Smart Contracts

...

Mark Friedenbach
Blockstream

Computation vs Validation

Role of a blockchain is to order and timestamp transactions, not to create them.

Typically to verify a computation requires fewer features and less effort than to do the original computation.

Example: prime factorization. Factoring large numbers into prime components requires complex algorithms and tremendous computation. But verifying that a number ***N*** is the product of two primes **a** and **b** is simple multiplication: ***N*** =?= **a*b** .

Example: cryptographic break bounty

2DUP EQUAL NOT VERIFY SHA1 SWAP SHA1 EQUAL

1. Takes two items from the stack as inputs.
2. Verify that both inputs are different byte strings.
3. Hash both inputs using SHA-1.
4. Verify that both hash to the same value.

This is a bounty for the demonstration of a SHA-1 hash collision. ~2.5 BTC were sent to this script in total, and claimed in February of 2017 when Google demonstrated the first SHA-1 collision.

It took Google 6,500 years of CPU and 110 years of GPU computation to generate the collision. Checking the spend tx takes a few milliseconds on a laptop.

“Smart” Contracts

A contract is an agreement between two or more parties, enforced by an agreed form of dispute mediation.

Smart contracts are those which use machines to mediate disputes, instead of courts or professional arbitrators.

Most smart contracts involve ownership of attested assets, or bitcoin, that are locked up for the duration of the contract, as collateral.

Example: escrow with timeout

Problem: Alice buys a house from Bob, but wants 30 days for inspections. Control over the funds is shared with an escrow agent until an agreement is reached, but with a timeout.

```
IF
    2 <pubAlice> <pubBob> <pubEscrow> 3 CHECK-MULTI-SIG
ELSE
    <30d> CHECK-SEQUENCE-VERIFY DROP <pubBob> CHECK-SIG
ENDIF
```

Example: tree signatures

Remember CHECK-MERKLE-BRANCH-VERIFY?

<proof> <leaf> <root> CHECK-MERKLE-BRANCH-VERIFY

Takes a Merkle root, a hash of one of its leaves, and the path through the tree from the root to the leaf. Confirms that the proof is valid by rebuilding the root hash and seeing if it matches the value given. Only really useful if the root hash is committed to in the contract address!

(We'll abbreviate it CMBV from here on.)

Example: tree signatures

We can use this to make a compact multi-signature contract with greater privacy. Take the following spend condition as an example:

(Alice && Bob) || 2of(Bob, Carol, Dave)

“2of()” means the threshold operator -- any two of the three keys specified would be sufficient for spending.

Example: tree signatures

(Alice && Bob) || 2of(Bob, Carol, Dave)

The 4 possible ways of spending this script are:

- Alice && Bob
- Bob && Carol
- Bob && Dave
- Carol && Dave

Example: tree signatures

(Alice && Bob) || 2of(Bob, Carol, Dave)

We build a script for each of these:

- [2 <pubAlice> <pubBob> 2 CHECK-MULTI-SIG]
- [2 <pubBob> <pubCarol> 2 CHECK-MULTI-SIG]
- [2 <pubBob> <pubDave> 2 CHECK-MULTI-SIG]
- [2 <pubCarol> <pubDave> 2 CHECK-MULTI-SIG]

Example: tree signatures

Now we make a Merkle tree of the possibilities and compute the root hash of this tree. The root is then used in the following contract script:

```
OVER HASH256 <root> CMBV # tail-call eval
```

If Bob and Dave use their keys to spend, they would use the following witness:

```
0 <sigBob> <sigDave> [ 2 <pubBob> <pubDave> 2 CHECK-MULTI-SIG ] <proof>
```

Observers only learn about the single condition that was used to spend the coin, without revealing the other policies, or cluttering the block chain with unexecuted code.

Example: tree signatures

```
... [subscript] <proof>                        ||  OVER HASH256 <root> CMBV
... [subscript] <proof> [subscript]            ||  HASH256 <root> CMBV
... [subscript] <proof> H(subscript)           ||  <root> CMBV
... [subscript] <proof> H(subscript) <root>    ||  CMBV
... [subscript]                                ||

[subscript] :
     scriptPubKey: [2 <pubBob> <pubDave> 2 CHECK-MULTI-SIG]
     scriptSig:    0 <sigBob> <sigDave>

     0 <sigBob> <sigDave> 2 <pubBob> <pubDave> 2 CHECK-MULTI-SIG
```

Example: tree signatures

```
$ e-cli validateaddress $(e-cli getnewaddress)
{
…
     "address":
"CTEsmDfug7XXxUoCmqNtyvn47uWV9HWdpR1YhnqYx8iodCEYk7pV4zmEVtqJhg4vU89VHH6jw7MLniNQ",
     "scriptPubKey": "76a914c717e2f52c4d77c409d476efa37e9f8133d40ec288ac",
     "pubkey": "02d91b1ccd5e40c37cdbf4c01c1289e0125d02af3e295ed3a194e0aff1d087bce6",
…
}
pubAlice="02d91b1ccd5e40c37cdbf4c01c1289e0125d02af3e295ed3a194e0aff1d087bce6"

# Use "pubkey" for Alice.
# Do the same for Bob, Carol, Dave
```

Example: tree signatures

```
$ e-cli compilescript "2 [0x$pubAlice] [0x$pubBob] 2 CHECKMULTISIG"
{
    "hex":
"522102d91b1ccd5e40c37cdbf4c01c1289e0125d02af3e295ed3a194e0aff1d087bce62103e475749b20
7650cb024ad7e9c7baa374a161957ff8e913a119b6df58f5894a30",
    "length": 69,
    "asm": "2 02d91b1ccd5e40c37cdbf4c01c1289e0125d02af3e295ed3a194e0aff1d087bce6
03e475749b207650cb024ad7e9c7baa374a161957ff8e913a119b6df58f5894a30",
    "type": "nonstandard"
}
scriptAliceBob=522102d91b1ccd5e40c37cdbf4c01c1289e0125d02af3e295ed3a194e0aff1d087bce6
2103e475749b207650cb024ad7e9c7baa374a161957ff8e913a119b6df58f5894a30

# scriptBobCarol, scriptBobDave, scriptCarolDave
```

Example: tree signatures

```
$ e-cli merklebranch '["'$scriptAliceBob'", "'$scriptBobCarol'",
                      "'$scriptBobDave'", "'$scriptCarolDave'"]'
{
  "root": "f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8c"
}
ROOT=f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8c

$ e-cli compilescript "OVER HASH256 [0x$ROOT] CHECKMERKLEBRANCHVERIFY"
{
  "hex": "78aa20f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8cc5",
  "asm": "OP_OVER OP_HASH256
f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8c
OP_CHECKMERKLEBRANCHVERIFY",
}
scriptPubKey=78aa20f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8cc5
```

Example: tree signatures

```
$ e-cli decodescript $scriptPubKey
{
  "asm": "OP_OVER OP_HASH256
f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8c
OP_CHECKMERKLEBRANCHVERIFY",
  "type": "nonstandard",
  "p2sh": "XEs2ycMxu54MTLJ7vjUbu9gTrdBvCQdpwG"
}
```

Example: tree signatures

```
$ e-cli merklebranch '["'$scriptAliceBob'", "'$scriptBobCarol'",
                                "'$scriptBobDave'", "'$scriptCarolDave'"]' 2
{
  "root": "f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8c",
  "branch": [
    "361e016ae64027c82a309ebeaeeb2ce64dffef3e1411eb7ed00272b228a5f055",
    "18dc77b0ea60ba33aaeeac35143b56c18471e67c078e53362d494158f557494d"
  ],
  "path": 2,
  "proof":
"55f0a528b27202d07eeb11143eefff4de62cebaebe9e302ac82740e66a011e364d4957f55841492d3653
8e077ce67184c1563b1435aceeaa33ba60eab077dc1802"
}
PROOF=55f0a528b27202d07eeb11143eefff4de62cebaebe9e302ac82740e66a011e364d4957f55841492
d36538e077ce67184c1563b1435aceeaa33ba60eab077dc1802
```

Example: tree signatures

```
sigBob=00 sigDave=00
$ e-cli compilescript "0 [0x$sigBob] [0x$sigDave] [2 [0x$pubBob] [0x$pubDave] 2
CHECKMULTISIG] [0x$PROOF] [OVER HASH256 [0x$ROOT] CHECKMERKLEBRANCHVERIFY]"
{
  "hex": "...",
  "length": 180,
  "asm": "0 <sigBob> <sigDave>
522103e475749b207650cb024ad7e9c7baa374a161957ff8e913a119b6df58f5894a302103ce91abb9c36
930934cd3c6fcf5fdb9fd7158ba733d70323b1ed49231a953846b52ae
55f0a528b27202d07eeb11143eefff4de62cebaebe9e302ac82740e66a011e364d4957f55841492d36538
e077ce67184c1563b1435aceeaa33ba60eab077dc1802
78aa20f9037bec5b42eae30dd6daabcf532272ff1f2ae829bcf1e99acca094cc2b2c8cc5",
  "type": "nonstandard"
}
```

A general approach to smart contracting

Designing and executing a smart contract involves the following steps:

1. Specify the set of possible outcomes, and the necessary conditions for each.
2. While a contract is in effect, some outcomes become enabled as their preconditions are met.
3. The actual outcome is whichever confirms on the block chain, to the exclusion of others.

NB: Rather than resolve a contract entirely, a multi-step contract can change the set of outcomes, disabling some and enabling others via an on-chain transaction.

A general template for smart contracts

Every smart contract can be reduced to "everybody signs, or <dispute resolution>"

```
IF          N <pub1> <pub2> ... <pubN> N CHECK-MULTI-SIG
ELSE        [...code...]
ENDIF
```

If everyone is available and in agreement about the final state, they simply sign and follow the first path. Otherwise, dispute resolution occurs in the “...”

This provides an *early out* optimization -- most amicable contract resolutions result in just a few signature checks, with no complex computation.

Exception: contracts without fixed participant set

Some smart contracts do NOT have a fixed set of participants.

Most notable example is the sidechain peg: anyone can peg in or out coins at any time. So what set of keys is used in the top-level “everyone signs” subscript? There is no set of keys that can be precommitted to.

Smart contracts with *dynamic participant sets* must use the explicit dispute resolution process for all resolutions.

But, these sorts of contracts are small in number, even if some (like the sidechain peg) are critically important.

The general template, generalized

Let's combine our escrow contract:

```
IF        2 <pubAlice> <pubBob> <pubEscrow> 3 CHECK-MULTI-SIG
ELSE      <30d> CHECK-SEQUENCE-VERIFY <pubAlice> CHECK-SIG
ENDIF
```

With our tail-call eval tree signatures:

```
Script:   OVER HASH256 CMBV # tail-call eval
Witness:  <arg1> <arg2> ... <argN> <subscript> <proof>
```

The general template, generalized

Two execution pathways, for the two possible resolutions:

> 2 <pubAlice> <pubBob> <pubEscrow> 3 CHECK-MULTI-SIG
> <30d> CHECK-SEQUENCE-VERIFY DROP <pubBob> CHECK-SIG

Hash each script, and provide just the one needed at resolution.

For example:

> Script: OVER HASH256 <root> CMBV # tail-call eval
> Witness: <sigBob> [<30d> CSV DROP <pubBob> CHECK-SIG] <proof>

MAST: Merkelized Abstract Syntax Trees

Programming languages are parsed into *syntax trees*. Conditionals like if/else/endif are branching nodes in this tree.

If you build a hash tree out of this data structure, you can then only need to reveal the branches actually executed, while still being able to verify that the program executed is a fragment of the original. Non-executed branches reduce to a single hash.

```
IF      <hash>          # not executed
ELSE    [...code...]# executed
ENDIF
```

MAST: Merkelized Abstract Syntax Trees

Taking this a step further, if one can enumerate and "flatten" all possible execution pathways to a single linear script, then execution becomes a matter of selecting which script to run, and running it from start to finish.

```
Script:     OVER HASH256 <root> CMBV # tail-call eval
Witness:  <arg1> <arg2> ... <argN> [subscript] <proof>
```

Hey, that looks like our general template!

MAST: Merkelized Abstract Syntax Trees

This is the concept of *Merkelized abstract syntax trees*.

- All scripts, no matter how complex, can reduce to a set of possible execution pathways, linearized sub-scripts.
- A Merkle tree is built from all of these possible scripts, and committed to in the contract address.
- At spend, you specify the subscript and its Merkle proof, and whatever arguments are required to satisfy the subscript.

There are computational limits to how big a Merkle tree can be constructed in reasonable time (~1bn leafs). Most practical smart contracts are unlikely to reach these limits. But if they do, e.g. because of combinatorial explosion, this trick can be combined with other approaches.

Additional Resources

- BIP-65 (CHECK-LOCK-TIME-VERIFY) provides examples of various lock-time related smart contracts -- escrow, two-factor wallets, and time-locked refunds. BIP-112 (CHECK-SEQUENCE-VERIFY) contains relative time-lock versions of the same.
- Tree Signatures https://blockstream.com/2015/08/24/treesignatures.html
- Covenants in Elements Alpha https://blockstream.com/2016/11/02/covenants-in-elements-alpha.html
- Lightning Payment Channels http://lightning.network/
- Lighthouse: Assurance Contracts for Crowdfunding https://github.com/vinumeris/lighthouse/blob/master/docs/Design%20doc.md