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** .

Computation vs Validation 計算量 vs 検証

ブロックチェーンの役割は、トランザクションを整理しタイムスタンプすることであって、トランザクションを作成することではありません。

一般的に、計算を検証するためには元の計算を行うよりも少ない機能と労力しか必要としません。

例 : 素因数分解。大きな数を素因数分解するには、複雑なアルゴリズムと膨大な計算を必要とします。しかし、数 ***N*** が2つの素数 **a** と **b** の積であることを検証するのは単純な乗算です: ***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.

Example: cryptographic break bounty 例: 暗号解読賞金

2DUP EQUAL NOT VERIFY SHA1 SWAP SHA1 EQUAL

1. 入力としてスタックから2つのアイテムを取り出します。
2. 両方の入力が違う文字列であることを確認します。
3. 両方の入力をSHA-1でハッシュします。
4. 両方のハッシュが等しいことを確認します。

これは、SHA-1ハッシュの衝突を実証するための賞金です。合計で2.5 BTC弱がこのスクリプトに送られ、Googleが最初のSHA-1衝突を実証した2017年の2月に請求されました。

Googleは衝突を発生させるために6,500年分のCPUと110年分のGPUの計算量が必要でした。Txの確認は、ラップトップでも数ミリ秒です。

“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.

“Smart” Contracts “スマート” コントラクト

コントラクトとは、合意された形式の紛争調停によって強制される複数の当事者間の合意です。

スマートコントラクトとは、裁判所やプロの仲裁人の代わりに、紛争を仲介するために機械(コンピュータ)を使用するものです。

ほとんどのスマートコントラクトには、そのコントラクトの期間中に担保として拘束されている、証明された資産またはビットコインの所有権が含まれます。

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: escrow with timeout 例: 期限付のエスクロー

問題：アリスはボブから家を購入しますが、検査のために30日を要します。資金のコントロールは、合意に達するまではアリス、ボブ、エスクローエージェントの3者で共有されますが、それには期限(30日以内に検査完了しなければボブのものとなる)があります。

```
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 例: ツリー署名

CHECK-MERKLE-BRANCH-VERIFY を覚えていますか?

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

Merkleルート、その1つのリーフのハッシュ、ルートからリーフまでのツリーのパスをとります。ルートハッシュを再構築し、与えられた値と一致するかどうかを確認することで、プルーフが有効であることを確認します。ルートハッシュが契約アドレスにコミットされている場合にのみ本当に便利です!

(これ以降、CMBVと略します。)

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 例: ツリー署名

これを使用して、より大きなプライバシーでコンパクトなマルチ署名のコントラクトを作成することができます。 例として、次のような支払い条件を考えてみましょう。

(Alice && Bob) || 2of(Bob, Carol, Dave)
(アリスとボブの2人) または (ボブとキャロルとデーブの3人のうちの2人)

“2of()” は、閾値演算子を意味します。指定された3つのキーのうちの2つあれば支払いに十分です。

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 例: ツリー署名

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

このスクリプトを実行する4つの方法は以下のとおりです:

- アリスとボブ
- ボブとキャロル
- ボブとデーブ
- キャロルとデーブ

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 例: ツリー署名

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

それぞれのスクリプトを以下のように作ります:

- [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 例: ツリー署名

今度は、(前ページの)可能性のあるMerkleツリーを作成し、このツリーのルートハッシュを計算します。ルートは次のコントラクトスクリプトで使用されます:

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

ボブとデーブが自分の鍵を使って支払う場合、次のウィットネスを使用します:

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

オブザーバーは、他のポリシーを明らかにしたり、実行されていないコードでブロックチェーンを混乱させたりすることなく、コインを使うために使用された単一の条件についてのみを知りえます。

Example: tree signatures 例: ツリー署名

```
... [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 例: ツリー署名

```
$ 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 例: ツリー署名

```
$ 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 例: ツリー署名

```
$ 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 例: ツリー署名

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

Example: tree signatures 例: ツリー署名

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

Example: tree signatures 例: ツリー署名

```
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 approach to smart contracting

スマートコントラクトの一般的なアプローチ

スマートコントラクトの設計と実行には、次の手順が必要です:

1. 可能な結果のセットとそれぞれに必要な条件を特定します。
2. コントラクトが成立している間は、前提条件が満たされたときにいくつかの結果が有効になります。
3. 実際の結果は、他を排除してブロックチェーンで確認されたものです。

注:複数ステップのコントラクトでは、コントラクトを完全に決定するのではなく、オンチェーン・トランザクションを介して結果のセットを変更したり、一部を無効にしたり、他のものを有効にすることができます。

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.

A general template for smart contracts

スマートコントラクトの一般的なテンプレート

すべてのスマート契約は「全員が署名するか、<紛争の解決>」という形に還元することができます。

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

全員が、最終的な状態について利用可能で合意した場合、全員が署名して最初のパスを辿ります。それ以外の場合、紛争解決が発生します。

これにより出力を早める最適化が提供されます。最も基本的なコントラクトの決定では、複雑な計算を必要とせずに、わずかな署名チェックが行われます。

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.

Exception: contracts without fixed participant set

例外：固定参加者が設定されていないコントラクト

固定された参加者セットがないスマートコントラクトもあります。

最も注目すべき例はサイドチェーンペグです：誰でもいつでもコインをペグイン・ペグアウトできます。どのようなキーのセットがトップレベルの“全員署名”サブスクリプトで使われるでしょうか？ 事前にコミットできるキーのセットはありません。

動的参加者セットのスマートコントラクトは、すべての決定について明確な紛争解決プロセスを使用する必要があります。

しかし、いくつか（サイドチェーンペグのような）が非常に重要であっても、これらの種類のコントラクトの数は少ないです。

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 一般的なテンプレート、一般化

エスクロー契約を組み合わせましょう:

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

組み合わせるのはテールコールEVALツリーの署名です:

```
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>

The general template, generalized 一般的なテンプレート、一般化

2つの実行パスと2つの可能な結果：

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

各スクリプトをハッシュし、結果に必要なものをひとつだけ提供します。

例えば：

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 マークル化抽象構文木

プログラミング言語は構文木に解析されます。if/else/endif のような条件は、このツリーの分岐ノードです。

このデータ構造からハッシュ・ツリーを構築すると、実行されたプログラムが元の断片であることを確認できるので、あとは実際に実行されたブランチを明らかにするだけです。実行されていないブランチは、単一のハッシュになります。

```
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 マークル化抽象構文木

これをさらに進めると、実行可能なすべての実行パスを単一の線形スクリプトに列挙して「平坦化」することができれば、実行はスクリプトを選択し最初から最後まで走らせるという問題になります。

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

前述の一般的なテンプレートに似ていますね!

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.

MAST: Merkelized Abstract Syntax Trees マークル化抽象構文木

以下はマークル化抽象構文木の概念です。

- すべてのスクリプトは、どんなに複雑であっても実行可能な一連の実行経路、線形化されたサブスクリプトに還元することができます。
- マークル木は、これらのすべての可能なスクリプトから構築され、コントラクトのアドレスにコミットされます。
- 支払う時は、サブスクリプトとそのマークルプルーフを指定し、そのサブスクリプトを満たす引数が必要です。

マークル木が妥当な時間に構築できる大きさには計算上の制限(~10億リーフ)があります。実用的なスマートコントラクトでは制限に達する可能性は低いですが、達する場合は組合せ爆発等のため、このトリックは他のアプローチと組み合わせることになります。

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