sCrypt

发布 *0.0.1*

2022 年 12 月 16 日

介绍

sCrypt（发音为“ess crypt”）是一种用于比特币 SV 的高级智能合约语言。比特币通过其类似 Forth 堆栈的脚本语言支持智能合约。然而，用原生脚本编写智能合约既麻烦又容易出错。当合约规模和复杂性增加时，它很快就会变得棘手。

sCrypt 旨在让开发链上运行的智能合约变得轻松。

- 这很容易学习。在语法上，sCrypt 类似于 Javascript 和 Solidity，使其更容易被现有的 Web 和智能合约开发人员更容易接受。
- 它是静态类型的。类型检查可以帮助在编译时检测许多错误。

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 中的合约在概念上类似于面向对象编程中的类。每个合约都为某种类型的合约（例如 P2PHK 或 multisig）提供了一个模板，可以将其实例化为具体的可运行合约对象。

```
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）

每个合约最多有一个构造函数。用于初始化合约的成员变量。例如，它可以初始化 P2PKH 合约的公钥哈希，或哈希谜题 hash puzzle 合约的哈希秘密。

1.1.1 默认构造函数

当没有提供构造函数时，编译器将自动生成一个默认构造函数，它按照声明的顺序初始化每个成员变量。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() 语句

`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 公共函数

每个合约至少有一个公共函数。它用 `public` 关键字表示并且不返回任何值。函数体对应锁定脚本，函数参数对应解锁脚本。公共函数在合约之外是可见的，并作为合约的入口点（就像 C 和 Java 中的 main 一样）。

公有函数的最后一个语句必须是 `require()` 。`require()` 也可以出现在公有函数的其他地方。只有在被调用的公有函数执行完毕，并且所有 `require()` 中的条件全部被满足时，合约才算执行成功。在上面的例子中，

只有 unlockingScript （即 y ）等于 this.x 时合约才能执行成功。

公共函数可以被视为布尔数学函数。f 是函数体，x 是函数参数。当且仅当 f(x) 返回 true 时，合约调用才会成功。

$$f(x)$$

1.3.1 多个公共函数

一个合约可以有多个公共函数，代表解锁合约的不同方式。一次只能调用一个公共函数。

```
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

集成开发环境

目前有三种 IDE 可用。

2.1 桌面 IDE

桌面版本是一个 Visual Studio Code 插件 Visual Studio Code Extension。通过在扩展市场中搜索 `sCrypt` 可以很容易地找到它。此 IDE 具有高级语言功能，适用于专业开发。你可以以 样板项目 为基础开始构建你自己的 sCrypt 项目。

这是关于如何使用此 IDE 的 文档 。

2.2 Studio

scrypt.studio 是一个基于浏览器的 IDE。无需任何安装即可快速学习 sCrypt，仅适用于小型合约。目前不支持导入合约功能。

2.3 Playground

sCrypt Playground 结合了桌面 IDE 的全部功能和 sCrypt Studio 的便利性。它完全在您的浏览器中促进合约开发的整个生命周期，包括开发、部署和管理 sCrypt 智能合约。它提供与桌面 IDE 完全相同的高级功能，因为它只是在浏览器中运行的 VSCode。它不需要任何设置，也可以作为学习和教授 sCrypt 的游乐场。

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。

行注释以 // 开头，块注释位于 /* 和 */ 之间。

3.2 类型

3.2.1 基本类型

- **bool** - 布尔值 true 或 false。
- **int** - 任意长度的有符号整数，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** - 一个可变长度的字节数组，其字面量是带引号的十六进制格式，前缀为 b，或双引号 UTF8 字符串。

```
bytes b1 = b'ffee1234';
bytes b2 = b
↪'414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffff00';
bytes b3 = b'1122' + b'eeff'; // b3 is b'1122eeff'
bytes str = "hello world"; // utf8 string
```

3.2.2 数组类型

数组是长度固定的，具有相同基本类型的值列表。

- **数组常量** - 以逗号分隔的表达式列表，括在方括号中。数组大小必须是大于零的整数常量。

```
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]
```

- **将数组初始化/设置为相同值** - 函数 `T[size] repeat(T e, static const int size)` 返回一个数组，其中所有 `size` 元素都设置为 `e` ，其中 `T` 可以是任何类型。注意 `size` 必须是编译时常量。

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

- **索引运算符** - 索引从 0 开始。越界访问立即使合约执行失败。

```
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;
// read
d = a[idx];
d = arr2D[idx][1];
// write
a[idx] = 2;
// 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 结构体

结构体是单个名称下的变量的集合。变量可以是不同的基本类型、数组或结构体

- **定义结构体**

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

(下页继续)

(续上页)

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

- **使用结构体**

```
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 类型推断

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 类型别名

类型别名为类型创建一个新名称。它实际上并没有创建一个新类型，它只是创建一个新名称来引用该类型。请注意，声明的右侧不能是未指定的泛型结构类型。

```
type Age = int;
type Coordinate = int[2];
type A = ST<int>; // this is fine.
type B = ST; // this is not allowd.
```

3.2.6 泛型/泛型类型

泛型类型是参数化类型。它允许库/结构体处理多种类型而不是单一类型。用户可以创建这些库/结构体并使用他们自己的具体类型。

- **声明泛型类型**

泛型类型可以在库级别声明并在库的范围内使用，也可以在结构级别声明并在内部使用。

```
// declare a library with two generic type variables: K & V
library HashedMap<K, V> {

  // use them as function parameters' type
  function set(K k, V v, int idx) {
    ...
  }

}

// declare a struct with two generic type variables: T & P
struct ST<T, P> {
  T x;
  P y;
}
```

- **实例化泛型类型**

```
// initialize a library with generics
HashedMap<bytes, int> map = new HashedMap();

// initialize a struct with generics
ST<int, bytes> st = {1, b'02'};
```

3.3 子类型

有几个特定于比特币上下文的子类型，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** - 一种公钥类型。

```
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
↪'3045022100b71be3f1dc001e0a1ad65ed84e7a5a0bfe48325f2146ca1d677cf15e96e8b80302206d74605e8234e
↪');
```

- **Ripemd160** - RIPEMD-160 哈希类型。

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

- **PubKeyHash** - *Ripemd160* 类型的别名。通常代表一个比特币地址。

```
PubKeyHash aliceAddress = PubKeyHash(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** - sighash 原像类型。

```
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 变量

声明为 const 的变量一旦初始化就不能更改。

```
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

if 条件可以是 int 和 bytes 类型，除了 bool 。它们像在 C 和 Javascript 中一样被隐式转换为 bool。int 表达式被评估为 false 当且仅当它为 0 （包括负数 0）。bytes 表达式被评估为 false 当且仅当它的每个字节都是 b'00' （包括空的 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
```

(下页继续)

(续上页)

```
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() 语句

exit(bool status); 语句终止合约执行。如果 status 为 true ，则合约执行成功；否则合约执行失败。

```
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 Code Separator 代码分隔符

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

```
contract P2PKH_OCS {
    Ripemd160 pubKeyHash;

    public function unlock(Sig sig, PubKey pubKey) {
        // code separator 1
        ***
        require(hash160(pubKey) == this.pubKeyHash);
        // code separator 2
```

(下页继续)

(续上页)

```
        *****
        require(checkSig(sig, pubKey));
    }
}
```

3.8 访问修饰符

可以使用三种类型的访问修饰符来帮助限制合约的属性和函数的范围：

- 默认：不需要关键字
- 私有的
- 公共：仅适用于函数

比特币交易只能从外部调用公共函数。

	default	private	public
合约内	Yes	Yes	Yes
其他合约	Yes	No	Yes
外部	No	No	Yes

3.9 运算符

优先级	运算符	关联性	注意
1	`() ++ -- .`	左	
2	`[]`	左	
3	`++ -- - ! ~`	右	
4	`* / %`	左	
5	`+ -`	左	
6	`<< >>`	左	要移位的位数必须为非负数，否则立即失败
7	`< <= > >=`	左	
8	`== !=`	左	
9	`&`	左	在两个长度不同的整数的情况下，较短的首先使用 *num2bin* 扩展为与较长的相同的长度
10	`^`	左	与 `&` 相同
11	`|`	左	与 `&` 相同
12	`&&`	左	
13	`||`	左	
14	`? :`	右	
15	`+= -= *= /= %= &= |= ^= <<= >>=`	右	

注解:

- 运算符 `&&`、`||` 和 `? :` 使用 短路评估.
- 在对整数执行按位运算后，例如运算符 `&`、`|`、`^` 和 `~`，编译器使用 `OP_BIN2NUM` 压缩运算结果。
- 不管是正数还是负数，都是以 原码 格式存储的，区别于计算机使用的 补码 格式。如果参与运算的操作数都是正数，则运算结果与补码的按位运算符一致。(除了 `~`)。否则，运行结果可能会不一致。

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
```

出于安全原因，比特币脚本本身不提供循环结构。sCrypt 通过重复循环体 `maxLoopCount` 次来实现循环。例如，循环

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

等效地展开为

```
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;
```

因为 循环展开 是在编译时完成的，编译器必须知道 `maxLoopCount`。也就是说，它必须是一个编译时常量。

如果 maxLoopCount 设置得太小，合约可能无法正常工作。如果 maxLoopCount 设置得太大，生成的脚本会不必要地膨胀，并且执行成本更高。有多种方法可以明智地选择正确的 maxLoopCount。一种方法是模拟链下合约并找到循环次数。另一种方法是利用循环本身的特性。例如，如果循环遍历 "sha256" 散列的每一位，maxLoopCount 就是 256。

4.1 归纳变量 (Induction Variable)

归纳变量 可以在需要循环索引时定义。

```
// 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 条件循环

```
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) {
            done = true;
        }
```

(下页继续)

(续上页)

```
    }
}
```

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，类似于 Solidity 中的 `private` 函数。

5.1.1 公共函数

如果公共函数运行完成，则返回 `true`，否则返回 `false。它没有返回类型和 返回部分，返回语句是是隐式包含的。换句话说，

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

在功能上等同于

```
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

可以通过合约名来直接引用静态函数/方法，而不必创建合约实例。类似于 Javascript 或 C++ 中的静态函数/属性。在定义它的合约中，静态属性/函数也可以在没有合约前缀的情况下被引用。

```
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 语句

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) {
```

(下页继续)

(续上页)

```
        return a;
    } else {
        return -a;
    }
}
```

可以改写为

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

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

5.1.4 递归

不允许递归。函数不能直接或间接地在其主体中调用自身。

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))

- `Sha256 flattenSha256(T a)`

  为任何类型的给定参数 a 返回 Sha256。如果 `T` 是基本类型，如 `bool / int / bytes` ，则返回与 `sha256(a)` 相同。如果 `T` 是复合类型（即数组和结构体），它将 a 的每个展平字段的所有 sha256 值连接起来形成一个联合字节，然后对其调用 `sha256` 以获得最终的结果。

5.2.3 签名验证

- `bool checkSig(Sig sig, PubKey pk)`

  如果签名与公钥匹配，则返回 true。如果签名是空字节数组，则返回 false。否则，由于 NULLFAIL 规则，整个合约立即失败。

- `bool checkMultiSig(Sig[M] sigs, PubKey[N] pks)`

  如果有且仅有 M 个签名与 N 个公钥中的 M 个匹配，则返回 true。M 和 N 可以是任意数字，只要 M <= N。如果所有签名都是空字节数组，则返回 false。否则，整个合约立即失效。

5.2.4 `bytes` 操作

- 与 `int` 之间的转换

`bytes` 可以使用函数 `unpack` 转换为 `int` 。使用采用小端格式的 符号-值表示法，其中最高有效位表示符号（`0` 表示正，`1` 表示负）。`int` 可以使用 `pack` 转换为 `bytes`。

```
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()` 返回长度。

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

- **切片操作符** - b[start:end] 返回 b 的子数组，从索引 start （包含）到 end （不包含）。start 如果省略则为 0 ， end 如果省略则为数组的长度。

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

- **拼接**

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

- reverseBytes(bytes b, static const int size)

  返回 b 的反向字节，size 是 b 字节的大小。注意 size 必须是编译时常量 。在小端和大端之间转换数字时，它通常很有用。

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

CHAPTER 6

标准合约

6.1 多个合约

一个文件可以定义多个合约。在这种情况下，最后一个合约作为主合约并且被编译。其他合约是依赖项。

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。

合约可以通过 new 来实例化。require 函数的参数是条件表达式，调用合约的 public 函数可以作为条件表达式传入。

```
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 库

库与合约相同，只是它不包含任何公共函数。它仅用于由合约或其他库导入。因此它不能被独立部署和调用。它经常用于对相关常量和静态函数进行分组。

```
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。

6.4.1 Utils 库

`Utils` 库提供了一组常用的实用函数，例如函数 `Utils.fromLEUnsigned` 将有符号整数 `n` 转换为小端字节序的无符号整数。函数 `buildOutput(bytes outputScript, int outputSatoshis) : bytes` 从其脚本和 satoshi 数量构建一个 tx 输出。

以下示例显示了如何在 `RabinSignature` 合约的中使用标准库的 `Utils` 库。

```
library RabinSignature {
    static function checkSig(bytes msg, RabinSig sig, RabinPubKey pubKey) : bool {
        int h = Utils.fromLEUnsigned(hash(msg + sig.padding));
        return (sig.s * sig.s) % pubKey == h % pubKey;
    }

    static 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 :]);
```

(下页继续)

(续上页)

```
    }
}
```

6.4.2 Tx 库

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. nVersion of the transaction (4-byte little endian)
2. hashPrevouts (输入的输出点哈希 32 字节哈希值)
3. hashSequence (序列号哈希 32 字节哈希值)
4. outpoint (此输入的输出点 32 字节哈希值 + 4 字节小端)
5. scriptCode of the input (输入对应的 UTXO 的锁定脚本)
6. value of the output spent by this input (此输入对应的输出中包含的聪数 8 字节小端)
7. nSequence of the input (此输入的序列号 4 字节小端)
8. hashOutputs (输出的哈希 32 字节哈希值)
9. nLocktime of the transaction (交易的 nLocktime 4 字节小端)
10. sighash type of the signature (签名类型 4 字节小端)

例如，合约 `CheckLockTimeVerify` 确保合约中的币是时间锁定的，并且不能在时间达到 `matureTime` 之前花费，类似于 OP_CLTV。

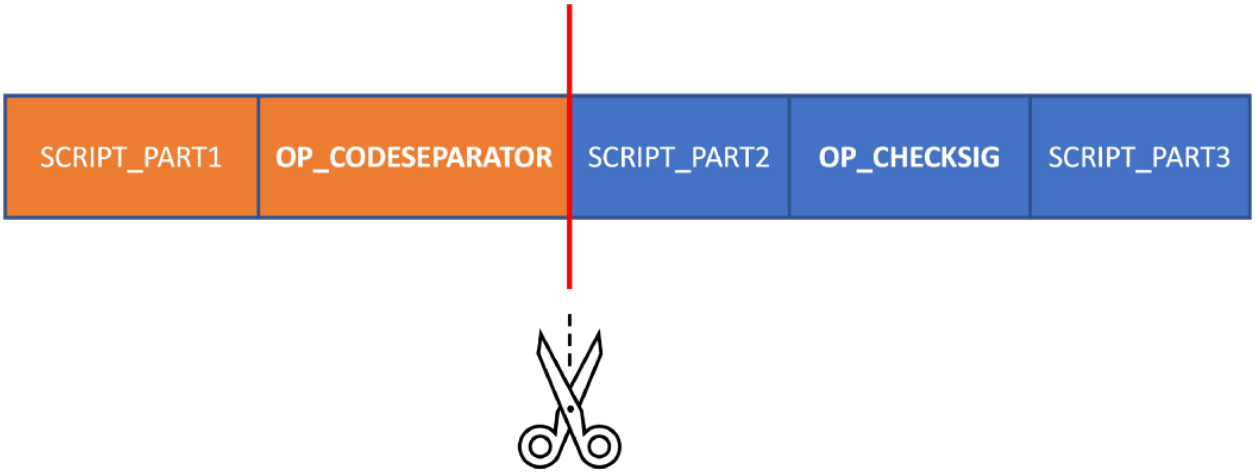
```
contract CheckLockTimeVerify {
    int matureTime;

    public function spend(SigHashPreimage txPreimage) {
        // using Tx.checkPreimage() to verify txPreimage
        require(Tx.checkPreimage(txPreimage));

        require(SigHash.nLocktime(txPreimage) >= this.matureTime);
    }
}
```

更多细节可以在这篇文章 OP_PUSH_TX 技术 中找到。要自定义 ECDSA 签名，例如选择 sighash 类型，有一个名为 `Tx.checkPreimageSigHashType()` 的版本，支持自定义 sighash 类型。要自定义临时密钥，有一个更通用的版本，称为 `Tx.checkPreimageAdvanced()`。请参阅 高级 OP_PUSH_TX 技术。

原像的 ScriptCode 通常包含整个锁定脚本。唯一的例外是当其中有 OP_CODESEPARATOR (OCS) 时。在这种情况下，scriptCode 是锁定脚本，但在执行 OP_CHECKSIG 之前删除包括最后执行的 OCS 之前的所有内容。



`Tx` 库提供了一组 OCS 版本方法来检查这个不包含完整锁定脚本的原像。在许多情况下，使用 OP_PUSH_TX 时不需要 scriptCode 或只需其一部分。OCS 可以用来削减它的大小。例如，在下面的合约中，只需要整个原像的 nLocktime。我们使用 `Tx.checkPreimageOCS()`，传统的 `Tx.checkPreimage()` 的变体。唯一的区别是 OCS 是在前者的 OP_CHECKSIG 之前插入的。另请注意，我们将 `Tx.checkPreimageOCS()` 作为最后一条语句以达到最大优化效果。

```
contract CheckLockTimeVerifyOCS {
    int matureTime;

    public function unlock(SigHashPreimage preimage) {
        require(SigHash.nLocktime(preimage) > this.matureTime);
        require(Tx.checkPreimageOCS(preimage));
    }
}
```

6.4.3 SigHash 库

sCrypt 还提供了一个 `SigHash` 库来访问原像中的各个字段。例如，我们通常使用 `SigHash.scriptCode` 访问原像的 `scriptCode` 字段，使用 `SigHash.value` 访问原像的 value 字段，即在此合约中花费的比特币数量。

```
contract Clone {

    public function unlock(SigHashPreimage txPreimage) {
        require(Tx.checkPreimage(txPreimage));

        bytes scriptCode = SigHash.scriptCode(txPreimage);
        int satoshis = SigHash.value(txPreimage);
        bytes output = Utils.buildOutput(scriptCode, satoshis);
        require(hash256(output) == SigHash.hashOutputs(txPreimage));
    }
}
```

6.4.4 HashedMap 库

HashedMap 库提供了一种类似于哈希表的数据结构。唯一键和它们对应的值在存储之前被散列。*HashedMap* 的大多数函数不仅需要一个键，还需要它的索引，按键的哈希升序排列。

构造函数

- `HashedMap(bytes data)` 使用一些初始数据创建一个 `HashedMap` 实例。

```
HashedMap<bytes, int> map = new HashedMap<bytes, int>(b'');
// key and value types can be omitted
HashedMap<int, bool> map1 = new HashedMap(b'');
// key and value types cannot be omitted since they cannot be inferred
auto map2 = new HashedMap<int, int>(b'');
```

SortedItem

`SortedItem <T>` 是一个通用结构体，它包含一个类型为 T 的 *item* 及一个键索引 *idx*。

```
struct SortedItem<T> {
  T item;
  int idx;
}
```

For most functions of *HashedMap*, a parameter named *keyWithIdx* of this type is required. It means that the *key* and its corresponding *keyIndex* should always be provided together.

实例方法

- `set(SortedItem <K> keyWithIdx, V val) : bool` 使用 *keyIndex* 给定的键索引插入或更新 (*key*, *val*) 对。如果成功，则返回 *true* ；否则返回 *false* 。

```
require(map.set({b'1234', 0}, 1)); // insert
require(map.set({b'1234', 0}, 2)); // update it
```

- `canGet(SortedItem <K> keyWithIdx, V val): bool` 检查是否存在 (*key*, *val*) 对，*keyIndex* 是键索引。如果成功，则返回 *true* ；否则返回 *false* 。

```
require(map.canGet({b'1234', 0}, 2));
```

- `has(SortedItem <K> keyWithIdx) : bool` 检查 map 中是否存在 *key*，其键索引为 *keyIndex*。如果两个条件都满足，则返回 *true* ；否则返回 *false*。

```
require(map.has({b'1234', 0}));
```

- `delete(SortedItem <K> keyWithIdx) : bool` 删除给定 *key* 的条目，键索引是 *keyIndex*。如果成功，则返回 *true*；否则返回 *false* 。

```
require(map.delete({b'1234', 0}));
```

- `clear() : bool` 删除 map 的所有条目。

```
map.clear();
```

- `size() : int` 返回 HashedMap 的大小，比如它包含的键的数量。

```
int s = map.size();
```

- `data() : bytes` 返回 HashedMap 的序列化数据表示。

```
bytes b = map.data();
// this creates a deep copy of the map
HashedMap<int, bool> mapCopy = new HashedMap(b);
```

6.4.5 HashedSet 库

HashedSet 库提供了一个类似集合的数据结构。它可以看作是一个特殊的 *HashedMap*，其中一个值与其键相同，因此被省略。唯一值在存储之前经过哈希处理。*HashedSet* 的大多数函数都需要一个索引，按值的 sha256 哈希值升序排列。与 *HashedMap* 类似，这些函数也使用 *SortedItem* 类型参数。

构造函数

- `HashedSet(bytes data)` 使用初始数据 data 创建一个 `HashedSet` 实例。

```
struct ST {
  int x;
  bool y;
}

HashedSet<ST> set = new HashedSet<ST>(b'');
// key and value types can be omitted
HashedSet<ST> set1 = new HashedSet(b'');
// key and value types cannot be omitted since they cannot be inferred
auto set2 = new HashedSet<ST>(b'');
```

实例方法

- `add(SortedItem <E> entryWithIdx) : bool` 添加 *entry* 以使用 *index* 给出的键索引进行设置。如果成功，则返回 *true*；否则返回 *false* 。

```
require(set.add({b'1234', 0}));
```

- *has(SortedItem <E> entryWithIdx) : bool* 检查集合中是否存在 *entry* 条目，其键索引为 *index* 。如果两个条件都满足，则返回 *true*；否则返回 *false* 。

```
require(set.has({b'1234', 0}));
```

- `delete(SortedItem <E> entryWithIdx) : bool` 删除给定 *entry* 条目，键索引为 *index* 。如果成功，则返回 *true* ；否则返回 *false* 。

```
require(set.delete({b'1234', 0}));
```

- `clear() : bool` 删除集合的所有条目。

```
set.clear();
```

- `size() : int` 返回集合的大小，即它包含的条目数。

```
int s = set.size();
```

- `data() : bytes` 返回集合的内部序列化数据。

```
bytes b = set.data();
// this creates a deep copy of the set
HashedSet<ST> setCopy = new HashedSet(b);
```

6.4.6 Constants 库

sCrypt 在 Constants 库中定义了一些常用的常量值。你可以在代码中的任何位置使用这些常量。

```
library Constants {

    // number of bytes to denote input sequence
    static const int InputSeqLen = 4;
    // number of bytes to denote output value
    static const int OutputValueLen = 8;
    // number of bytes to denote a public key (compressed)
    static const int PubKeyLen = 33;
    // number of bytes to denote a public key hash
    static const int PubKeyHashLen = 20;
    // number of bytes to denote a tx id
    static const int TxIdLen = 32;
    // number of bytes to denote a outpoint
    static const int OutpointLen = 36;
}
```

6.4.7 完整列表

合约	构造函数参数	公共函数
Utils	None	toLEUnsigned(int n, int l) : bytes fromLEUnsigned(bytes b) : int readVarint(bytes b) : bytes writeVarint(bytes b) : bytes buildOutput(bytes outputScript, int outputSatoshis) : bytes buildPublicKeyHashScript(PubKeyHash pubKeyHash) : bytes buildOpreturnScript(bytes data) : bytes isFirstCall(SigHashPreimage preimage) : bool // return whether is the first call of series public function calls in stateful contract

Tx	None	checkPreimage(SigHashPreimage preimage) : bool
		checkPreimageOpt(SigHashPreimage rawTx) : bool
		checkPreimageOpt_(SigHashPreimage rawTx) : bool // set sigHashType in ASM
		checkPreimageSigHashType(SigHashPreimage txPreimage, SigHashType sigHashType) : bool
		checkPreimageAdvanced(SigHashPreimage rawTx, PrivKey privKey, PubKey pubKey, int inverseK, int r, bytes rBigEndian, SigHashType sigHashType) : bool

CHAPTER 7

编译时常量

编译时常量 (Compile Time Constant) 是可以在编译时计算的值。编译时常量有四种类型。

- 常量，比如数字、布尔值、*bytes* 值
- 归纳变量
- 合约的静态常量属性，用常量初始化
- 静态常量函数参数

有几种情况只允许使用编译时常量。

- 循环限制
- 数组大小
- 使用索引运算符 写入数组元素
- 声明为 `static const`[1] 的函数参数，例如 `reverseBytes(bytes b, static const int size)` 和 `repeat(T e, static const int size)` 中的 `size`

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

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

(下页继续)

[1] 注意：顺序很重要：声明函数参数时不允许使用 `const static`，但在声明属性时允许。

(续上页)

```
    // 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

有状态合约

比特币/sCrypt 合约使用未花费交易输出 (UTXO) 模型：合约位于 UTXO 内，决定如何使用 UTXO 中的比特币。当一个 UTXO 被花费（即成功调用 sCrypt 合约公共函数）时，合约终止。对于保持状态并能够在携带可变状态的同时被多次调用的合约，必须遵循以下这些步骤。

8.1 状态装饰器

使用装饰器 `@state` 声明属于状态的任何属性。状态属性可以像普通属性一样使用。

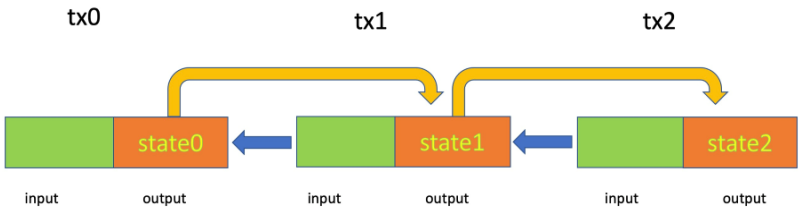
```
contract Counter {
    @state
    int counter;

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

}
```

8.2 更新状态

合约可以通过将状态存储在锁定脚本中来跨链式交易保持状态。在以下示例中，合约从 state0 到 state1，然后到 state2。交易 1 tx1 中的输入是在 tx0 中花费 UTXO，而 tx2 花费 tx1。



当你准备好将新状态传递到下一个 UTXO 时，只需使用两个参数调用内置函数 this.updateState()：- *txPreimage*：它是*preimage* 当前支出交易。它必须只有一个输出并且其中包含新状态。- *amount*：单个输出中的 *satoshis* 数量。

该函数是为每个有状态合约自动生成的，即，一个合约至少有一个用 @state 修饰的属性。如果你需要自定义的 *sighash* 类型，请使用 updateStateSigHashType，不同于默认的 SigHash.ALL | SigHash.FORKID。

下面是一个示例合约，它记录了 increment() 被调用的次数。

```
contract Counter {
    @state
    int counter;

    public function increment(SigHashPreimage txPreimage, int amount) {
        // mutate state
        this.counter++;

        require(this.updateState(txPreimage, amount));

        // customed sighash type
        // require(this.updateStateSigHashType(txPreimage, amount, SigHash.SINGLE |␣
↪SigHash.FORKID));
    }
}
```

8.3 高级

如果你需要对状态进行更细粒度的控制，例如，在支出交易中有多个输出，你可以调用另一个内置函数 this.getStateScript() 来获取包含最新状态属性的锁定脚本。

接下来，你使用*OP_PUSH_TX* 来确保包含新状态的输出进入当前的支出交易。它相当于上面使用 this.updateState() 的合约。

```
contract Counter {
    @state
    int counter;

    public function increment(SigHashPreimage txPreimage, int amount) {
        // mutate state
        this.counter++;

        require(Tx.checkPreimage(txPreimage));

        // get the locking script containing the latest stateful properties
        bytes outputScript = this.getStateScript();

        // construct an output from its locking script and satoshi amount
        bytes output = Utils.buildOutput(outputScript, amount);
        // only 1 input here
        require(hash256(output) == SigHash.hashOutputs(txPreimage));
    }
}
```

8.4 限制

对于任何访问状态属性的公共函数，它必须包含一个通过 `Tx.checkPreimage()` 验证的 `SighashPreimage` 参数，即使用*OP_PUSH_TX*。这不适用于任何非公共函数，包括构造函数。

```
contract Counter {
    @state
    int counter;

    constructor(int counter) {
        // OK: not a public function
        this.counter = counter;
    }

    public function increment(SigHashPreimage txPreimage, int amount) {
        // OK
        this.counter++;

        require(Tx.checkPreimage(txPreimage));
```

(下页继续)

(续上页)

```
    }

    public function foo(SigHashPreimage txPreimage, int amount) {
        require(Tx.checkPreimageOpt(txPreimage));

        // OK
        this.counter++;

        require(true);
    }

    public function bar(SigHashPreimage txPreimage) {
        // Not OK: missing Tx.checkPreimage*()
        this.counter++;

        require(true);
    }

    public function baz(int i) {
        // Not OK: missing SigHashPreimage
        this.counter++;

        require(true);
    }

    function baz() : int {
        // OK: not a public function
        return this.counter;
    }
}
```

CHAPTER 9

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 10

R-Puzzle

在 R-puzzle 中，永远不会公开临时私钥 `k`。而是公开 `r` 和来自 `r` 的签名，`r` 是 `k` 对应的公钥的 x 坐标，并使用现有的 `checkSig` 来证明知道 `k`。更多信息可以在 R-Puzzle 章节中找到。

```
contract RPuzzle {
    Ripemd160 rhash;

    // extract r from DER-encoded signature
    static function extractRFromSig(Sig sig) : bytes {
        int rlen = unpack(sig[3 : 4]);
        return sig[4 : 4 + rlen];
    }

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

CHAPTER 11

阿克曼 (Ackermann) 函数

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：

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

sCrypt 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。下面我们提供一个更简单的版本。

```
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 12

拉宾签名 (Rabin Signature)

拉宾签名 是比特币中使用的 ECDSA 的另一种数字签名形式。

```
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 13

多方哈希谜题 (Multiparty Hash Puzzles)

在哈希谜题合约中，花费者必须提供一个原像 x，它散列到一个预定义的值 y 来解锁一个 UTXO。它可以扩展到多方，因此必须提供多个原像，使得 y1 = H(x1), y2 = H(x2), ···, yN = H(xN))[1]. 下面显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 是一个哈希函数。在线哈希计算器

当 N 很大时，上述解决方案是有问题的，因为所有的 N 哈希都必须包含在锁定脚本中，从而使交易膨胀。相反，我们可以将所有 y 组合成一个 y，使得 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] || 表示串联。

CHAPTER 14

内联汇编

脚本是一种低级语言，充当 比特币虚拟机 的程序集。通常，开发人员不必直接处理它，可以使用 sCrypt 等高级语言。但是，有些情况下需要使用 Script。例如，自定义脚本经过优化，因此比 sCrypt 生成的脚本更高效。或者脚本是使用外部工具（如 MiniForth）生成的，需要集成到 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。

14.1 调用约定

对于用脚本编写的函数，它的整个函数主体必须用 `asm` 块包围。函数参数位于栈顶，与声明的顺序相反。例如，对于签名为 `function foo(int p0, bytes p1, bool p2) : int` 的函数，`p2` 位于栈顶，`p1` 是从栈顶倒数第二个，而 `p0` 进入 *asm* 是第三个。当退出 `asm` 模式时，需要弹出栈里的所有参数，并把返回值放到栈顶。栈里其他元素保持不变。

三个汇编函数如下所示。

```
// 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
    }
}
```

14.2 汇编变量

变量可以通过前缀 $ 在 asm 模式中使用。与脚本的其余部分不同，它被完整复制到最终脚本输出中，变量以其作用域为前缀以避免名称冲突，由它所在的函数和契约唯一标识。例如，假设在合约 contractFoo 内的函数 func 中使用了一个变量 $var，它将在最终的脚本输出中显示为 $contractFoo.func.var。

一个例子如下所示。

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

14.3 循环

循环语法 也可以在 asm 中使用。

```
public function unlock(int x) {
    asm {
        OP_DUP
        loop (N) : i {
            loop (N) : j {
                i
                j
                OP_ADD
                OP_ADD
            }
        }
        13
        OP_NUMEQUAL
        OP_NIP
    }
}
```

等价的 sCrypt 代码是:

```
public function unlock(int x) {
    int sum = x;
    loop (N) : i {
        loop (N) : j {
            sum += (i + j);
        }
    }
    require(sum == 0x13);
}
```

i 和 j 是归纳变量。

14.4 字符串

字符串是一个双引号 UTF8 字符串，可以在 asm 中使用。

```
static function equal(bytes msg) : bool {
    asm {
        "你好 world! "
        OP_EQUAL
    }
}
```

14.5 注意

内联汇编绕过了 sCrypt 的许多功能，例如类型检查。使用此高级功能时必须格外小心。此外，为了与外部工具兼容，它不区分大小写。

CHAPTER 15

联系方式

scrypt.io

Slack

Telegram

CHAPTER 16

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 。