Working on scripts with logical opcodes

Thomas Kerin

- Thanks to the speakers committee and conference organizers!

Logical opcodes

Script has logical opcodes - `IF/NOTIF/ELSE/ENDIF`. Allows for different redeem conditions to be set, depending on who is spending the bitcoins, what information they have, etc.

LN: Commitment transaction to self output:

```
OP_IF
        [revocationPubKey]
OP_ELSE
        [relativeTimelock] OP_CHECKSEQUENCEVERIFY OP_DROP
        [localDelayedPubKey]
OP_ENDIF
OP_CHECKSIG
```

Two people can spend from this script. Immediately one can spend with the revocation key, or with the delayed key, wait for the timelock to elapse.

https://github.com/lightningnetwork/lightning-rfc/blob/master/03-transactions.md

LN: Offered HTLC Outputs

```
OP_DUP OP_HASH160 <RIPEMD160(SHA256(revocationpubkey))> OP_EQUAL
OP_IF
    OP_CHECKSIG
OP_ELSE
    <remote_htlcpubkey> OP_SWAP OP_SIZE 32 OP_EQUAL
    OP_NOTIF
        # To local node via HTLC-timeout transaction (timelocked).
        OP_DROP 2 OP_SWAP <local_htlcpubkey> 2 OP_CHECKMULTISIG
    OP_ELSE
        # To remote node with preimage.
        OP_HASH160 <RIPEMD160(payment_hash)> OP_EQUALVERIFY
        OP_CHECKSIG
    OP_ENDIF
OP_ENDIF
```

https://github.com/lightningnetwork/lightning-rfc/blob/master/03-transactions.md

They're as big as they need to be, so really want software to be able to deal with these, by determining a scripts signing requirements, and checking that signing instructions are correct.

Quick intro, logical opcode internals

Logical opcodes control the scripts execution flow by managing a `vector<bool> vfExec`.

Before every opcode is interpreted, we check if execution is currently active:

```
bool fExec = !count(vfExec.begin(), vfExec.end(), false);

/* .. */

if (fExec && 0 <= opcode && opcode <= OP_PUSHDATA4) {
    /* .. */
} else if (fExec || (OP_IF <= opcode && opcode <= OP_ENDIF)) {
    switch (opcode) {
       // interpret actual instructions
    }
}
```

`OP_IF` / `OP_NOTIF` pop from the mainStack and cast the value into a boolean.

`OP_IF` pushes the `fValue` directly, `OP_NOTIF` pushes `!fValue`

```
case OP_IF:
case OP_NOTIF:
{
    // <expression> if [statements] [else [statements]] endif
    bool fValue = false;
    if (fExec)
    {
       if (stack.size() < 1)
            return set_error(serror, SCRIPT_ERR_UNBALANCED_CONDITIONAL);
        valtype& vch = stacktop(-1);
        fValue = CastToBool(vch);
        if (opcode == OP_NOTIF)
            fValue = !fValue;
        popstack(stack);
    }
    vfExec.push_back(fValue);
}
break;
```

`OP_ELSE` inverts the first value on vfExec

```
    case OP_ELSE:
    {
        if (vfExec.empty())
            return set_error(serror, SCRIPT_ERR_UNBALANCED_CONDITIONAL);
        vfExec.back() = !vfExec.back();
    }
    break;
```

`OP_ENDIF` pops a value from the vfExec, moving us out of this conditional section

```
    case OP_ENDIF:
    {
        if (vfExec.empty())
            return set_error(serror, SCRIPT_ERR_UNBALANCED_CONDITIONAL);
        vfExec.pop_back();
    }
    break;
```

Execution flow with conditional sections

So we know there's a stack which controls execution flow, and depending on its state different sections of the script will be active, meaning different requirements in terms of signing.

In simple scripts `fExec` is `true` the entire time, as `vfExec` is never touched.

```
     [ ]
      t   OP_DUP
      t   OP_HASH160
      t   [pubKeyHash1]
      t   OP_EQUALVERIFY
      t   OP_CHECKSIG
```

```
[01] [00] [1]
  t    t     t     OP_HASH160 OP_DUP [Hash160(commitPreimage)] OP_EQUAL
  t    t     t     OP_IF
                        [timeout: 0x0300a41f] OP_CHECKSEQUENCEVERIFY OP_2DROP
                        [aliceKey]
  t    t     t     OP_ELSE
  t    t               [Hash160(revokePreimage)] OP_EQUAL
  t    t               OP_NOTIF
  t                        0x044ad6ed5a OP_CHECKLOCKTIMEVERIFY OP_DROP
  t    t               OP_ENDIF
  t    t               [bobKey]
  t    t     t     OP_ENDIF
  t    t     t     OP_CHECKSIG

   [1]: Sig(AliceKey) [CommitPreimage]
   [00]: Sig(BobKey) OP_0 OP_0
   [01]: Sig(BobKey) [RevokePreimage] OP_0
```

The [1] [0,1], [0,0] values are so named after the values pushed by IF / NOTIF in order to direct flow control along that pathway, bearing in mind they get modified by other instructions later.

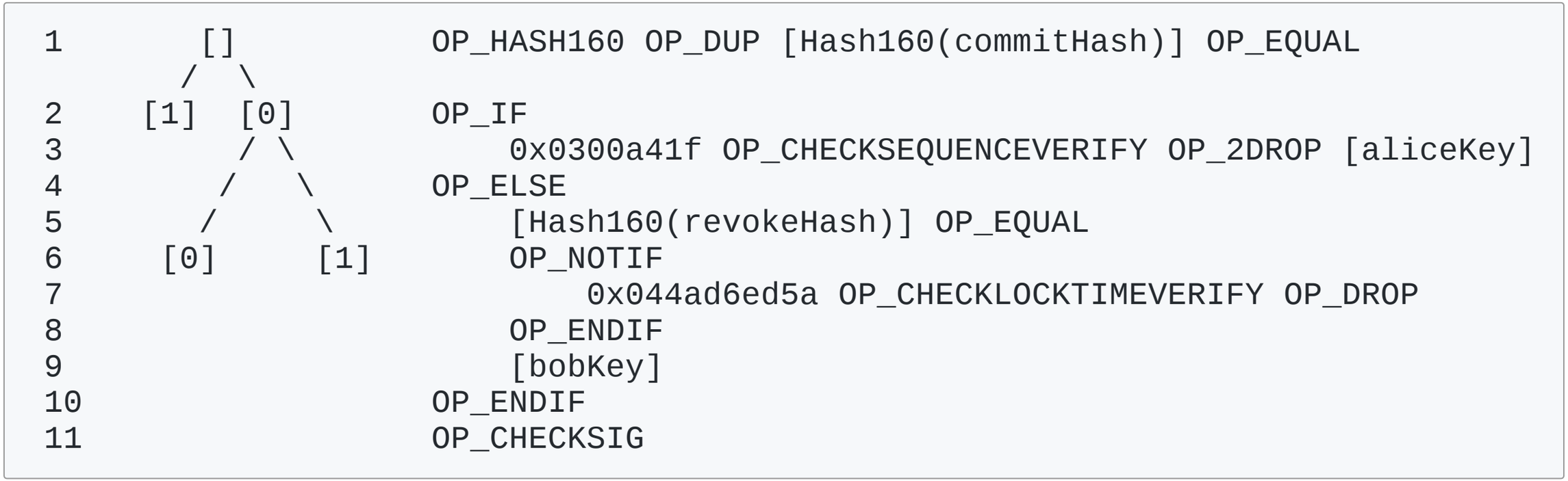
Determining individual branches, and their identifier

For us to ask software that supports signing 'everything' to sign for a certain branch, we need to be able to identify the branch, and so it can then check what the signing requirements are.

To determine all the branches, we iterate over the scripts instructions and partially interpret them: only processing logical opcodes and maintaining a vfExec for each code path as they are discovered.

We start out with one code path. We learn about new codepaths when we process IF and NOTIF. And as we run through opcodes, we associate them with any currently active branches.

After processing the script, we can detect unbalanced conditional sections by checking each vfExec is empty.


1 [] OP_HASH160 OP_DUP [Hash160(commitHash)] OP_EQUAL
2 [1] [0] OP_IF
3 0x0300a41f OP_CHECKSEQUENCEVERIFY OP_2DROP [aliceKey]
4 OP_ELSE
5 [Hash160(revokeHash)] OP_EQUAL
6 [0] [1] OP_NOTIF
7 0x044ad6ed5a OP_CHECKLOCKTIMEVERIFY OP_DROP
8 OP_ENDIF
9 [bobKey]
10 OP_ENDIF
11 OP_CHECKSIG

1. All code paths (1 currently). share the start because vfExec = []
2. Execution forks with OP_IF. [] forks into [1] or [0].
3. This is the first conditional branch, for it to run it's vfExec would need to be [1]
4. ELSE inverts the only `vfExec` value, either (0 -> 1) or (1 -> 0).
5. Code paths for which the first IF yielded [0] will execute this section, because ELSE mutated the only value on vfExec to true.
6. Code paths that started from [0] fork into [0, 1], and [0, 0].
7. This section is only active where the NOTIF branch pushed true, hence this path is [0, 1]
8. The conditional section 7 is closed. [0, 1] loses a vfExec value, and still remains active. [0, 0] loses a value, becoming active.
9. This section is shared for [0, 1] and [0, 0]
10. ENDIF pops the last vfExec value from all code paths, making execution active globally again.

Path [1] - requires knowledge of the preimage of Hash160(commitHash), a CSV lock of 24 hours, and a signature by Alice.

```
1   OP_HASH160 OP_DUP [commitHash: 0x1436e11d] OP_EQUAL
2   OP_IF
3       0x0300a41f OP_CHECKSEQUENCEVERIFY OP_2DROP [aliceKey: 0x21037569...]
4   OP_ELSE
6       OP_NOTIF
8       OP_ENDIF
10  OP_ENDIF
11  OP_CHECKSIG
```

Path [0, 1] - allows Bob to spend without the preimage of Hash160(revokeHash), but only after some date.

```
1   OP_HASH160 OP_DUP [commitHash: 0x1436e11d] OP_EQUAL
2   OP_IF
4   OP_ELSE
5       [revokeHash: 0x14442328...] OP_EQUAL
6       OP_NOTIF
7           0x044ad6ed5a OP_CHECKLOCKTIMEVERIFY OP_DROP
8       OP_ENDIF
9       [bobKey: 0x21036172...]
10  OP_ENDIF
11  OP_CHECKSIG
```

Path [0, 0] - allows Bob to spend with the preimage of Hash160(revokeHash) at any time.

```
1   OP_HASH160 OP_DUP [commitHash] OP_EQUAL
2   OP_IF
4   OP_ELSE
5       [revokeHash: 0x14442328b49e3071bddbafc0f8b7b6aad0b7dbcc63] OP_EQUAL
6       OP_NOTIF
8       OP_ENDIF
9       [bobKey]
10  OP_ENDIF
11  OP_CHECKSIG
```

Signing

https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki#escrow-with-timeout

```
OP_IF
    2
    0x039dcd06ba35bdb98ce55b848f19e5171b7e8e1200be8036cbf76389dc3229f5b2
    0x03e4dd72c88df5759c89e13432cec27080349c82f3d78ad2cce94d9ca595ecef8c
    0x0252819de2777f73a94457e67570ce9fa33bb9344e8c8467149e98d4f8d8fc5712
    3
    OP_CHECKMULTISIG
OP_ELSE
    0x805141 OP_CHECKSEQUENCEVERIFY OP_DROP
    0x039dcd06ba35bdb98ce55b848f19e5171b7e8e1200be8036cbf76389dc3229f5b2
    OP_CHECKSIG
OP_ENDIF
```

MAST:

MAST "Merkelized Abstract Syntax Tree" allows for for different redemption pathways to be committed for a particular script, such that redeeming on one pathway reveals only that pathways instructions - the others remain private.

(Currently) Two proposals to add MAST to bitcoin:

- BIP 114 "[Segwit V1] Merkelized Abstract Syntax Tree" by jl2012.
- BIP's 116+117 "CHECKMERKLEBRANCHVERIFY" and "Tail Call Execution Semantics" by maaku.

This line of research was started after reading the BIP114 MAST proposal and wondering about a method to flatten a legacy script into it's MAST equivalents.

The example had so many features which made it seem a technically interesting problem. I haven't yet succeeded at this, although it lead me through a few topics which were worthwhile (branch processing) which is useful for people dealing with logical opcodes on the network today.

```
HASH160 DUP [revokeHash] EQUAL
IF
    "24h" CHECKSEQUENCEVERIFY 2DROP
    [AliceKey]
ELSE
    [commitRevocationHash] EQUAL
    NOTIF
        "Timestamp" CHECKLOCKTIMEVERIFY DROP
    ENDIF
    [BobKey]
ENDIF
CHECKSIG
```

- HASH160 [revokeHash] EQUALVERIFY "24h" CHECKSEQUENCEVERIFY DROP [AliceKey] CHECKSIGVERIFY
- HASH160 [commitRevocationHash] EQUALVERIFY [BobKey] CHECKSIGVERIFY
- "Timestamp" CHECKLOCKTIMEVERIFY DROP [BobKey] CHECKSIGVERIFY

To MASTIFY a script:

- Determine mutually exclusive code paths of a script
- IF/NOTIF (tests for flow control) replaced with VERIFY / NOT VERIFY (assertions about the state of the stack)
- Joining [OP_CHECKSIG, OP_CHECKMULTISIG, OP_EQUAL] with a following OP_VERIFY
- Trim instructions used by the old script purely for flow control purposes, which are unnecessary in a MASTified branch (hashlocks, for example)

Projects doing some of this already:

- https://github.com/Bit-Wasp/bitcoin-php (branch parsing, not MAST)
- https://github.com/kallewoof/btcdeb (branch parsing, MAST, but without instruction optimization)
- a web based tool for MAST by Nicolas Dorier (branch parsing, MAST, but without instruction optimization)

That's all folks!