sCrypt

Release 0.0.1

Jun 11, 2020

Introduction

sCrypt (pronounced “ess crypt”) is a high-level smart contract language for Bitcoin SV. Bitcoin supports smart contract with its Forth-like stack based Script language. However writing smart contract in native Script is cumbersome and error-prone. It quickly becomes intractable when the contract size and complexity grow.

sCrypt is designed to facilitate writing smart contract running on chain.

- It is easy to learn. Syntactically, sCrypt is similar to Solidity, making it easier to be adopted by existing smart contract developers. However, the resemblance is only superficial, since sCrypt is compiled into Bitcoin Script by the underlying compiler, instead of EVM bytecode.
- It is statically typed. Type checking can help detect many errors at compile time.

Warning: sCrypt is still in experimental phase and is currently only intended for small amount usage.

CHAPTER 1

A Simple Smart Contract

Contract in sCrypt is conceptually similar to class in Object Oriented Programming. Each contract provides a template for a certain type of contracts (e.g., P2PHK or multisig), which can be instantiated into concrete runnable contract objects.

```
contract Test {
    int x;

    constructor(int x) {
        this.x = x;
    }

    public function equal(int y) {
        require(y == this.x);
    }
}
```

1.1 Constructor

Each contract has at most one constructor. It is where contract member variables are initialized. For example, it can initialize the public key hash of a P2PHK contract, or the hash of a secret in a hash puzzle contract.

1.1.1 Default Constructor

When no constructor is provided, the compiler will automatically generate a default constructor that initializes every member variable in the order they are declared. For example,

```
contract Test {
    int x1;
    bytes x2;
    bool x3;
```

(continues on next page)

(continued from previous page)

```
    public function equal(int y) {}
}
```

is functionally equivalent to

```
contract Test {
    int x1;
    bytes x2;
    bool x3;

    constructor(int x1, bytes x2, bool x3) {
        this.x1 = x1;
        this.x2 = x2;
        this.x3 = x3;
    }

    public function equal(int y) {}
}
```

1.2 require()

The `require()` function specifies terms/conditions of a contract. It consumes a boolean condition. If the condition is not met, the contract will abort execution and fail. Otherwise, the execution will resume.

1.3 Public Function

Each contract has at least one public function. It is denoted with the `public` keyword and does not return any value. The function body corresponds to locking script (commonly referred to as `scriptPubKey`) and its arguments unlocking script (aka, `scriptSig`). It is visible outside the contract and acts as the entry point into the contract (like `main` in `C` and `Java`).

A public function must end with a `require()` call. `require()` can also appear in other parts of a public function. A contract can only be fulfilled and succeed when its called public function runs to completion without violating any conditions in `require()`. In the above example, only `scriptSig` (i.e., `y`) equal to `this.x` can fulfill the contract.

1.3.1 Multiple Public Functions

A contract can have multiple public functions, representing different ways to fulfill a contract. Only one of the public functions can be called at a time. In this case, the last operator of `scriptSig` has to be the index of the public function called, starting from `1`. For example, if public function `larger` is called, `scriptSig` of `y 3` can fulfill the contract below, in which `3` is the public function index.

```
contract Test {
    int x;

    public function equal(int y) {
        require(y == this.x);
    }
```

(continues on next page)

(continued from previous page)

```
    public function smaller(int y) {
        require(y < this.x);
    }

    public function larger(int y) {
        require(y > this.x);
    }
}
```

CHAPTER 2

IDEs

There are currently two IDEs available.

2.1 Desktop IDE

A desktop version is available as a Visual Studio Code Extension. It can easily be found by searching `sCrypt` in the Extensions Marketplace. This IDE comes with advanced language features and is intended for professional development. A boilerplate project is a good base to kickstart your own sCrypt project.

2.2 Web IDE

A browser-based IDE can be found at scrypt.studio. It is for quick learning of sCrypt immediately without any installation and suitable only for small contracts. Currently, it does not support importing contracts.

CHAPTER 3

Syntax Specification

3.1 Formal Specification

$$
\begin{aligned}
program &::= [importDirective]^* \ [contract]^+ \\
importDirective &::= \text{import "ID";} \\
contract &::= \text{contract ID } \{ \ [var]^* \ [constructor] \ [function]^+ \ \} \\
var &::= formal; \\
formal &::= \text{TYPE ID} \\
constructor &::= \text{constructor}([formal[,\ formal]^*]) \ \{ \ [stmt]^* \ \} \\
function &::= \text{[public|static] function ID}([formal[,\ formal]^*]) \ \text{[returns (TYPE])} \ \{ \ [stmt]^* \ \text{[return } expr; \text{] } \} \\
stmt &::= \text{TYPE ID} = expr; \\
&\quad | \ \text{ID ID} = \text{new ID}(expr^*); \\
&\quad | \ \text{ID} = expr; \\
&\quad | \ \text{require}(expr); \\
&\quad | \ \text{exit}(expr); \\
&\quad | \ \text{if } (expr) \ stmt \ \text{[else } stmt] \\
&\quad | \ \text{loop } (intConst) \ stmt \\
&\quad | \ \{ \ [stmt]^* \ \} \\
&\quad | \ \text{CODESEPARATOR} \\
expr &::= \text{UnaryOp } expr \\
&\quad | \ expr \ \text{BinaryOp } expr \\
&\quad | \ \text{ID}(expr[,\ expr]^*) \\
&\quad | \ \text{ID.ID} \\
&\quad | \ \text{ID.ID}(expr[,\ expr]^*) \\
&\quad | \ \text{ID}[expr : expr] \\
&\quad | \ (expr) \\
&\quad | \ \text{ID} \\
&\quad | \ boolConst \\
&\quad | \ intConst \\
&\quad | \ bytesConst
\end{aligned}
$$

Most of the syntax is self explanatory. Syntax unique to sCrypt will be covered later.

Line comment starts with `//` and block comment comes between `/*` and `*/`.

3.2 Types

3.2.1 Basic Types

- **bool** - a boolean value `true` or `false`.
- **int** - a signed integer of arbitrary length, whose literals are in decimal or hexadecimal format.

```
int a1 = 42;
int a2 = -4242424242424242;
int a3 =␣
↪550662630222773436695787188951685343262506034537775941755001873603891167292 40;
↪
int a4 = 0xFF8C;
```

- **bytes** - a variable length array of bytes, whose literals are in quoted hexadecimal format prefixed by `b`.

```
bytes b1 = b'ffee1234';
bytes b2 = b
↪'414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffff00';
```

`bytes` can be converted to `int` using function `unpack`. Little-endian sign-magnitude representation is used, where the most significant bit indicates the sign (`0` for positive, `1` for negative). `int` can be converted to `bytes` with `pack`.

```
int a1 = unpack(b'36');     // 54 decimal
int a2 = unpack(b'b6');     // -54
int a3 = unpack(b'e803');   // 1000
int a4 = unpack(b'e883');   // -1000
bytes b = pack(a4);         // b'e883'
```

- **auto** keyword - The `auto` keyword specifies that the type of the variable, of basic type, declared will be automatically deducted from its initializer.

```
auto a1 = b'36';       // bytes a1 = b'36';
auto a2 = 1 + 5 * 3;   // int a2 = 1 + 5 * 3;
```

3.2.2 Subtypes of `bytes`

These subtypes are more specific versions of `bytes`, used to further improve type safety. To cast a supertype `bytes` to them, a function of the type name must be explicitly called.

- **PubKey** - a public key type.

```
PubKey pubKey = PubKey(b
↪'020011223344556677889 9aabbccddeeffffeeddccbbaa99887766554433221100');
```

- **Sig** - a signature type in DER format, including signature hash type, which is `SIGHASH_ALL | SIGHASH_FORKID` (`0x41`) in the below example.

```
Sig sig = Sig(b
↪'3045022100b71be3f1dc001e0a1ad65ed84e7a5a0bfe48325f2146ca1d677cf15e96e8b80302206d74605e823
↪');
```

(continues on next page)

(continued from previous page)

```
```

- **Ripemd160** - a RIPEMD-160 hash type.

```
Ripemd160 r = Ripemd160(b'00112233445566778899998877665544332211 00');
```

- **Sha1** - a SHA-1 hash type.

```
Sha1 s = Sha1(b'00112233445566778899998877665544332211 00');
```

- **Sha256** - a SHA-256 hash type.

```
Sha256 s = Sha256(b
↪'00112233445566778899aabbccddeeffffeeddccbbaa99887766554433221100');
```

- **SigHashType** - a sighash type.

```
SigHashType s = SigHashType(b'01');
SigHashType s = SigHash.ALL | SigHash.ANYONECANPAY;
```

- **OpCodeType** - a OpCode type.

```
OpCodeType s = OpCode.OP_DUP + OpCode.OP_ADD;
```

3.2.3 Subtypes of `int`

- **PrivKey** - a private key type.

```
PrivKey privKey =␣
↪PrivKey(0x00112233445566778899aabbccddeeffffeeddccbbaa99887766554433221100);
↪
```

3.3 `if` statement

`if` condition can be of type `int` and `bytes`, besides `bool`. They are implicitly converted to `bool` as in C and Javascript. An `int` expression is evaluated to `false` if and only if it is `0` (including negative `0`). A `bytes` expression is evaluated to `false` if and only if every of its byte is `b'00'` (including empty `bytes` `b''`).

```
int cond = 25; // true
int cond = 0;  // false
int cond = unpack(b'80') // false since it is negative 0
int cond = unpack(b'000080') // false since it is negative 0
if (cond) {} // equivalent to if (cond != 0) {}

bytes cond = b'00'; // false
bytes cond = b''; // false
bytes cond = b'80'; // true. Note b'80' is treated as false if converted to␣
↪int
bytes cond = b'10' & b'73'; // true since it evaluates to b'10'
if (cond) {}
```

3.4 exit()

`exit(bool status);` statement terminates contract execution. If `status` is `true`, contract succeeds; otherwise, it fails.

```
contract TestPositiveEqual {
    int x;

    constructor(int x) {
        this.x = x;
    }

    public function equal(int y) {
        if (y <= 0) {
          exit(true);
        }
        require(y == this.x);
    }
}
```

3.5 Code Separator

Three or more `*` in a line inserts an OP_CODESEPARATOR. It is used to exclude what comes before (and including itself) it from being part of the signature. Note there is no `;` at the end.

```
contract TestSeparator {
    public function equal(int y) {
        int a = 0;
        // separator 1
        ***
        int b = 2;
        // separator 2
        *****
        require(y > 0);
    }
}
```

3.6 Operators

Precedence	Operator	Associativity
1	`- ! ~`	right-associative
2	`* / %`	left-associative
3	`+ -`	left-associative
4	`<< >>`	left-associative
5	`< <= > >=`	left-associative
6	`== !=`	left-associative
7	`&`	left-associative
8	`^`	left-associative
9	`|`	left-associative
10	`&&`	left-associative
11	`||`	left-associative

3.7 Scoping

Scoping in sCrypt follows the prevailing scoping rules of C99 and Solidity. Outer scope variable is shadowed by the inner scope variable of the same name.

CHAPTER 4

Loop

```
loop (maxLoopCount)
    loopBody
```

Bitcoin script does not provide looping constructs natively for security reasons. sCrypt achieves looping by repeating the loop body `maxLoopCount` times. For example, the loop

```
loop (10) {
    x = x * 2;
}
```

is equivalently unrolled to

```
x = x * 2;
x = x * 2;
x = x * 2;
x = x * 2;
x = x * 2;
x = x * 2;
x = x * 2;
x = x * 2;
x = x * 2;
x = x * 2;
```

Because loop unrolling is done at compile time, the compiler must know `maxLoopCount`, which has to be a constant number.

If `maxLoopCount` is set too small, the contract may not work correctly. If `maxLoopCount` is set too large, the resulting script is bloated unnecessarily and costs more to execute. There are a number of ways to choose the right `maxLoopCount` judiciously. One way is to simulate the contract off chain and find the number of loops. Another way is to exploit the characteristics of the looping itself. For example, if a loop iterates over each bit of a `sha256` hash, `maxLoopCount` is `256`.

4.1 Access loop index

```
int i = 0;
loop (3) {
    // i becomes the loop index
    i = i + 1;
    x = x * 2;
}
```

4.2 Conditional loop

```
loop (3) {
    // place condition here
    if (x < 8) {
        x = x * 2;
    }
}
```

4.3 break

```
bool done = false;
loop (3) {
    if (!done) {
        x = x * 2;
        if (x >= 8) {
            done = true;
        }
    }
}
```

CHAPTER 5

Functions

5.1 User-Defined Functions

sCrypt enables developers to define their own functions as exemplified below:

```
function sum(int a, int b) returns (int) {
    return a + b;
}
```

They are only visible within the contract, similar to `private` functions in Solidity.

5.1.1 public function

A public function returns `true` if it runs to completion and `false` otherwise. It does not have `returns` and `return` parts, as they are included implicitly. In other words,

```
public function sum(int a) {
    require(a == 0);
}
```

is functionally equivalent to

```
public function sum(int a) returns (bool) {
    require(a == 0);
    return true;
}
```

5.1.2 static function and property

A static function/property can be referenced with contract name without an instantiated contract, similar to a static function/property in Javascript or C+.

```
contract Foo {
    int i;
    static int N = 0;

    static function incByN(int a) returns (int) {
            return a + Foo.N;
    }

    function double(int x) returns (int) {
        return Foo.incByN(x) + this.i;
    }
}

contract Bar {
    public function unlock(int y) {
        require(y == Foo.sum(1, 2));
    }
}
```

5.1.3 return

Due to the lack of native `return` semantics support in script, a function currently must end with a `return` statement and it is the only valid place for a `return` statement. This requirement may be relaxed in the future. This is usually not a problem since it can be circumvented as follows:

```
function abs(int a) returns (int) {
    if (a > 0) {
        return a;
    } else {
        return -a;
    }
}
```

can be rewritten as

```
function abs(int a) returns (int) {
    int ret = 0;

    if (a > 0) {
        ret = a;
    } else {
        ret = -a;
    }
    return ret;
}
```

5.1.4 Recursion

Recursion is disallowed. A function cannot call itself in its body.

Warning: Indirect recursion detection is currently not implemented. If function A calles function B, which in turn calls A, the compilation process will hang. Care must be taken to avoid doing so.

5.2 Library Functions

The following functions come with sCrypt and are available globally.

5.2.1 Math

- `int abs(int a)`
- `int min(int a, int b)`
- `int max(int a, int b)`
- `bool within(int x, int min, int max)`

5.2.2 Hashing

- `Ripemd160 ripemd160(bytes b)`
- `Sha1 sha1(bytes b)`
- `Sha256 sha256(bytes b)`
- `Ripemd160 hash160(bytes b)`

  ripemd160(sha256(b))
- `Sha256 hash256(bytes b)`

  sha256(sha256(b))

5.2.3 Signature Verification

- `bool checkSig(Sig sig, PubKey pk)`
- `bool checkMultiSig(Sig[] sigs, PubKey[] pks)`

5.2.4 bytes Operations

- `bytes b[start:end]`

  Returns subarray from index `start` (inclusive) to `end` (exclusive). `start` is `0` if omitted, `end` is `length(b)` if omitted.

```
bytes b = b'0011223344556677';
// b[3:6] == b'334455'
// b[:4] == b'00112233'
// b[5:] = b'556677'
```

- `b1 + b2`

  Returns the concatenation of bytes `b1` and bytes `b2`.

- `reverseBytes20(bytes b) reverseBytes32(bytes b)`

  Returns reversed bytes of `b`, which is of 20/32 bytes. They are often useful when converting a number between little-endian and big-endian.

```
// returns b'6cfeea2d7a1d51249f0624ee98151bfa259d095642e253d8e2dce1e79df33f79'
reverseBytes32(b'793ff39de7e1dce2d853e24256099d25fa1b1598ee24069f24511d7a2deafe6c')
```

- `bytes num2bin(int num, int size)`

  Converts a number `num` into a byte array of certain size `size`, including the sign bit. It fails if the number cannot be accommodated.

- `int length(bytes b)`

  Returns the length of `b`.

CHAPTER 6

Standard Contracts

6.1 Multiple Contracts

A single file can define multiple contracts. In this case, the last contract acts as the main contract and is what gets compiled. Other contracts are dependencies.

In the following example, a standard P2PKH contract is rewritten using two other contracts: a hash puzzle contract that checks the public key matches the public key hash, and a Pay-to-PubKey (P2PK) contract that verifies signature matches public key.

```
contract HashPuzzle {
    Ripemd160 hash;

    public function spend(bytes preimage) {
        require(hash160(preimage) == this.hash);
    }
}

contract Pay2PubKey {
    PubKey pubKey;

    public function spend(Sig sig) {
        require(checkSig(sig, this.pubKey));
    }
}

contract Pay2PubKeyHash {
    Ripemd160 pubKeyHash;

    public function spend(Sig sig, PubKey pubKey) {
        HashPuzzle hp = new HashPuzzle(this.pubKeyHash);
        require(hp.spend(pubKey));

        Pay2PubKey p2pk = new Pay2PubKey(pubKey);
```

(continues on next page)

(continued from previous page)

```
        require(p2pk.spend(sig));
    }
}
```

6.2 import

Alternatively, the contact above can be broken into three files. The `Pay2PubKeyHash` contact `imports` other two contracts as dependencies. This allows reusing contracts written by others and forms the basis of contract libraries.

A contract can be instantiated by `new`. A `public` function can be called from `require`, which takes boolean expression as input.

```
import "./hashPuzzle.scrypt";
import "./p2pk.scrypt";

contract Pay2PubKeyHash {
    Ripemd160 pubKeyHash;

    public function spend(Sig sig, PubKey pubKey) {
        HashPuzzle hp = new HashPuzzle(this.pubKeyHash);
        require(hp.spend(pubKey));

        Pay2PubKey p2pk = new Pay2PubKey(pubKey);
        require(p2pk.spend(sig));
    }
}
```

6.3 Standard Contracts

sCrypt comes with standard libraries that define many commonly used contracts. They are included by default and do not require explicit `import` to be used.

The following example shows usage of the standard contract `P2PKH` that corresponds to Pay-to-PubKey-Hash contract.

```
contract P2PKHStdDemo {
    Ripemd160 pubKeyHash;

    public function unlock(Sig sig, PubKey pubKey) {
        P2PKH p2pkh = new P2PKH(this.pubKeyHash);
        require(p2pkh.spend(sig, pubKey));
    }
}
```

6.3.1 Contract `OP_PUSH_TX`

One grave misconception regarding bitcoin script is that its access is only limited to the data provided in the locking script and corresponding unlocking script. Thus, its scope and capability are greatly underestimated.

sCrypt comes with a powerful contract called `Tx` that allows inspection of the **ENTIRE TRANSACTION** containing the contract itself, besides the locking script and unlocking script. It can be regarded as a pseudo opcode `OP_PUSH_TX`

that pushes the current transaction into the stack, which can be inspected at runtime. More precisely, it enables inspection of the preimage used in signature verification defined in BIP143. The format of the preimage is as follows:

1. nVersion of the transaction (4-byte little endian)
2. hashPrevouts (32-byte hash)
3. hashSequence (32-byte hash)
4. outpoint (32-byte hash + 4-byte little endian)
5. scriptCode of the input (serialized as scripts inside CTxOuts)
6. value of the output spent by this input (8-byte little endian)
7. nSequence of the input (4-byte little endian)
8. hashOutputs (32-byte hash)
9. nLocktime of the transaction (4-byte little endian)
10. sighash type of the signature (4-byte little endian)

As an example, contract `CheckLockTimeVerify` ensures coins are timelocked and cannot be spent before `matureTime` is reached, similar to OP_CLTV.

```
contract CheckLockTimeVerify {
    int matureTime;

    public function spend(bytes sighashPreimage) {
        // this ensures the preimage is for the current tx
        require(Tx.checkPreimage(txPreimage));

        // parse nLocktime
        int len = length(sighashPreimage);
        int nLocktime = this.fromLEUnsigned(sighashPreimage[len - 8 : len - 4]);

        require(nLocktime >= this.matureTime);
    }

    function fromLEUnsigned(bytes b) returns (int) {
        // append positive sign byte. This does not hurt even when sign bit is␣
↪already positive
        return unpack(b + b'00');
    }
}
```

More details can be found in this article. To customize ECDSA signing, such as choosing ephemeral key, there is a more general version called TxAdvanced.

6.3.2 Full List

Contract	Constructor parameters	Public function
P2PKH	Ripemd160 pubKeyHash	spend(Sig sig, PubKey pubKey)
P2PK	PubKey pubKey	spend(Sig sig)
HashPuzzleX[1]	Y[2] hash	spend(bytes preimage)
Tx	None	checkPreimage(bytes sighashPreimage)

[1] X is hashing function and can be Ripemd160/Sha1/Sha256/Hash160
[2] Y is hashing function return type and can be Ripemd160/Sha1/Sha256/Ripemd160

CHAPTER 7

Pay to Public Key Hash

Pay-to-PubKey-Hash (P2PKH) contract is used to send bitcoins to a bitcoin address. It is the most common contract on the Bitcoin network. Such contracts are unlocked by the public key and a signature created by the corresponding private key.

```
contract P2PKH {
    Ripemd160 pubKeyHash;

    public function unlock(Sig sig, PubKey pubKey) {
        require(hash160(pubKey) == this.pubKeyHash);
        require(checkSig(sig, pubKey));
    }
}
```

CHAPTER 8

R-Puzzle

In R-puzzle, an ephemeral key `k` is never revealed. Instead `r`, the x coordinate its its corresponding public key, is revealed and from `r` along with the signature, the knowledge of `k` can be proved using existing `checkSig`.

One crucial step in R-Puzzle is to extract `r` from DER encoded signature. The following is much easier than what is presented in the R-Puzzle talk.

```
contract RPuzzle {
    Sig s;  // s = b
↪'3045022100948c67a95f856ae875a48a2d104df9d232189897a811178a715617d4b090a7e90220616f6ced5ab219fe1b
↪'

    function getSigR(Sig sig) returns (bytes) {
        bytes lenBytes = sig[3:4];
        int len = unpack(lenBytes);
        bytes r = sig[4:4+len];
        return r;
    }

    // r = b'00948c67a95f856ae875a48a2d104df9d232189897a811178a715617d4b090a7e9'
    public function unlock(bytes r) {
        require(r == this.getSigR(this.s));
    }
}
```

CHAPTER 9

Ackermann Function

The Ackermann function is a classic example of a recursive function, notable especially because it is not a primitive recursive function. It grows very quickly in value, as does the size of its call tree. The Ackermann function is usually defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

nCrypt has devised a way to calculate the value of the Ackermann function using native scripts. But it is definitely non-trivial. Below we present a much simpler version.

```
contract Ackermann {
    int a;  // a = 2
    int b;  // b = 1

    function ackermann(int m, int n) returns (int) {
        bytes stk = num2bin(m, 1);

        // run this function off chain to get the loop count and set it here
        // e.g., (2, 1) requires 14 loops, (3, 5) 42438
        loop (14) {
            if (length(stk) > 0) {
                bytes top = stk[0:1];
                m = unpack(top);

                // pop
                stk = stk[1:length(stk)];

                if (m == 0) {
                    n = n + m + 1;
                } else if (n == 0) {
                    n = n + 1;
                    m = m - 1;
                    // push
```

(continues on next page)

(continued from previous page)

```
                    stk = num2bin(m, 1) + stk;
                } else {
                    stk = num2bin(m - 1, 1) + stk;
                    stk = num2bin(m, 1) + stk;
                    n = n - 1;
                }
            }
        }

        return n;
    }

    // y = 5
    public function unlock(int y) {
        require(y == this.ackermann(this.a, this.b));
    }
}
```

CHAPTER 10

Rabin Signature

Rabin signature is an alternative form of digital signature to ECDSA used in Bitcoin.

```
contract RabinSignature {
    public function verifySig(int sig, bytes msg, bytes padding, int n) {
        int h = this.fromLEUnsigned(this.hash(msg + padding));
        require((sig * sig) % n == h % n);
    }

    function hash(bytes x) returns (bytes) {
        // expand into 512 bit hash
        bytes hx = sha256(x);
        int idx = length(hx) / 2;
        return sha256(hx[:idx]) + sha256(hx[idx:]);
    }

    function fromLEUnsigned(bytes b) returns (int) {
        // append positive sign byte. This does not hurt even when sign bit is␣
↪already positive
        return unpack(b + b'00');
    }
}
```

CHAPTER 11

Multiparty Hash Puzzles

In a hash puzzle contract, the spender has to provide a preimage `x` that hashes to a predefined value `y` to unlock a UTXO. It can be extended to multiple parties so that multiple preimages have to be provided such that `y1 = H(x1)`, `y2 = H(x2)`, ..., `yN = H(xN)`[1]. Below shows an examples of three parties.

```
contract MultiPartyHashPuzzles {
    Sha256 hash1;   // hash1 = b
↪'136523B9FEA2B7321817B28E254A81A683D319D715CEE2360D051360A272DD4C'
    Sha256 hash2;   // hash2 = b
↪'E222E30CF5C982E5F6251D755B0B16F608ACE631EB3BA9BDAF624FF1651ABF98'
    Sha256 hash3;   // hash3 = b
↪'2A79F5D9F8B3770A59F91E0E9B4C379F7C7A32353AA6450065E43A8616EF5722'

    // preimage1: e.g., "bsv" -> b'627376'
    // preimage2: e.g., "sCrypt" -> b'734372797074'
    // preimage3: e.g., "IDE" -> b'494445'
    public function unlock(bytes preimage1, bytes preimage2, bytes preimage3) {
        require(sha256(preimage1) == this.hash1);
        require(sha256(preimage2) == this.hash2);
        require(sha256(preimage3) == this.hash3);
    }
}
```

The above solution is problematic when `N` is large since all `N` hashes have to be included in the locking script, bloating the transaction. Instead, we can combine all `y`'s into a single `y` such that `y = H(H(y1 || y2) || y3)`[2] as shown below.

```
contract MultiPartyHashPuzzlesCompact {
    // only 1 hash needs to go into the locking script, saving space
    Sha256 combinedHash; // combinedHash = b
↪'C9392767AB23CEFF09D207B9223C0C26F01A7F81F8C187A821A4266F8020064D'
```

(continues on next page)

[1] `H` is a hash function. An online hash calculator can be found here.

[2] `||` is concatenation.

(continued from previous page)

```
    // preimage1: e.g., "bsv" -> b'627376'
    // preimage2: e.g., "sCrypt" -> b'734372797074'
    // preimage3: e.g., "IDE" -> b'494445'
    public function unlock(bytes preimage1, bytes preimage2, bytes preimage3) {
        Sha256 hash1 = sha256(preimage1);
        Sha256 hash2 = sha256(preimage2);
        Sha256 hash3 = sha256(preimage3);
        Sha256 hash12 = sha256(hash1 + hash2);
        Sha256 hash123 = sha256(hash12 + hash3);

        require(hash123 == this.combinedHash);
    }
}
```

CHAPTER 12

Contact

Slack

Telegram

CHAPTER 13

Contributing

See CONTRIBUTING.md at our GitHub for more information on what we're looking for and how to get started.