Acacia Dai
Cryptocurrency Cabal
12/7/2015

The Evolution of Bitcoin Script Interpreter

Background

The goal of this project was to explore how the Bitcoin Script Interpreter evolved over time, as well as to look into major bugs in the Bitcoin Core and the process by which updates are made to the Core.

The Bitcoin Script Interpreter is the code that defines opcodes and methods used in transaction scripts. The Interpreter code has been continuously changing since Bitcoin was launched, sometimes with corrections to overlooked mistakes and sometimes with entirely new methods. To explore these changes over time, I compared different releases of the Script Interpreter and looked for notable changes such as updated opcodes and newly added methods.

Comparing Release 0.1.0 (Original) to 0.3.24 (7/8/2011)

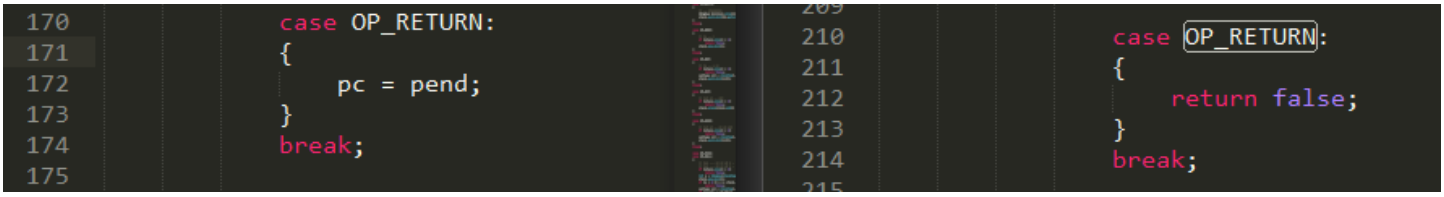
- A script size limit of 10,000 opcodes was added.

```
72  bool EvalScript(vector<vector<unsigned char> >& stack, const
        CScript& script, const CTransaction& txTo, unsigned int
        nIn, int nHashType)
73  {
74      CAutoBN_CTX pctx;
75      CScript::const_iterator pc = script.begin();
76      CScript::const_iterator pend = script.end();
77      CScript::const_iterator pbegincodehash = script.begin();
78      opcodetype opcode;
79      valtype vchPushValue;
80      vector<bool> vfExec;
81      vector<valtype> altstack;
82      if (script.size() > 10000)
83          return false;
84      int nOpCount = 0;
```

- Pushed values greater than 520 and irrelevant opcodes are ignored.

```
96              if (!script.GetOp(pc, opcode, vchPushValue))
97                  return false;
98              if (vchPushValue.size() > 520)
99                  return false;
100             if (opcode > OP_16 && ++nOpCount > 201)
101                 return false;
102
103             if (opcode == OP_CAT ||
104                 opcode == OP_SUBSTR ||
105                 opcode == OP_LEFT ||
106                 opcode == OP_RIGHT ||
107                 opcode == OP_INVERT ||
108                 opcode == OP_AND ||
109                 opcode == OP_OR ||
110                 opcode == OP_XOR ||
111                 opcode == OP_2MUL ||
112                 opcode == OP_2DIV ||
113                 opcode == OP_MUL ||
114                 opcode == OP_DIV ||
115                 opcode == OP_MOD ||
116                 opcode == OP_LSHIFT ||
117                 opcode == OP_RSHIFT)
118                 return false;
```

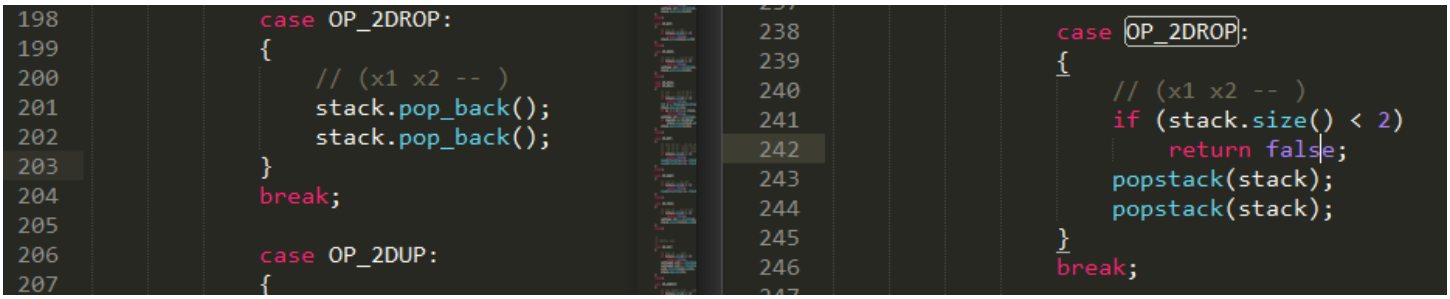
- OP_RETURN: changed to `return false` instead of `pc = pend`.
  - Fixes the 1 RETURN Bug
  - This change is also seen in OP_VERIFY, OP_CHECKSIG, OP_CHECKSIGVERIFY, OP_CHECKMULTISIG, CHECKMULTISIGVERIFY, OP_EQUAL, and OP_VERIFY



```
170             case OP_RETURN:
171             {
172                 pc = pend;
173             }
174             break;
175
```

```
210             case OP_RETURN:
211             {
212                 return false;
213             }
214             break;
```

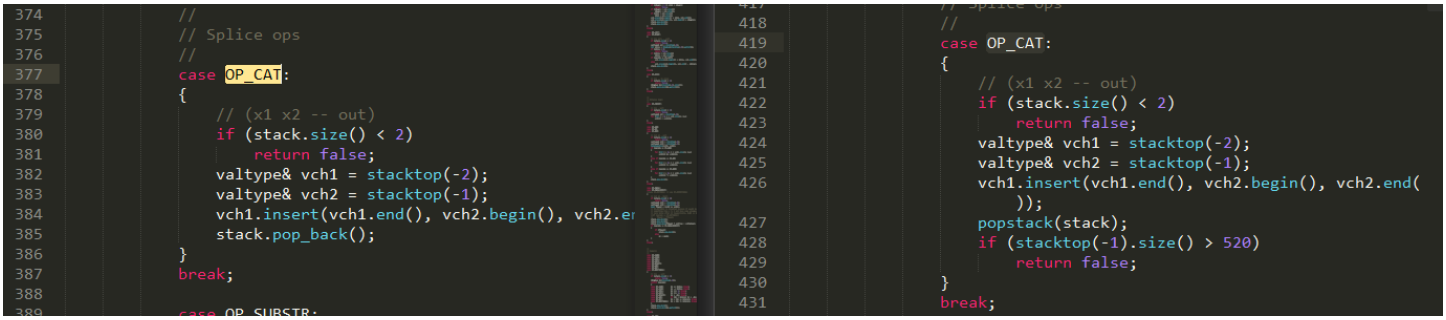
- OP_2DROP: added a check to make sure the stack has two opcodes to pop off.
  - Interestingly, this kind of check was already in place for other opcodes that would need to check the size of the stack. It seems like an oversight that this was missing in the 0.1.0 version.



```
198             case OP_2DROP:
199             {
200                 // (x1 x2 -- )
201                 stack.pop_back();
202                 stack.pop_back();
203             }
204             break;
205
206             case OP_2DUP:
207             {
```

```
238             case OP_2DROP:
239             {
240                 // (x1 x2 -- )
241                 if (stack.size() < 2)
242                     return false;
243                 popstack(stack);
244                 popstack(stack);
245             }
246             break;
```

- OP_CAT: added size limit of 520 on the data being concatenated.



```
374             //
375             // Splice ops
376             //
377             case OP_CAT:
378             {
379                 // (x1 x2 -- out)
380                 if (stack.size() < 2)
381                     return false;
382                 valtype& vch1 = stacktop(-2);
383                 valtype& vch2 = stacktop(-1);
384                 vch1.insert(vch1.end(), vch2.begin(), vch2.e
385                 stack.pop_back();
386             }
387             break;
388
389             case OP_SUBSTR:
```

```
418             //
419             case OP_CAT:
420             {
421                 // (x1 x2 -- out)
422                 if (stack.size() < 2)
423                     return false;
424                 valtype& vch1 = stacktop(-2);
425                 valtype& vch2 = stacktop(-1);
426                 vch1.insert(vch1.end(), vch2.begin(), vch2.end(
                        ));
427                 popstack(stack);
428                 if (stacktop(-1).size() > 520)
429                     return false;
430             }
431             break;
```

- Other new methods such as `IsStandard()` & `VerifyScript()`.

Comparing Release 0.3.24 to 0.8.0 (2/19/2013)

- New methods `IsCanonicalPubkey()` and `IsCanonicalSignature()` check to make sure the public key and signature are in a standard format. These are incorporated into `OP_CHECKSIG`, `CHECKSIGVERIFY`, `CHECKMULTISIG`, and `CHECKMULTISIGVERIFY`.

- New class `CSignatureCache` which caches valid signatures so that elliptic curve signature checking only needs to be done once instead of twice (once when accepted into memory pool and again when accepted into the block chain).

```
1193  // Valid signature cache, to avoid doing expensive ECDSA signature checking
1194  // twice for every transaction (once when accepted into memory pool, and
1195  // again when accepted into the block chain)
1196
1197  class CSignatureCache
1198  {
1199  private:
1200      // sigdata_type is (signature hash, signature, public key):
1201      typedef boost::tuple<uint256, std::vector<unsigned char>, std::vector<
          unsigned char> > sigdata_type;
1202      std::set< sigdata_type> setValid;
1203      boost::shared_mutex cs_sigcache;
1204
```

- New method `SignN()` checks if a multisignature transaction has the right number of signatures.

```
1419  bool SignN(const vector<valtype>& multisigdata, const CKeyStore& keystore,
          uint256 hash, int nHashType, CScript& scriptSigRet)
1420  {
1421      int nSigned = 0;
1422      int nRequired = multisigdata.front()[0];
1423      for (unsigned int i = 1; i < multisigdata.size()-1 && nSigned <
          nRequired; i++)
1424      {
1425          const valtype& pubkey = multisigdata[i];
1426          CKeyID keyID = CPubKey(pubkey).GetID();
1427          if (Sign1(keyID, keystore, hash, nHashType, scriptSigRet))
1428              ++nSigned;
1429      }
1430      return nSigned==nRequired;
1431  }
```

- Method `IsStandard()` is expanded to handle multisignature transactions.

```
1083  bool IsStandard(const CScript& scriptPubKey)
1084  {
1085      vector<pair<opcodetype, valtype> > vSolution;
1086      return Solver(scriptPubKey, vSolution);
1087  }
```

```
1497  bool IsStandard(const CScript& scriptPubKey)
1498  {
1499      vector<valtype> vSolutions;
1500      txnouttype whichType;
1501      if (!Solver(scriptPubKey, whichType, vSolutions))
1502          return false;
1503
1504      if (whichType == TX_MULTISIG)
1505      {
1506          unsigned char m = vSolutions.front()[0];
1507          unsigned char n = vSolutions.back()[0];
1508          // Support up to x-of-3 multisig txns as standard
1509          if (n < 1 || n > 3)
1510              return false;
1511          if (m < 1 || m > n)
1512              return false;
1513      }
1514
1515      return whichType != TX_NONSTANDARD;
1516  }
```

- Other new methods such as `SignSignature()`, `CombineMultisig()`, `CombineSignatures()`, `getSigOpCount()` and new class `CScriptCompressor`

Comparing Release 0.8.0 to 0.11.2 (11/13/2015)

- Separated into two different files: `script.cpp` and `interpreter.cpp`.
- New method `IsCompressedOrUncompressedPubKey()` checks if public key is compressed.

```
66  bool static IsCompressedOrUncompressedPubKey(const valtype &vchPubKey) {
67      if (vchPubKey.size() < 33) {
68          //  Non-canonical public key: too short
69          return false;
70      }
71      if (vchPubKey[0] == 0x04) {
72          if (vchPubKey.size() != 65) {
73              //  Non-canonical public key: invalid length for uncompressed key
74              return false;
75          }
76      } else if (vchPubKey[0] == 0x02 || vchPubKey[0] == 0x03) {
77          if (vchPubKey.size() != 33) {
78              //  Non-canonical public key: invalid length for compressed key
79              return false;
80          }
81      } else {
82            //  Non-canonical public key: neither compressed nor uncompressed
83            return false;
84      }
85      return true;
86  }
```

- The method `CheckCanonicalSignature()` was expanded and replaced by several methods: `IsValidSignatureEnccoding()`, `IsDefinedHashTypeSignature()`, `CheckSignatureEncoding()`, and `CheckPubKeyEncoding()`.

- All methods and opcodes now return `set_error` codes that provide information about invalidities instead of just returning false.

```
270             if (opcode == OP_CAT ||
271                 opcode == OP_SUBSTR ||
272                 opcode == OP_LEFT ||
273                 opcode == OP_RIGHT ||
274                 opcode == OP_INVERT ||
275                 opcode == OP_AND ||
276                 opcode == OP_OR ||
277                 opcode == OP_XOR ||
278                 opcode == OP_2MUL ||
279                 opcode == OP_2DIV ||
280                 opcode == OP_MUL ||
281                 opcode == OP_DIV ||
282                 opcode == OP_MOD ||
283                 opcode == OP_LSHIFT ||
284                 opcode == OP_RSHIFT)
285                 return set_error(serror, SCRIPT_ERR_DISABLED_OPCODE); //
                    Disabled opcodes.
```

```
416                 case OP_VERIFY:
417                 {
418                     // (true -- ) or
419                     // (false -- false) and return
420                     if (stack.size() < 1)
421                         return set_error(serror,
                                SCRIPT_ERR_INVALID_STACK_OPERATION);
422                     bool fValue = CastToBool(stacktop(-1));
423                     if (fValue)
424                         popstack(stack);
425                     else
426                         return set_error(serror, SCRIPT_ERR_VERIFY);
427                 }
428                 break;
429
430                 case OP_RETURN:
431                 {
432                     return set_error(serror, SCRIPT_ERR_OP_RETURN);
433                 }
434                 break;
```

- About 400 lines of code, containing methods added between Versions 0.3.24 and 0.8.0 have been removed from the end of the program.

Bitcoin Core Bugs

Many bugs have been discovered in the Bitcoin Core code since its launch. Most of the more egregious ones were fixed in 2010 but there were vulnerabilities exploited as recently as last year that resulted in major theft.

1 RETURN Bug (July 2010)

In July 2010, Satoshi and fellow developer Gavin Andresen received a tip-off about this bug that would have made it possible to spend anyone else's Bitcoins[1]. In the early releases of the Bitcoin Core, `OP_RETURN` was defined as:

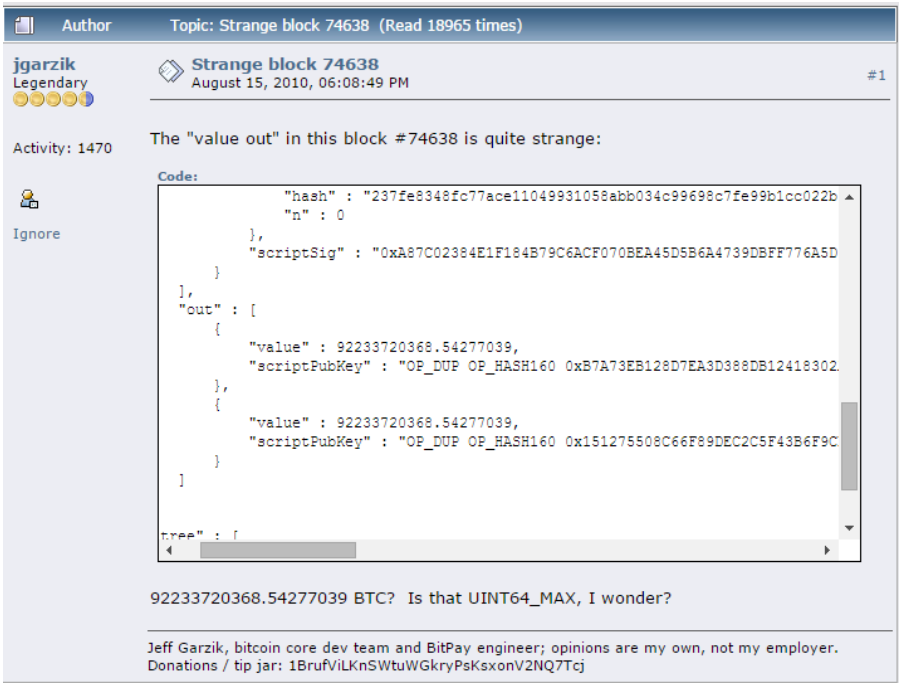
```
    case OP_RETURN:
    {
        pc = pend;
    }
    break;
```

An unlocking script of `OP_1 OP_RETURN` would then always result in a 1 on the top of the stack, as `OP_RETURN` would end the script evaluation without modifying whatever was already on the stack. This was fixed by changing the opcode to return false and end the script.

Combined Output Overflow (Aug 2010)

In August 2010, Block 74638 was discovered to contain a transaction that created over 180 billion (or twice times INT64_MAX) bitcoins for three different recipients[2]. This occurred because the Bitcoin Core code did not account for checking transactions that had combined outputs so large that they overflowed.

The bug was posted about in a forum on bitcointalk.org and a fix was quickly released[3]. The blockchain was forked and the fixed chain overtook the bad chain 53 blocks later, thus the 180 billion bitcoins created by the bug no longer exist.



Never-Confirming Transactions (Sep 2010)

In September 2010, a user posted on bitcointalk.org about having none of their sent transactions being confirmed[4]. The user had been sending out payments smaller than the transaction fee, which then were never confirmed. The change leftover was also then never confirmed but was combined to create larger transactions which would never get confirmed. These transactions would then contaminate the wallets of their recipients.

The problem was solved by adding in checks to make sure that coins would only be selected for combining if they had at least 1 confirmation. The contaminated transactions were eventually removed by those who had originated them.

Mt. Gox – Transaction Malleability (Feb 2014)

In February 2014, a large coordinated attack was made on multiple Bitcoin Exchanges, with Mt. Gox suffering the most due to having used a custom client and an automated system that approved withdrawals before thoroughly checking them[5]. The exploited weakness was Transaction Malleability, in which an attacker changes the unique ID of a transaction before it is confirmed, thus making it seem like a transaction had not happened and allowing bitcoins to be spent twice. This weakness occurs because the formats of transaction signatures are not always properly checked, allowing an attacker to create different hashes for the same transaction[6].

This vulnerability was known to the Bitcoin community since 2011—the Bitcoin client couldn't handle badly-formed signatures. The developers had also failed to ensure all 3rd party users would check the signatures when using the software. The problem was corrected in the 0.8.0 release with new methods in the Script Interpreter such as `IsCanonicalSignature()`.

Fixing Bugs in the Core

Bitcoin is open-source but is maintained mainly by a small team of developers, led by Gavin Andresen[7]. The developers communicate through chats and mailing lists. Anyone can submit a pull request to modify the Core, but only a select few developers can merge code into the Core after a consensus is reached amongst the miners. Users can also submit Bitcoin Improvement Protocols (BIPs), which are requests for changes. Consensus occurs when a majority of miners use a particular branch of the code, causing all miners to jump onto that branch and make it official.

The open-source nature of Bitcoin does have its weaknesses. It can be slow for miners to reach consensus on a modification. Individuals may also agree to changes just because they see others doing so, or may remain silent on issues if doing so would benefit themselves. Changing to a more closed-source system would mean more focused, faster development, but it would also concentrate power in the hands of the developers and reduce the appeal that brings a lot of users into Bitcoin in the first place.

Citations

1. StackExchange Bitcoin, “What is the ‘1 RETURN’ bug?.” Jun 17, 2015. http://bitcoin.stackexchange.com/questions/38037/what-is-the-1-return-bug

2. BitcoinWiki, “Value overflow incident.” Aug 17, 2015. https://en.bitcoin.it/wiki/Value_overflow_incident

3. Bitcoin Forum, “Topic 822: Strange block 74638.” Aug 18, 2010. https://bitcointalk.org/index.php?topic=822.0

4. Bitcoin Forum, “Topic 1306: I broke my wallet, sends never confirm now.” Sep 29, 2010. https://bitcointalk.org/index.php?topic=1306.0

5. CoinDesk, “Bitcoin Exchanges Under ‘Massive and Concerted Attack.’” Feb 11, 2014. http://www.coindesk.com/massive-concerted-attack-launched-bitcoin-exchanges/

6. CoinDesk, “What the ‘Bitcoin Bug’ Means: A Guide to Transaction Malleability.” Feb 12, 2014. http://www.coindesk.com/bitcoin-bug-guide-transaction-malleability/

7. Motherboard, “Who’s Building Bitcoin? An Inside Look at Bitcoin’s Open Source Development.” May 7, 2013. http://motherboard.vice.com/blog/whos-building-bitcoin-an-inside-look-at-bitcoins-open-source-development