the outputs.

1. For each input, if its corresponding bit is set in the input bitmask, keep that input. If the bit is clear, remove it.
2. For each output, if its corresponding bit is set in the output bitmask, keep that output. Otherwise, remove it.
3. The input being signed must have its bit set!

The bitmasks are in the witness, not covered by the signature. This allows the bitmasks to be updated as new inputs and outputs are added.