Using CLTV Time-Locked, 2/2 Multi-Signature P2SH Bitcoin Contracts in Practice

Justin Smith
Swisscom Blockchain AG
XMR Systems LLC
Zürich, Switzerland
Contact: http://www.jsmith.website

***Abstract*—Traditional financial institutions seeking to offer professional cryptoasset custody or storage services can use the Bitcoin Script programming language to implement well-established security procedures and best practices to protect against sophisticated threats with minimal impact to accessibility of funds. Requiring multiple endorsements for a spend transaction can deter network and hardware attacks, while preventing funds from being spent until a certain amount of time has been reached may deter threats of physical violence or coercion.**

In this paper a manual workflow of creating, funding, and spending a simple CHECKLOCKTIMEVERIFY time-locked, Pay-to-Script-Hash (P2SH) contract requiring two of two possible signatures using off-the-shelf computer hardware and open source software is demonstrated, and security, privacy, and cost concerns are discussed.

I. Introduction

Bitcoin transactions are electronic contracts, where cryptographic signatures are created and validated using common computer hardware. Bitcoin contracts are validated and enforced by the Bitcoin network. A person which knows the passphrase used to encrypt the instance of the bitcoin software application running on the *primary-signing device* is called the *primary-signer*. A person who knows the passphrase used to encrypt the instance of the bitcoin software application running on the *co-signing device* is called the *co-signer*. The co-signer and primary-signer together are called *participants*. The action of a person entering a valid passphrase to decrypt a private key on a given machine is called *signing*.

II. Motivation

A common mistake financial institutions and enterprises make when approaching cryptofinance system architecture is to substitute a multi-*approval* as illustrated in figure [Appendix, Fig.1] design for a multi-*signature* design as illustrated in figure [Appendix, Fig.2]. Some early multi-approval systems have been designed to allow remote authorization of a Pay-to-Public-Key Hash (P2PKH) or Pay to Script Hash (P2SH) single signature transaction via an API accessible over an open network. This would mean an attacker would only need to take control of the enterprise server to steal all of the cryptoassets protected by the dedicated "key storage" machine. A multi-signature system by contrast requires approvers to also be signers, and to apply signatures to a P2SH transaction in a round-robin format before that transaction can be submitted to the network and validated. An attacker would have to have access to the hardware devices of all participants, and somehow learn each of their passphrases. A time-lock encumbrance on Unspent Transaction Outputs (UTXOs) could make this process even harder to attack, as any unexpected "withdrawal" attempts would be immediately detected and automatically declined.

A. Hardware Required

A laptop with a network connection is used as the *co-signing device*. A separate, dedicated desktop machine or Single Board Computer (SBC) with a network connection is used as the *primary signing device*.

B. Software Required

Download and verify the source of Bitcoin Core [10], build the latest stable version on both the *co-signing device* and *primary-signing device*. Building from source mitigates the risk of accidentally installing malicious pre-compiled binaries [18] [19].

Magic-wormhole [9] is installed on both machines and port 4000 is opened to allow the magic-wormhole program to access the default magic-wormhole rendezvous server.

Libbitcoin explorer [5] is used for a variety of transaction creation and encoding operations, and is called with *bx*.

C. Extracting Co-signer Public Key

The first step in creating a multi-signature contract is for the *co-signer* to share their public key with the *primary-signer*. The method by which the public key is shared is important; it should ensure the key cannot be viewed by a third party, and it should not encourage behavior which can be exploited by an attacker, such as swapping USB drives [26]. The relay software magic-wormhole is used here, with security and privacy issues detailed in the discussion section below. Other methods of transmitting data, such as optically or by radio, are possible.

The co-signer creates a private key from a random seed on the co-signing device, then derives the public key.

```
$ bx seed | bx ec-new
46d170451e8565c9b5233ca160ba1aee9680
f24059fa565af17ef11f80cc7d43
```

```
$ bx ec-to-public 46d170451e8565c9b5
233ca160ba1aee9680f24059fa565af17ef1
1f80cc7d43
02bceab65ad6b5e3c1d6683b1d0ffd7bce57
2ce3ece0f1976fb04ba34bb0359c1b
```

D. Transmission of Co-signer Public Key to Primary Signing Device

The co-signer securely transmits public key generated on the co-signing device to the *primary-signer* using magic-wormhole. On the co-signing device:

```
$ wormhole send --text 02bceab65ad6b
5e3c1d6683b1d0ffd7bce572ce3ece0f1976
fb04ba34bb0359c1b
On the other computer, please run:
wormhole receive 9-corporate-gremlin
```

Immediately after the co-signer enters the command to start magic-wormhole on the co-signing device, the primary-signer enters the following command on the primary signing device:

```
$ wormhole receive 9-corporate-greml
in
02bceab65ad6b5e3c1d6683b1d0ffd7bce57
2ce3ece0f1976fb04ba34bb0359c1b
```

The primary-signer should verify the public key received from the co-signer by verbally repeating the first four and last four characters of the public key received back to the co-signer. On the primary signing device, the primary-signer will create a public key using the same procedure.

The two public keys are written to the primary signing device's local disk.

- Co-Signer Public Key: `02bceab65ad6b5e3c1d6683b1d0ffd7bce57 2ce3ece0f1976fb04ba34bb0359c1b`
- Primary-Signer Public Key: `02c4916c1c044bda56670f28efdb6637fa24 448f5184839f917a473abfd81e6d3d`

E. Specifying a Lock Time

The lock time, or date before which the UTXOs may not be re-spent, can be specified as Unix time or block height. Values greater than 500'000'000 are interpreted as Unix timestamps, values equal to or below that threshold are interpreted as block height. To obtain the current time on a Unix terminal:

```
$ date +%s
1552996392
```

The timestamp (1552996392) or 03/19/2019 @ 11:53am (UTC) is used as the current time. For this particular workflow, the timestamp (1553002337) for 03/19/2019 @ 1:32pm (UTC) is used as the "lock UTXO until" time.

F. Translation of the Lock Time Integer

The integer time-lock 1553002337 is translated to hexadecimal format using Christopher Allen's bash scripts [1].

```
$ printf '%08x\n' 1553002337 | sed '
s/^|(00|)*//'
5c90ef61
```

Bitcoin uses Little-Endian format to optimize CPU performance on most machines.

```
$echo 5c90ef61 | grep -o .. | tac |
echo $(tr -d '\n')
61ef905c
```

G. Creating the Assembly Script

The *Assembly Script* specifies the locking conditions. The receiving address is created on the primary signing device from the assembly script. The -v 196 parameter specifies a testnet address.

```
$ bx script-to-address -v 196 "[61ef
905c] checklocktimeverify drop 2 [02
bceab65ad6b5e3c1d6683b1d0ffd7bce572c
e3ece0f1976fb04ba34bb0359c1b][02c491
6c1c044bda56670f28efdb6637fa24448f51
84839f917a473abfd81e6d3d] 2 checkmul
tisig"
2N78HgpbsE9YihQUp2mAeiQv5dPjD1GPc5W
```

A funding transaction sending 1,000,000 testnet Satoshis (0.01 Bitcoin) to the Locking Script address via the *bitcoin-cli*, with Base16 hash `b7b04278cafbd7335a7ee3941d 8bc30c884f9e5ff1ae703c8291b4b324e49b0f`. The transaction hash is also called the *Transaction ID, or TXID*.

H. Encoding the Spend Transaction

From the 1'000'000 available, 800'000 Satoshis (0.008 BTC) will be sent to a new address, `2NDqirycbrTGHG YCPRA8gALAcwgWfMiEMBS`. In a banking use-case, this new address might be specified by a client as a withdrawal destination. The remaining 200'000 Satoshis will be kept by the miner who includes this transaction in a mined block.

The funding transaction hash and unspent transaction output's (UTXO) position of 0 are specified.

```
$ bitcoin-cli -named createrawtransa
ction inputs=
'''[  "txid": "'b7b04278cafbd7335a7
ee3941d8bc30c884f9e5ff1ae703c8291b4b3
24e49b0f'", "vout": '0'  ]''' output
s=''' "'2NDqirycbrTGHGYCPRA
8gALAcwgWfMiEMBS'": 0.008 '''
locktime=1553004188
02000000010f9be424b3b491823c70aef15f
9e4f880cc38b1d94e37e5a33d7fbca7842b0
b70000000000feffffff0100350c00000000
0017a914e1e8c2dd74e80d01f56a232fb074
8ee2905b3211879cf6905c
```

The resulting encoded transaction is now transmitted from the primary signing device to the co-signing device using the magic-wormhole technique.

I. Signing the Spend Transaction

On the co-signing device, the co-signer will create the *co-signer endorsement* using the co-signing device private key, the assembly script, and the encoded transaction. This yields the first endorsement:

```
$ bx input-sign 46d170451e8565c9b523
3ca160ba1aee9680f24059fa565af17ef11f
80cc7d43 "[61ef905c] checklocktimeve
rify drop 2 [02bceab65ad6b5e3c1d6683
b1d0ffd7bce572ce3ece0f1976fb04ba34bb
0359c1b] [02c4916c1c044bda56670f28ef
db6637fa24448f5184839f917a473abfd81e
6d3d] 2 checkmultisig" 02000000010f9
be424b3b491823c70aef15f9e4f880cc38b1
d94e37e5a33d7fbca7842b0b70000000000f
efffffff0100350c000000000017a914e1e8c
2dd74e80d01f56a232fb0748ee2905b32118
79cf6905c
3045022100abe6fa9c720fb13b13d87aedc6
35c9be15e9fa22498f7f615220d10160cca8
a00220769913c7c7b43a6e98b574eea90478
7894600cf88a7ec0ad3cb12d66fb6ac2dc01
```

The *co-signer endorsement* is transferred back to the primary signing device, again using magic-wormhole, and written to disk. The primary device now creates the *primary-signer endorsement* using the same command, but with the primary signing device private key `53064d99236c9e5fe30d2fe2dc3a5f6cd2374cac3f5c997a327218381d41c7d7`. This yields the second endorsement:

```
304402203ca822083273dce4cb989fd12e12
54fe221cf418ff67eafa058f04a25934b3bb
022065ecd41688668ace8e35d2b4453da234
285ccb5b7769fdcd1b9edf7d3d137fc701
```

The primary-signer encodes the assembly script.

```
$ bx script-encode "[61ef905c] check
locktimeverify drop 2 [02bceab65ad6b
5e3c1d6683b1d0ffd7bce572ce3ece0f1976
fb04ba34bb0359c1b] [02c4916c1c044bda
56670f28efdb6637fa24448f5184839f917a
473abfd81e6d3d] 2 checkmultisig"
0461ef905cb175522102bceab65ad6b5e3c1
d6683b1d0ffd7bce572ce3ece0f1976fb04b
a34bb0359c1b2102c4916c1c044bda56670f
28efdb6637fa24448f5184839f917a473abf
d81e6d3d52ae
```

The primary-signer combines the co-signer and primary-signer endorsements, the assembly script, and the encoded transaction.

```
$ bx input-set "zero [3045022100abe6
fa9c720fb13b13d87aedc635c9be15e9fa22
498f7f615220d10160cca8a00220769913c7
c7b43a6e98b574eea904787894600cf88a7e
c0ad3cb12d66fb6ac2dc01][304402203ca8
22083273dce4cb989fd12e1254fe221cf418
ff67eafa058f04a25934b3bb022065ecd416
88668ace8e35d2b4453da234285ccb5b7769
fdcd1b9edf7d3d137fc701][0461ef905cb1
75522102bceab65ad6b5e3c1d6683b1d0ffd
7bce572ce3ece0f1976fb04ba34bb0359c1b
2102c4916c1c044bda56670f28efdb6637fa
24448f5184839f917a473abfd81e6d3d52ae
]" 02000000010f9be424b3b491823c70aef
15f9e4f880cc38b1d94e37e5a33d7fbca784
2b0b70000000000feffffff0100350c00000
0000017a914e1e8c2dd74e80d01f56a232fb
0748ee2905b3211879cf6905c
02000000010f9be...f6905c
```

The resulting <transaction_hex> has a size of 309 bytes. The size of the script is an important consideration, as larger scripts must pay a proportionally larger transaction fee to ensure miners are incentivized to include this larger transaction in the next block. If the transaction is broadcast to the network before the locktime expires, the bitcoin-cli software will give the following error:

```
non-final(code64)
```

It is possible to broadcast this transaction into the mempool, but the UTXOs cannot be re-spent until the locktime has expired. After locktime expiration, the UTXO may be spent. This transaction was broadcast the next day, with Unix timestamp 1553075595 (03/20/2019 @ 9:53am UTC):

```
$ bitcoin-cli sendrawtransaction <tr
ansaction_hex>
bae3cccc3e430a3265dbee63e0a06c128910
9d1ff73bb103334fea85f434139e
```

III. Discussion

Compiling and testing Bitcoin Script Assembly language requires specialized tools and knowledge. For enterprise use, creating a compiler for Bitcoin Script using a well-known library such as Crypto++ [7] or Project Nayuki [8] would provide confidence that the cryptography and random number generation techniques are sound, integers are translated properly, and transactions are encoded properly. These considerations are important in the context of topics such as the "2038 problem" [21]. Libbitcoin explorer (bx) is useful for quickly creating and testing prototype contracts on Bitcoin's testnet, but would unexpectedly encode a different receiving address from what was specified in the command when encoding transactions. An interesting but untested (by the author) alternative to bx is BTC-Suite for Go [27].

A. Attack Vectors

The random number generation process can be attacked on each machine which contributes an endorsement. This is due to the risk the hardware device or processor architecture may be compromised [28]. But by diversifying or randomizing the device hardware which contribute endorsements and requiring multiple such devices, exploits for any given hardware, processor, or operating system can not affect the overall security of the system and funds cannot be stolen. An attacker would have to 1) possess exploits for all of the different hardware which is used to create endorsements 2) be present at the time all required keys are generated. Using a variety of mobile devices, desktops, workstations, and dedicated machines can help ensure such attacks are infeasible. In practice this might mean the co-signing device is an 32 or 64 bit ARM or RISC-V processor, while the primary signing device is an X86 processor.

B. Confidentiality

Unsolicited UTXOs sent to the script's receiving address can be used for surveillance or for frustrating surveillance attemps [22]. These small amounts should be taken into consideration when developing policies which relate to creating spend transaction policies. Redeem scripts and public keys should be kept private. If any of the endorsers' public keys become known, they could be used against the signers in a common-input heuristic attack [23]. To help mitigate this attack, new public keys should be used for every transaction. Loss of confidentiality is a key concern during the round-robin signing process. In this paper magic-wormhole's default server was used to transport data; this exposes the participants' IP addresses and other metadata. To maintain confidentiality a private magic-wormhole rendezvous server could be used over Tor.

C. Cost

Multisignature transactions using ECDSA signatures are presently created using scripts, which are larger in size (bytes) leading to a higher Satoshi-per-byte cost. Re-writing the assembly code for Pay to Witness Script Hash (P2WSH) [24] or using Schnorr signatures in a e.g. MuSig scheme [13] [14] could lead to a smaller transaction size, and therefore lower transaction cost and perhaps better privacy. However, such cryptographic signatures schemes are new and could be risky to use in production systems [29].

D. Recovery

If an endorsement script is lost before being broadcast, it can be re-created by following the same signing procedure. This means that all of the required signing devices must be properly backed up. If one of the devices cannot be recovered in a 2/2 transaction, all funds which are locked will be lost. Adding at least one "recovery signer" and using a scheme such as 2/3 or 3/5 would reduce the probability of total loss of funds, without necessarily increasing the probability of theft through collusion of the signers.

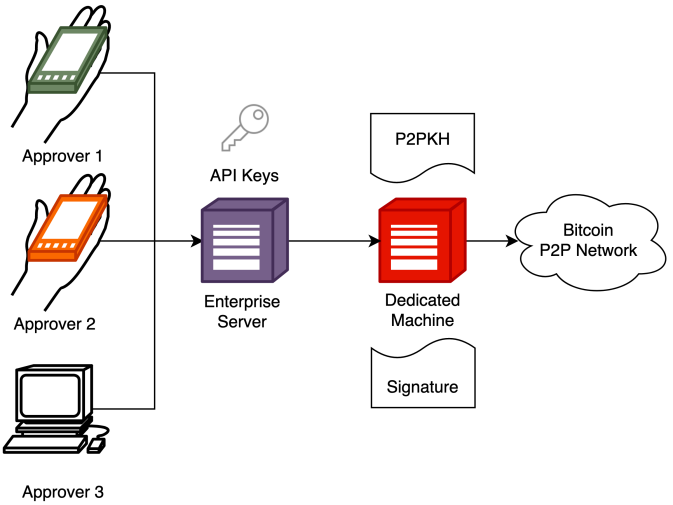

Fig. 1. Single signature architecture; anyone with access to the enterprise server can trigger outgoing transactions.

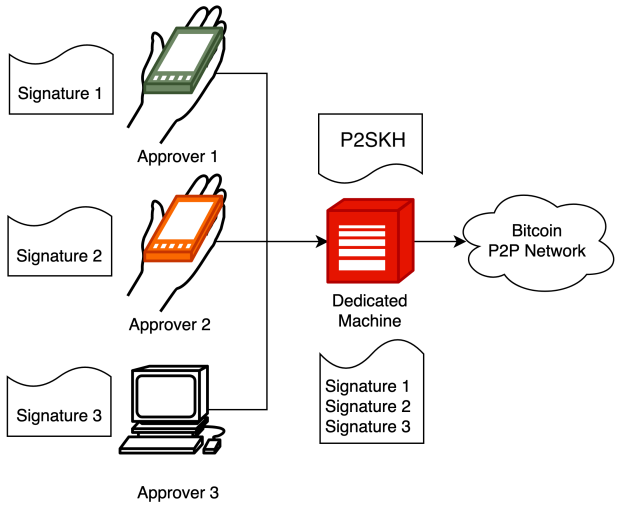

Fig. 2. Multi-signature architecture means an attacker must compromise a minimum number of approvers. Adding a time constraint would make a successful attack even more unlikely.

Acknowledgment

The author would like to thank Dr. Christian Decker for his review of this paper and suggestions for improvement.

REFERENCES

[1] C. Allen, *Building the Structure of P2SH*, accessed 5-Mar-2019. https://github.com/ChristopherA

[2] The Bitcoin Wiki. https://en.bitcoin.it/wiki/Transaction

[3] A. Jaramillo, *Libbitcoin-Explorer: Bitcoin Transaction via Command Line*, accessed 4-Apr-2017. http://aaronjaramillo.org/libbitcoin-explorer-bitcoin-transaction-via-command-line

[4] *Learn me a Bitcoin website*, accessed 11-Mar-2019. http://learnmeabitcoin.com/glossary/locktime

[5] *How to Spend From a Multisig Address*, accessed 11-Mar-2019. https://github.com/libbitcoin/libbitcoin-explorer/wiki/How-to-Spend-From-a-Multisig-Address

[6] *Coin.bin website*, accessed 11-Mar-2019. https://coinb.in/#verify

[7] W. Dai, *Crypto++ Library 8.1*, last modified 22-Feb-2019. https://cryptopp.com/

[8] *Project Nayuki*, accessed 11-Mar-2019. https://www.nayuki.io/page/bitcoin-cryptography-library

[9] *Magic Wormhole Application*, accessed 11-Mar-2019. https://github.com/warner/magic-wormhole

[10] *Bitcoin Core Repository*, accessed 11-Mar-2019. https://github.com/bitcoin/bitcoin/

[11] G. Maxwell, *Offline Signing*, accessed 11-Mar-2019. https://people.xiph.org/ greg/signdemo.txt

[12] M. Drijvers, *On the Security of Two-Round Multi-Signatures*, accessed 11-Mar-2019. https://eprint.iacr.org/2018/417.pdf

[13] *MuSig for multiparty signatures*, Bitcoin OpTech Newsletter. https://bitcoinops.org/en/newsletters/2019/02/26/

[14] G. Maxwell and A. Poelstra and Y. Seurin and P. Wuille, *Simple Schnorr Multi-Signatures with Applications to Bitcoin*, Cryptology ePrint Archive, Report 2018/068. https://eprint.iacr.org/2018/068

[15] *Bitcoin Transaction Check Webapp*. https://blockchainprogramming.azurewebsites.net/checktx

[16] *BIP65: OP_CHECKLOCKTIMEVERIFY*. https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki

[17] *Bitcoin Address Types*. https://en.bitcoin.it/wiki/List_of_address_prefixes

[18] D. Goodin, *Widely used open source software contained bitcoin-stealing backdoor*. https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/. Arstechnica, 26-Nov-2018.

[19] C. Cimpanu, *Node.js Package Manager Vulnerable to Malicious Worm Packages*. https://news.softpedia.com/news/node-js-package-manager-vulnerable-to-malicious-worm-packages-502216.shtml. Arstechnica, 26-Mar-2016.

[20] A. Antonopoulos, *Mastering Bitcoin*. https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc#a-simple-script

[21] *2038 Problem*. https://en.wikipedia.org/wiki/Year_2038_problem

[22] *Crypto Dusting is a New Type of Blockchain Spam that Corrodes Reputations and Impacts Cryptocurrency AML*, ciphertrace.com website. https://ciphertrace.com/crypto_dusting/

[23] *Receiving donations spied on with mystery shopper payments*. https://en.bitcoin.it/wiki/Privacy

[24] L. Johnson and P. Wuille, *BIP143: Transaction Signature Verification for Version 0 Witness Program*. https://github.com/bitcoin/bips/blob/master/bip-0143

[25] *How To Create Time-locked Transactions with Bitcoin*. https://steemit.com/bitcoin/@daan/how-to-create-time-locked-transactions-with-bitcoin-free-bitcoins-inside

[26] *Stuxnet*. https://en.wikipedia.org/wiki/Stuxnet

[27] *Bitcoin in Go*. https://github.com/btcsuite

[28] L. Newman, *Spectre-Like Flaw Undermines Intel Processors' Most Secure Element*. https://www.wired.com/story/foreshadow-intel-secure-enclave-vulnerability/

[29] M. Drijvers and K. Edalatnejad and B. Ford and E. Kiltz and J. Loss and G. Neven and I. Stepanovs, *On the Security of Two-Round Multi-Signatures*. https://eprint.iacr.org/2018/417