sCrypt

发布 *0.0.1*

2021 年 09 月 11 日

简介

sCrypt (发音为 “ess crypt”) 是 Bitcoin SV 的智能合约高级语言。Bitcoin 通过基于栈的类 Forth 脚本语言来支持智能合约功能。但是，用原生脚本语言写智能合约既麻烦又容易出错。随着合约规模和复杂度的上升，很快就变得不可行了。

sCrypt 会让开发链上运行的智能合约变得轻松。

- It is easy to learn. Syntactically, sCrypt is similar to Javascript and Solidity, making it easier to be adopted by existing web and smart contract developers.
- 静态类型。类型检查可以在编译时就帮助检查出很多错误。

警告： sCrypt 仍处于试验阶段，仅用于小规模使用。

CHAPTER 1

一个简单的智能合约示例

sCrypt 中的合约（contract）在概念上类似于面向对象编程中的类（class）。每个 contract 都为特定类型的合约（如：P2PKH 或多重签名）提供了模板，可被实例化为可运行的合约对象。

```
contract Test {
    int x;

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

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

1.1 构造函数（constructor）

每个合约最多只能有一个构造函数。用于初始化合约的成员变量。例如，初始化 P2PKH 合约的公钥哈希，或者哈希谜题（hash puzzle）合约的 secret hash。

1.1.1 默认构造函数

当没有构造函数时，编译器会生成一个默认的构造函数，按照声明顺序初始化每一个成员变量。例如：

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

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

在功能上等同于

```
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()

`require()` 函数指定合约的限制条款/条件。它的参数是一个布尔条件表达式。如果条件表达式的值为假，合约将终止执行并失败。否则，将继续执行。

1.3 公有函数

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 and its arguments unlocking script. 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 `unlockingScript` (i.e., `y`) equal to `this.x` can fulfill the contract.

A public function can be regarded as a mathematical boolean function. `f` is the function body and `x` the function arguments. A contract call succeeds if and only if `f(x)` returns true.

$$f(x)$$

1.3.1 多个公有函数

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 `unlockingScript` has to be the index of the public function called, starting from `0`. For example, if public function `larger` is called, `unlockingScript` of `y 2` can fulfill the contract below, in which `2` is the public function index.

```
contract Test {
    int x;

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

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

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

CHAPTER 2

IDE

目前有两种 IDE 可供使用。

2.1 桌面 IDE

桌面版是一个 Visual Studio Code 插件 。在插件市场中搜索 `sCrypt` 就可以找到。这个 IDE 具有高级的功能，可用于专业开发。你可以以 样板项目 为基础开始构建你自己的 sCrypt 项目。

2.2 网页 IDE

scrypt.studio 是一个基于浏览器的 IDE。无需任何安装就可以立即快速学习 sCrypt。此 IDE 仅适用于小型合约的开发。目前，它还不支持导入合约功能。

CHAPTER 3

语法规范

3.1 形式规范

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

大部分语法含义都是显而易见的。sCrypt 特有的语法会在后面介绍。

行注释用 `//` 开头，在 `/*` 和 `*/` 之间的是块注释。

3.2 类型

3.2.1 基本类型

- **bool** - 布尔类型，值为 `true` 或 `false` 。
- **int** - 有符号的任意长度整数类型，字面量（literals）有十进制和十六进制两种格式。

```
int a1 = 42;
int a2 = -4242424242424242;
int a3 =␣
↪55066263022277343669578718895168534326250603453777594175500187360389116729240;
↪
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 b3 = b'1122' + b'eeff'; // b3 is b'1122eeff'
```

3.2.2 数组类型

An array is a fixed-size list of values of the same basic type.

- **Array Literals** - a comma-separated list of expressions, enclosed in square brackets. Array size must be an integer constant greater than zero.

```
bool[3] b = [false, false && true || false, true || (1 > 2)];
int[3] c = [72, -4 - 1 - 40, 833 * (99 + 9901) + 8888];
bytes[3] a = [b'ffee', b'11', b'22'];
int[2][3] d = [[11, 12, 13], [21, 22, 23]];
// array demension can be omitted when declared
int[] e = [1, 4, 2];  // e is of type int[3]
int[][] f = [[11, 12, 13], [21, 22, 23]]; // f is of type int[2][3]
```

- **Initialize/set an array to the same value** - Function `T[size] repeat(T e, static const int size)` returns an array with all `size` elements set to `e`, where T can be any type. Note `size` must be a *compile time constant*.

```
// a == [0, 0, 0]
int[3] a = repeat(0, 3);
// arr2D == [[0, 0, 0], [0, 0, 0]]
int[2][3] arr2D = repeat(0, 2);
int[4] flags = [false, true, false, true]
// set all flags to be false
flags = repeat(false, 4);
```

- **Index Operator** - index starting from 0. Out of bound access fails contract execution immediately.

```
int[3] a = [1, 4, 2];
int[2][3] arr2D = [[11, 12, 13], [21, 22, 23]];
int d = a[2];
a[1] = -4;
int idx = 2;
// variable index is allowed when reading an array
d = a[idx];
d = arr2D[idx][1];
// variable index is disallowed when writing into an array
a[idx] = 2;
// only a compile-time constant (CTC) can be used as an index when writing
a[2] = 2;
a[N] = 3; // N is a CTC
// assign to an array variable
a = arr2D[1];
// b is a new copy and the same as a
int[3] b = a;
// two arrays are equal if and only if they are of the same size and all␣
↪elements are equal
require(a == b);
```

3.2.3 Struct Types

A struct (or structure) is a collection of variables (can be of different basic types) under a single name.

- **Define Struct**

```
struct Point {
  int x;
  int y;
}

struct Line {
  // nested struct
  Point start;
  Point end;
}
```

- **Use Struct**

```
Point p = {10, -10};
int x = p.x;
p.y = 20;
// Define a variable q of type Point, and set members to the same values as␣
↪those of p
Point q = p;
require(p == q); // true
// nested
Line l = {p, q};
l.start.x = l.end.y + 1;
```

3.2.4 类型接口

auto 关键字表示变量的类型由变量的初始值自动推导出来。

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

3.2.5 Type Aliases

Type aliases create a new name for a type. It does not actually create a new type, it merely creates a new name to refer to that type.

```
type Age = int;
type Coordinate = int[2];
```

3.3 领域子类型

如下是一些在比特币语境中特定的子类型，用于进一步提高类型安全性。

3.3.1 bytes 的子类型

要把 bytes 类型强制转换成某个子类型，必须显式调用与该子类型同名的函数。

- **PubKey** - 公钥类型。

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

- **Sig** - DER 格式的签名类型。包含 签名哈希类型 ，如下例子中的签名哈希类型是 SIGHASH_ALL | SIGHASH_FORKID (0x41)。

```
Sig sig = Sig(b
↪'3045022100b71be3f1dc001e0a1ad65ed84e7a5a0bfe48325f2146ca1d677cf15e96e8b80302206d74605e8234ea
↪');
```

- **Ripemd160** - RIPEMD-160 哈希类型。

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

- **Sha1** - SHA-1 哈希类型。

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

- **Sha256** - SHA-256 哈希类型。

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

- **SigHashType** - 签名哈希类型。

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

- **SigHashPreimage** - 签名哈希原像类型（a sighash preimage type）。

```
SigHashPreimage s = SigHashPreimage(b'0100000028bcef7e73248aa273db19d73');
```

- **OpCodeType** - 操作码类型

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

3.3.2 int 的子类型

- **PrivKey** - 私钥类型

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

3.4 const Variables

Variables declared const cannot be changed once initialized.

```
contract Test {
    const int x;

    constructor(int x) {
        this.x = x; // good, since this is initialization
    }

    public function equal(const int y) {
        y = 1; // <-- error

        const int a = 36;
        a = 11; // <-- error

        require(y == this.x);
    }
}
```

3.5 if 语句

除了 bool 类型，if 条件还可以是 int 和 bytes 。这些类型会被隐式转换为 bool 类型，与 C 和 Javascript 语言中的处理方式一样。当且仅当 int 为 0 （包括负 0 ）时，为 false 。当且仅当 bytes 的每个字节都是 b'00' （包括空 bytes b'' ）时，为 false 。

```
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) {}
```

3.6 exit()

`exit(bool status);` 语句用于结束合约的执行。如果 `status` 参数是 `true`，合约执行成功；否则执行失败。

```
contract TestPositiveEqual {
    int x;

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

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

3.7 代码分隔符

一行中三个或更多 * 表示插入一个 OP_CODESEPARATOR 。该操作符之前的内容（包括操作符本身）不参与签名计算。注意，该语句行末没有 ；。

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

3.8 Access Modifiers

There are three types of access modifiers available to help restrict the scope of properties and functions of a contract:

- Default: no keyword required
- Private
- Public: only applies to functions

Only public functions can be called externally by Bitcoin transactions.

	default	private	public
Same contract	Yes	Yes	Yes
Other contract	Yes	No	Yes
Externally	No	No	Yes

3.9 操作符

优先级	操作符	关联性
1	`- ! ~`	右
2	`* / %`	左
3	`+ -`	左
4	`<< >>`	左
5	`< <= > >=`	左
6	`== !=`	左
7	`&`	左
8	`^`	左
9	`|`	左
10	`&&`	左
11	`||`	左
12	`? :`	右

Operator `&&`, `||`, and `? :` use short-circuit evaluation.

3.10 作用域

sCrypt 的作用域遵循 C99 和 Solidity 的现行作用域规则。外部作用域的变量会被内部作用域的同名变量覆盖。

CHAPTER 4

循环

```
loop (maxLoopCount) [: loopIndex]
    loopBody
```

因为安全原因，比特币脚本没有提供循环结构。sCrypt 通过重复循环体 `maxLoopCount` 次来实现循环。例如，下面的循环

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

相当于展开成如下形式

```
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`. That is, it must be a *compile time constant*.

如果 `maxLoopCount` 设置得太小，合约可能无法正常运行。如果 `maxLoopCount` 设置得太大，那么生成的脚本大小会不必要地膨胀，增加执行成本。有一些方法可以把 `maxLoopCount` 的值设置得更合理。其中一个方法是在链下模拟运行合约以找到合理的循环次数。另一种方法是利用循环自身的特征。比如，如果一个循环需要遍历 `sha256` 哈希值的每个比特，那么 `maxLoopCount` 就应该设置为 `256` 。

4.1 Induction variable

Induction variable can be defined when loop index is needed.

```
// int[3][4] matrix;
// i & j are induction variables
loop (3) : i {
    // i is the outer loop index
    loop (4) : j {
        // j is the inner loop index
        matrix[i][j] = i + j;
    }
}
```

4.2 有条件的循环

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

4.3 跳出循环

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

(下页继续)

(续上页)

```
            done = true;
        }
    }
}
```

CHAPTER 5

函数

5.1 用户自定义函数

sCrypt 允许开发者定义自己的函数，如下所示：

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

这些函数只在合约内可见，类似 Solidity 里的 `private` 函数。

5.1.1 公有函数

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

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

功能上等同于

```
public function isZero(int a): bool {
    require(a == 0);
```

(下页继续)

(续上页)

```
    return true;
}
```

5.1.2 静态函数和静态属性

A static property/function can be referenced with contract name as prefix without an instantiated contract, similar to a static function/property in Javascript or C++. A static property/function can also be referenced without the contract prefix, but only in the contract it is defined in.

```
library Foo {
    static int N = 0;

    static function incByN(int a): int {
        // N is used with and without Foo prefix
        return a + Foo.N + N;
    }

    static function double(int x): int {
        // incByN() is called with prefix and without
        return Foo.incByN(x) + incByN(x);
    }
}

contract Bar {
    public function unlock(int y) {
        require(y == Foo.double(2));
        require(y == Foo.N);
        // N cannot be referenced without Foo prefix
        // require(y == N);
    }
}
```

5.1.3 return

由于比特币脚本缺乏对 return 语义的支持，所以函数必须以 return 语句结尾，并且 return 语句只能放在函数末尾，不能放在其他位置。将来可能会放松这个限制。一般来说这不是问题，可以用如下方式避免在其他位置返回：

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

可以改写成

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

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

5.1.4 递归

Recursion is disallowed. A function cannot call itself in its body, either directly or indirectly.

5.2 库函数

sCrypt 实现了如下库函数，在全局可见。

5.2.1 数学

- `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 哈希

- `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 签名校验

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

5.2.4 字节数组操作

- 与 `int` 互相转换

用函数 `unpack` 可以把 `bytes` 类型转换为 `int` 类型。采用小端格式的 符号-值表示法 ，最高位比特表示符号（`0` 为正，`1` 为负）。用函数 `pack` 可以把 `int` 类型转换为 `bytes` 类型。

```
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'
```

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

  把数字 `num` 转换为字节数为 `size` 的字节数组，包括符号比特。如果字节数组无法容纳被转换的数字，则会转换失败。

- `len()` Returns the length.

  ```
  int a = len(b'ffee11'); // a == 3
  ```

- **Slicing Opeartor** - `b[start:end]` returns subarray of `b` from index `start` (inclusive) to `end` (exclusive). `start` is `0` if omitted, `end` is length of array if omitted.

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

- **Concatenation**

```
bytes b = b'00112233' + b'334455'  // b == b'00112233334455'
```

- `reverseBytes(bytes b, static const int size)`

  Returns reversed bytes of `b`, which is of `size` bytes. Note `size` must be a *compile time constant*. It is often useful when converting a number between little-endian and big-endian.

```
// returns b
↪'6cfeea2d7a1d51249f0624ee98151bfa259d095642e253d8e2dce1e79df33f79'
reverseBytes(b
↪'793ff39de7e1dce2d853e24256099d25fa1b1598ee24069f24511d7a2deafe6c', 32)
```

CHAPTER 6

标准合约

6.1 多个合约

一个文件中可以定义多个合约。在这种情况下，最后一个合约是主合约，这个合约会被编译。其他合约都被主合约依赖。

在下面这个例子中，标准的 P2PKH 合约被改写为两个其他合约：一个用来检查公钥和公钥哈希是否匹配的哈希谜题（hash puzzle）合约，还有一个检查签名和公钥是否匹配的 Pay-to-PubKey（P2PK）合约。

```
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));
    }
}
```

(下页继续)

(续上页)

```
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));
    }
}
```

6.2 导入（import）

或者，可以将上述合约分到三个文件中。Pay2PubKeyHash 合约 import 其他两个合约作为依赖。这就可以重用其他人写的合约，成为构建合约库的基础。

可以通过 new 来实例化一个合约。require 函数的参数是条件表达式，在条件表达式里可以调用合约的 public 函数。

```
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 Library

A library is the same with a contract, except it does not contain any public function. It is only intended to be imported by a contract or other libraries. It thus cannot be independently deployed and called. It is frequently used to group related constants and static functions.

```
library Util {
    // number of bytes to denote some numeric value
    static const int DataLen = 1;
    // number of bytes to denote length serialized state, including varint prefix (1␣
↪byte) + length (2 bytes), change length to 4 when you need PushData4
    static const int StateLen = 3;

    // convert signed integer `n` to unsigned integer of `l` bytes, in little endian
    static function toLEUnsigned(int n, int l): bytes {
        // one extra byte to accommodate possible negative sign byte
        bytes m = num2bin(n, l + 1);
        // remove sign byte
        return m[0 : len(m) - 1];
    }
}
```

6.4 标准合约

sCrypt 自带标准库，里面定义了许多常用的合约。标准库是默认就导入的，不需要写 `import` 语句。

如下例子展示了对标准合约 `P2PKH` 的使用。

```
contract P2PKHStdDemo {
    Ripemd160 pubKeyHash;

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

6.4.1 OP_PUSH_TX 合约

对比特币脚本的一个严重误解是，脚本只能访问锁定脚本以及对应的解锁脚本中提供的数据。因此，脚本的范围和能力被大大低估了。

sCrypt 提供了一个强大的合约叫做 `Tx`，它允许合约访问合约所在的 **整个交易**，包括锁定脚本和解锁脚本。我们把这种方法当成一个伪操作码 `OP_PUSH_TX`，它可以把当前交易压到栈里，这样就可以在运行时访问了。更准确地说，可以访问的是在签名校验时用到的原像（preimage），在 BIP143 中有原像的详细定义。原像的数据格式如下：

1. 交易的版本号（nVersion of the transaction）（4 字节小端）
2. 输入的输出点哈希（hashPrevouts）（32 字节哈希值）
3. 序列号哈希（hashSequence）（32 字节哈希值）
4. 此输入的输出点（outpoint）（32 字节哈希值 + 4 字节小端）
5. 此输入的锁定脚本（scriptCode of the input）（在 CTxOuts 中序列化为脚本）
6. 此输入对应的输出中包含的聪数（value of the output spent by this input）（8 字节小端）
7. 此输入的序列号（nSequence of the input）（4 字节小端）
8. 输出的哈希（hashOutputs）（32 字节哈希值）
9. 交易的 nLocktime（nLocktime of the transaction）（4 字节小端）
10. 交易的签名哈希类型（sighash type of the signature）（4 字节小端）

举个例子，合约 `CheckLockTimeVerify` 确保合约中的币是时间锁定的，在 `matureTime` 这个时间点之前不能被花掉。其功能类似 OP_CLTV。

```
contract CheckLockTimeVerify {
    int matureTime;

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

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

        require(nLocktime >= this.matureTime);
    }

    function fromLEUnsigned(bytes b): int {
        // append positive sign byte. This does not hurt even when sign bit is already␣
↪positive
```

(下页继续)

(续上页)

```
        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 Tx.checkPreimageAdvanced().

6.4.2 完整列表

合约	构造参数	公有函数
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 是哈希函数，可以是 Ripemd160/Sha1/Sha256/Hash160

[2] Y 是哈希函数的返回值类型，可以是 Ripemd160/Sha1/Sha256/Ripemd160

CHAPTER 7

Compile Time Constant

A compile time constant (CTC) is a value that can be computed at compile time. There are four types of compile time constants.

- literals
- *induction variables*
- static constant properties of a contract, initialized with a literal
- static constant function parameters

There are several cases where only compile time constants are allowed.

- loop bound
- array size
- write to an array element using *index operator*
- function parameters that are declared as `static const`[1] such as `size` in `reverseBytes(bytes b, static const int size)` and `repeat(T e, static const int size)`

```
contract CTC {
    static const int N = 4;
    static const int LOOPCOUNT = 30;
```

(下页继续)

[1] Note: the order is important: `const static` is not allowed when declaring a function parameter, but allowed when declaring a property.

(续上页)

```
    // A is not a CTC because the right hand size is an expression, not a literal
    static const int A = 2 + 1;
    // B is not a CTC because it is not static
    const int B;

    // FC is a CTC declared in function parameters
    // it can be used within this function, including parameters after it & return type
    function incArr(static const int FC, int[FC] x) : int[FC] {
        loop(FC): i {
            x[i] += i; // induction variable CTC
        }
        return x;
    }

    public function unlock(int y) {
        int[N] arr0 = [1, 2, 3, 4];
        // use `N` to initialize CTC parameter `FC` of function `incArr`
        int[N] arr1 = this.incArr(N, repeat(1, N));
        loop(N) : i {
            require(arr0[i] == arr1[i]);
        }

        int z = 0;
        loop (LOOPCOUNT)
        {
            if (z<y) z += 4;
        }
        require(y == 1);
    }
}
```

CHAPTER 8

支付到公钥哈希（Pay to Public Key Hash）

支付到公钥哈希（Pay-to-PubKey-Hash）（P2PKH）合约被用来把比特币发送到比特币地址。是比特币网络中最常见的合约。这种合约可以用两个参数来解锁：公钥和对应私钥创建的签名。

```
contract P2PKH {
    Ripemd160 pubKeyHash;

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

CHAPTER 9

R-Puzzle

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

```
contract RPuzzle {
    Ripemd160 rhash;

    constructor(Ripemd160 rhash) {
        this.rhash = rhash;
    }

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

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

(下页继续)

(续上页)

```
}
```

CHAPTER 10

阿克曼函数（Ackermann Function）

阿克曼函数是递归函数的经典示例，特别值得注意的是它并不是一个原始递归函数。它的值增长得非常快，调用树也增长得非常快。阿克曼函数通常定义如下：

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

nCrypt 设计了一种使用 原生脚本 计算阿克曼函数值的方法，该方法是非常复杂的。下面我们给出了一个简单得多的版本。

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

    function ackermann(int m, int n): 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 (len(stk) > 0) {
                bytes top = stk[0:1];
                m = unpack(top);
```

（下页继续）

(续上页)

```
                // pop
                stk = stk[1:len(stk)];

                if (m == 0) {
                    n = n + m + 1;
                } else if (n == 0) {
                    n = n + 1;
                    m = m - 1;
                    // push
                    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 11

拉宾签名（Rabin Signature）

拉宾签名 是比特币中使用的 ECDSA 数字签名的一种替代形式。

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

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

CHAPTER 12

多方哈希谜题（Multiparty Hash Puzzles）

在哈希谜题（hash puzzle）合约中，花费方必须提供一个原像（preimage）`x`，让 `x` 的哈希值等于预先定义好的值 `y`，才可以解锁 UTXO。这种合约可以扩展到多方，提供多个原像来满足 `y1 = H(x1)`，`y2 = H(x2)`，…，`yN = H(xN)`[1]。下面展示了一个三方的例子。

```
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);
    }
}
```

[1] `H` 哈希函数。这里 有个在线的哈希计算器。

上面的方案在当 N 比较大时会有问题，因为在锁定脚本里要把 N 个哈希值都包含进去，这会让交易变大。有个替代方案，我们可以把所有的 y 值放到一个里面：y = H(H(y1 || y2) || y3)[2]，如下所示。

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

    // 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);
    }
}
```

[2] || 表示连接。

CHAPTER 13

内联汇编

脚本是一种低级语言，可以看作是 比特币虚拟机 的汇编语言。通常，开发人员不用直接使用它，而是用像 sCrypt 这样的高级语言。但有些情况下用脚本语言更合适。例如，直接写出更优化的脚本，比用 sCrypt 生成的更高效。或者脚本是用外部工具（如 MiniForth ）生成的，想集成到 sCrypt 中。

用汇编表示法可以直接把脚本嵌入到 sCrypt 源代码中。可以用脚本编写 sCrypt 函数，并像正常的 sCrypt 函数一样被调用。

13.1 调用规范

For a function to be written in Script, its entire body must be enclosed by `asm` mode. Function parameters are on top of the stack, in reverse order as declared. For example, for a function with signature `function foo(int p0, bytes p1, bool p2): int`, `p2` will be top of the stack, `p1` second from top, and `p0` third from top when entering `asm` mode. When exiting `asm` mode, the return value is on top of the stack. All other items in the stack before the call must remain intact.

Three assembly functions are shown below.

```
// compute length of "b" in bytes
function len(bytes b): int {
    asm {
        op_size
    }
}
```

(下页继续)

(续上页)

```
// this is also fine since the return is on top and none in the old stack is changed
function len(bytes b): int {
    asm {
        op_size
        op_nip
    }
}

function lenFail(bytes b): int {
    // this is wrong since the original top of stack is dropped
    asm {
        op_drop
        op_size
    }
}
```

13.2 汇编变量

在 asm 模式下，可以用 $ 前缀来使用变量。其他脚本都是逐字复制到最终脚本输出中的，而汇编变量不是。变量以它的作用域为前缀，这样可以避免重名，由变量所在的合约和函数进行唯一标识。例如，在合约 contractFoo 中的函数 func 中有个变量 $var ，在最终脚本输出中会被转换成 $contractFoo.func.var。

下面展示一个例子。

```
function p2pkh(Sig sig, PubKey pubKey): bool {
    asm {
        op_dup
        op_hash160
        $pkh
        op_equalverify
        op_checksig
    }
}
```

13.3 注意

内联汇编绕过了 sCrypt 的许多功能，例如类型检查等。所以使用这个高级功能时要格外小心。此外，该功能在与外部工具的兼容性上是不区分大小写的。

CHAPTER 14

联系方式

scrypt.io

Slack

Telegram

CHAPTER 15

贡献

有关我们需要什么以及如何上手的更多信息，请参见我们 GitHub 上的 CONTRIBUTING.md 。