nChain
Data without question

PUSHTX and its Building Blocks

White Paper

WP1605

Created: 20/05/2021
Last updated: 14/12/2021

Copyright

Disclaimer

Authors

Name	Title
Wei Zhang	Principal Researcher

Contents

1 PUSHTX and Its Building Blocks

The core idea of PUSHTX (originally invented by Y. Chan and D. Kramer at nChain in 2017) is to generate a signature in-script on a data element on the stack and call OP_CHECKSIG to verify the signature. If it passes, it implies that the message constructed by OP_CHECKSIG is identical to the data element pushed to the stack. Therefore, it achieves the effect of pushing the current spending transaction to the stack. Since its adoption by sCrypt, STAS token and Sensible Contracts, the PUSHTX idea has been widely discussed and tested in practice. The purpose of this document is to offer some security insight and optimisations while revisiting PUSHTX with an example of a perpetually enforcing locking script. We focus on two most important building blocks of PUSHTX, the signature generation and the message construction. The size of its spending transaction is intended to be a benchmark for reference. Other techniques of pushing transactions to the stack are out of scope of this white paper.

A perpetually enforcing locking script (PELS) is a locking script that enforces some condition or conditions on all future transactions in the spending chain that originates from the output that contains this locking script. One example to achieve this is to design a locking script that forces the locking script in the spending transaction to be the same as itself. Note that a locking script with an enforcement only on the next spending transaction would have a much simpler design. PELS are particularly useful for the sender (originator) as they can be ensured that all future spending transactions will follow the rules which they set out in the locking script. Any violation of the rules would invalidate the transaction in terms of script execution.

1.1 Generating the signature in-script

The first building block is to generate the signature for a given message m. We assume that the following script segment is part of a locking script, and the input data can be either in an unlocking script or hard coded in the same locking script.

```
[sign]:= OP_HASH256 k^-1 OP_MUL k^-1*ra OP_ADD n OP_MOD r [toDER]
SIGHASH_FLAG OP_CAT
```

```
Input data: m
```

Remarks

1. The equation for computing s in the ECDSA signature is $k^{-1}(z + ra)\ mod\ n$, where z is the double SHA256 of the message m to be signed. We write the equation in script as $(k^{-1}z + k^{-1}ra)\ mod\ n$ to indicate that k^{-1} and $k^{-1}ra$ can be precomputed. It would be very costly to compute modular inverse $k^{-1}\ mod\ n$ in script. As we are not using the signature for authenticity, the private key a and the ephemeral key k can be chosen at wish and shown publicly.
2. The script segment `[sign]` as part of the locking script needs to fix both the ephemeral key k and the private key a. Although anyone can generate a valid signature using `[sign]`, the focus is on the input m. The requirement is that there is only one value of m that can pass `OP_CHECKSIG` for any given spending transaction. If the private key or the public key is not fixed, then the transaction will be malleable. The detail can be found in Section 2. If the ephemeral key k is not fixed, then anyone can

use a different k to create a valid transaction with a different transaction ID, which is not desirable in some use cases.

3. To further optimise the script segment, one can choose small values for k and a such as 1, and they can be the same every time. Note that if $k = a = 1$, then $s = z + G_x \ mod \ n$, where G_x is the x-coordinate of the generator point G. The compressed public key will be G_x too. The definition of `[sign]` can be re-written as

```
[sign]:= OP_HASH256 G_x OP_ADD n OP_MOD G_x [toDER]
SIGHASH_FLAG OP_CAT
```

4. The script segment `[toDER]` is to convert the pair (r, s) to the canonical DER format. This is the only format accepted by `OP_CHECKSIG`. It forces s to be in the range between 0 and $n/2$ to avoid transaction ID malleability. Although this is a policy rule in the Bitcoin network, it seems that Bitcoin nodes are unlikely to accept alternatives.
5. Note that `SIGHASH_FLAG` in `[sign]` is compulsory as `OP_CHECKSIG` expects it. It can be used to specify which part of the spending transaction should be pushed to the stack. For example, the flag ALL would require all the inputs and outputs to be included in the message m, while `SINGLE|ANYONECANPAY` would require the input corresponding to this locking script and its paired output to be included in m.
6. If we extend the script segment to "`OP_DUP [sign]` $< PK >$" with input m, the stack from bottom to top will look like `[`$m, \ Sig, \ PK$`]` after its execution. A call to `OP_CHECKSIGVERIFY` will consume the signature and the public key, leaving m on the top of the stack. If the verification is successful, then one can be convinced that the message m left on the stack is an accurate representation of the spending transaction.

1.2 Constructing the message in-script

The signed message in its serialised format is different from the serialised transaction. The latter gives away all the information about the transaction, while the signed message unintentionally conceals some information about the transaction in hash values and offers some information about the output being spent, i.e., its value and its locking script.

The message m cannot be fully embedded in the locking script as it contains the locking script itself and some unknown information on the future spending transaction. Only some of the fields can be explicitly enforced in the locking script, e.g., version, sequence number, or locktime. The message m is either provided in the unlocking script in its entirety or constructed in script with some inputs from the unlocking script and instructions from the locking script. We will focus on the latter as it is more restrictive from the perspective of a spending transaction. Note that the goal of constructing the message is to enforce desired values for some data fields in the spending transaction. The table below captures all the data fields in the message and whether they should or can be fixed in the locking script. "Optional" is given if it is use-case specific.

Table 1 - Components of a signed message

	Items	Fixed explicitly in locking script or not
1	Version 4 bytes little endian	Optional
2	Hash of input outpoints 32 bytes	Infeasible due to circular reference of TxID
3	Hash of input sequence numbers 32 bytes	Optional, recommend “Not” to allow more flexibility in spending transaction
4	Input outpoint 32 bytes + 4 bytes in little endian	Infeasible due to circular reference of TxID (although 4 bytes index can be optional)
5	Length of previous locking script	Optional, recommend “Not” for simplicity
6	Previous locking script	Infeasible due to circular reference of the locking script
7	Value of previous locking script 8 bytes (little endian)	Optional
8	Sequence number 4 bytes (little endian)	Optional
9	Hash of outputs 32 bytes	Optional if it is known before hand, otherwise, infeasible to be fixed.
10	Locktime 4 bytes in little endian	Optional
11	Sighash flag 4 bytes in little endian	Recommend being fixed for more restrictiveness

From now on, the data fields in the table will be referred as item 1, 2, 3, etc.

When it is optional, whether to provide the data in the locking or unlocking script depends on use cases. A general rule is that if the data is available or known at the time of creating the locking script, then they can be included in the locking script. Another aspect to consider is the size of the transaction and its spending transaction. By shifting the data between the locking and unlocking script, one can shift some of the transaction fee cost between the senders of the two transactions.

Note that when we say infeasible due to circular references, the granularity is set at date fields. For example, partial locking script or even a small part of a transaction ID (e.g., fixing the first two bytes of a 32-byte transaction ID and allow iterations through some fields in a serialised transaction) can be fixed in the locking script if required.

As mentioned earlier, although the focus is to construct the message m, the goal is to use m to enforce values on different fields in the spending transaction. To enforce the data behind the hash values, i.e., item 9, the locking script should be designed to request the pre-image, hash

them in-script, and then construct the message to be signed in-script. Taking item 9 as an example, to enforce the outputs in the current transaction, we can have

```
[outputsRequest]:= OP_DUP OP_HASH256 OP_ROT OP_SWAP OP_CAT <item 10 and
11> OP_CAT

Input data: <item 1 to 8> <serialised outputs in current transaction>
```

Remarks

1. The script segment `[outputsRequest]` takes item 1 to 8 and the serialised outputs on the stack to construct item 9, and concatenate with item 10 and 11 to obtain the message m in-script. By calling `[sign]` `<`G_x`>` `OP_CHECKSIGVERIFY` after `[outputsRequest]` and passing the verification, one can be convinced that the serialised outputs left on the top of the stack is a true representation of the outputs in the current transaction.
2. It is also very useful to leave a copy of `<item 1 to 7>` on the stack for comparison. This can be achieved by modifying the script segment as below.

   ```
   [outputsRequest]:= OP_2DUP OP_HASH256 OP_SWAP <item 8> OP_CAT
   OP_SWAP OP_CAT <item 10 and 11> OP_CAT
   ```

   ```
   Input data: <item 1 to 7> <serialised outputs in current
   transaction>
   ```

   After executing the modified `[outputsRequest]` on the input data, we can call `[sign]` `<`G_x`>` `OP_CHECKSIGVERIFY` to consume the message. The stack will have the current serialised outputs on the top followed by `<item 1 to 7>`. We will use the modified script segment later to enforce the output in a spending transaction.
3. It is simpler if consecutive items are grouped together as in `<item 1 to 7>`. They are either all in an unlocking script or all fixed in a locking script. However, a more granular approach is available at a potential cost of having a more complex script.

Note that the serialisation format for current outputs is

a. value of the output 8 bytes (little endian),
b. length of the locking script,
c. the locking script, and
d. concatenate serialised outputs in order if there is more than one output.

The serialisation format for previous output (item 5 to 7) in a signed message is

a. length of the locking script,
b. the locking script, and
c. value of the output 8 bytes (little endian).

In the following example, we will compare the previous output with the output in the current spending transaction and force them to be identical. The two formats will be useful for designing the locking script for the comparison.

1.3 Example of a PELS

Suppose that Alice is a root Certificate Authority (CA) and Bob is a subordinate CA. Alice is going to delegate some work to Bob which would require Bob to publish transactions on-chain as attestations to certificates. Alice does not want Bob to spend the output on anything else. Therefore, Alice is going to force all the subsequent spending transactions to have a fixed `[P2PKH Bob]` locking script and a fixed output value. Bob can spend the output as he can generate valid signatures, but he cannot choose any output other than sending the same amount to himself. For illustration purpose and simplicity, we ignore OP_RETURN payload throughout the example.

Alice constructs the initial transaction as shown below.

<table>
<tr><th colspan="5">$TxID_0$</th></tr>
<tr><td>Version</td><td colspan="2">1</td><td>Locktime</td><td>0</td></tr>
<tr><td>In-count</td><td colspan="2">1</td><td>Out-count</td><td>1</td></tr>
<tr><td colspan="3">Input list</td><td colspan="2">Output list</td></tr>
<tr><td>Outpoint</td><td>Unlocking script</td><td>nSeq</td><td>Value</td><td>Locking script</td></tr>
<tr><td>$Outpoint_A$</td><td>$< Sig_A >$
$< PK_A >$</td><td>FFFFFFFF</td><td>1000</td><td>[outputsRequest][sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 $<H(PK_B)>$ OP_EQUALVERIFY OP_CHECKSIG</td></tr>
</table>

Figure 1 - Initial transaction that contains a PELS output

The script segments are defined as:

```
[outputsRequest]:= OP_2DUP OP_HASH256 OP_SWAP <item 8> OP_CAT OP_SWAP
OP_CAT <item 10 and 11> OP_CAT
[sign]:= OP_HASH256 G_x OP_ADD n OP_MOD [toDER] SIGHASH_FLAG OP_CAT
G_compressed
[toDER]:= [toCanonical][concatenations]
[toCanonical][1]:= OP_DUP n/2 OP_GREATERTHAN OP_IF n OP_SWAP OP_SUB OP_ENDIF
[concatenations]:= OP_SIZE OP_DUP <0x24> OP_ADD <0x30> OP_SWAP OP_CAT
<0220||G_x||02> OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
```

Note that "`OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY`" is to extract the previous output from the signed message (verified and left on the top of the stack), then swap the position of the value field and the locking script to form the expected current output, and finally compare it with the actual current output provided in the unlocking script and its integrity is implied by the integrity of the signed message.

The length of the locking script is roughly

[1] This is to make sure that s is in the range between 0 and $n/2$. If $s > n/2$, we let $s = n - s$.

$(7 + 12) + (6 + 32 + 32 + 33) + (6 + 32 + 32) + (11) + (15 + 34) + (14 + 20)$
$= 286 = 0x011e.$

Note that these numbers are not meant to be precise, but they should be accurate enough to be in the right magnitude. For a more accurate result, please refer to Appendix.

To spend the transaction, Bob creates the spending transaction as below.

$TxID_1$				
Version	1		**Locktime**	0
In-count	1		**Out-count**	1
Input list			**Output list**	
Outpoint	**Unlocking script**	**nSeq**	**Value**	**Locking script**
$TxID_0 \|\| 0$	$< Sig_B >$ $< PK_B >$ $< Data_1 >$ $< Data_2 >$	FFFFFFFF	1000	[outputsRequest][sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG

Figure 2 - spending the PELS output

The unlocking script contains a data element $Data_1$ which represents items 1 to 7 and can be written as:

01000000 2268f59280bdb73a24aae224a0b30c1f60b8a386813d63214f86b98261a6b8763
bb13029ce7b1f559ef5e747fcac439f1455a2ec7c5f09b72290795e70665044 $TxID_0$ 00000
000 011e {[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT
OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP
OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG} e803000000000000

Table 2 - Data element $data1$ explained

	Items	**Value**
1	version	01000000
2	Hash of input outpoints	2268f59280bdb73a24aae224a0b30c1f60b8a386813d63214f86b98261a6b876
3	Hash of input sequence numbers	3bb13029ce7b1f559ef5e747fcac439f1455a2ec7c5f09b72290795e70665044
4	Input outpoint	$TxID_0$00000000
5	Length of previous locking script	011e

6	Previous locking script	{[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG}
7	Value of previous locking script	e803000000000000

The data element $Data_2$ represents the output in $TxID_1$ (value || locking script length || locking script) and can be written as:

```
e8030000000000000011e{[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP
<0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT
OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG}
```

The full script to be executed during the validation of $TxID_1$ is

$< Sig_B >< PK_B >< Data_1 >< Data_2 >$ [outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG

After the first OP_CHECKSIGVERIFY, we will have $< Sig_B > < PK_B > < Data_1 > < Data_2 >$ on the stack (rightmost on the top).

Table 3 - Stack execution

Step	The stack	To execute
1	$< Sig_B > < PK_B > < Data_1 > < Data_2 >$	OP_SWAP <0x68>
2	$< Sig_B > < PK_B > < Data_2 > < Data_1 >$ <0x68>	OP_SPLIT OP_NIP
3	$< Sig_B > < PK_B > < Data_2 >$ <011e {[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG} e803000000000000>	OP_SWAP OP_8 OP_SPLIT
4	$< Sig_B > < PK_B > < Data_2 >$ <011e {[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY	OP_SWAP OP_CAT

	OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG} e803000000000000> <e803000000000000> <011e {[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG}>	
5	$< Sig_B > < PK_B > < Data_2 >$ <011e {[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG} e803000000000000> <011e {[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG} e803000000000000>	OP_EQUALVERIFY
6	$< Sig_B > < PK_B >$	OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG
7	True	

The size of $TxID_1$ is
$version + locktime + input + output$
$= 4 + 4 + (36 + 72 + 33 + 104 + 287 + 8 + 287 + 8 + 4) + (8 + 287)$
$= 1142$ bytes.

Transaction Fee

Given the current setting, Bob can add his own input to cover the transaction fee. If Alice uses `SIGHASH_SINGLE|ANYONECANPAY` in the script segment `[sign]`, then Bob can add another output to collect changes. This effectively makes the enforcement from Alice's locking script perpetual. One can think of this as a node-enforced smart contract between Alice and Bob.

It is also possible for the locking script to take the transaction fee into consideration. After step 3, the top element on the stack is the value from the previous output. By adding `<`$TxFee$`>` `OP_SUB` before the concatenation in step 4, Alice allows Bob to pay the transaction fee from the previous output. This will lead to diminishing value of the output over spends, which can act as a desired feature as it sets the total number of spends Bob are entitled to.

1.4 Optimisation

As $Data_1$ contains $Data_2$, we can construct $Data_2$ from $Data_1$. In other words, we assume that the current output is identical to the previous output and use the previous output to construct the message. If it passes `OP_CHECKSIG`, then the two outputs must be identical. The script segment of `[outputsRequest]` can be re-written as:

```
[outputsRequest]:= OP_2DUP OP_CAT OP_TOALTSTACK OP_SWAP OP_CAT OP_HASH256
<item 8> OP_SWAP OP_CAT OP_FROMALTSTACK OP_SWAP OP_CAT OP_CAT <item 10
and 11> OP_CAT
```

`Input data: <item 1 to 4> <item 5 and 6> <item 7>`

With this new `[outputsRequest]`, we can update the locking script in $TxID_0$ and $TxID_1$ as:

```
[outputsRequest][sign] OP_CHECKSIGVERIFY OP_DUP OP_HASH160 <H(PK_B)>
OP_EQUALVERIFY OP_CHECKSIG
```

and the unlocking script as:

$< Sig_B > \ < PK_B > \ < Data_1 > \ < Data_2 > \ < Data_3 >$

where $Data_1$ is item 1 to 4:

`01000000`2268f59280bdb73a24aae224a0b30c1f60b8a386813d63214f86b98261a6b8763bb13029ce7b1f559ef5e747fcac439f1455a2ec7c5f09b72290795e70665044$TxID_0$`00000000`

$Data_2$ is item 5 and 6:

```
011b{[outputsRequest] [sign] OP_CHECKSIGVERIFY OP_SWAP <0x68> OP_SPLIT
OP_NIP OP_SWAP OP_8 OP_SPLIT OP_SWAP OP_CAT OP_EQUALVERIFY OP_DUP
OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG}
```

$Data_3$ is item 7: `e803000000000000`.

The size of $TxID_1$ is 941 bytes. A step-by-step execution is given below, where Step 1 to 5 is from `[outputsRequest]`.

Table 4 - Stack execution with optimisation

Step	The stacks	To execute
1	$< Sig_B >< PK_B >< Data_1 >< Data_2 >< Data_3 >$	`OP_2DUP OP_CAT OP_TOALTSTACK`

2	$< Sig_B > < PK_B > < Data_1 > < Data_2 > < Data_3 >$ ALTSTACK: <item 5 to 7>	OP_SWAP OP_CAT OP_HASH256
3	$< Sig_B > < PK_B > < Data_1 >$ <item 9> ALTSTACK: <item 5 to 7>	<item 8> OP_SWAP OP_CAT
4	$< Sig_B > < PK_B > < Data_1 >$ <item 8 and 9> ALTSTACK: <item 5 to 7>	OP_FROMALTSTACK OP_SWAP OP_CAT
5	$< Sig_B > < PK_B > < Data_1 >$ <item 5 to 9>	OP_CAT <item 10 and 11> OP_CAT
6	$< Sig_B > < PK_B >$ <item 1 to 11>	[sign] OP_CHECKSIGVERIFY
7	$< Sig_B > < PK_B >$	OP_DUP OP_HASH160 <H(PK_B)> OP_EQUALVERIFY OP_CHECKSIG
8	True	

Further improvement can be made by using the alt stack for storing G_x and n. Each of them is of size 32 bytes. As $G_{compress}$ is G_x and $\frac{n}{2}$ can be derived from n, we can use several opcodes to reference them from the alt stack. However, it would introduce complications when designing the script. For example, a comparison of two versions of `[sign]` is shown below.

Before:

```
[sign]:= OP_HASH256 G_x OP_ADD n OP_MOD [toDER] SIGHASH_FLAG OP_CAT G_x

[toDER]:= [toCanonical][concatenations]

[toCanonical]:= OP_DUP n/2 OP_GREATERTHAN OP_IF n OP_SWAP OP_SUB OP_ENDIF

[concatenations]:= OP_SIZE OP_DUP <0x24> OP_ADD <0x30> OP_SWAP OP_CAT
<0220||G_x> OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
```

After:

```
[sign]:= OP_HASH256 G_x OP_DUP OP_TOALTSTACK OP_ADD n OP_DUP OP_TOALTSTACK
OP_MOD [toDER] SIGHASH_FLAG OP_CAT OP_FROMALTSTACK

[toDER]:= [toCanonical][concatenations]

[toCanonical]:= OP_DUP OP_FROMALTSTACK OP_DUP OP_TOALTSTACK OP_2 OP_DIV
OP_GREATERTHAN OP_IF OP_FROMALTSTACK OP_SWAP OP_SUB OP_ENDIF

[concatenations]:= OP_SIZE OP_DUP <0x24> OP_ADD <0x30> OP_SWAP OP_CAT
<0220> OP_FROMALTSTACK OP_DUP OP_TOALTSTACK OP_CAT OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT
```

We added 15 opcodes and removed two instances of G_x and two instances of n. The total saving is $(32 \times 2 + 32 \times 2) - 15 = 113$ bytes. Therefore, the size of the spending transaction $TxID_1$ can be further reduced to 828 bytes.

It is safe to say that with all available optimisations, the size of a spending transaction can be around 1KB.

Note that any example scripts shown above are not meant to be used in mainnet transactions. They are written for illustration purpose. Some simplifications are implicitly assumed. For example, reversing endianness is omitted. For a more practical and comprehensive guide, please refer to Appendix.

2 Security Analysis

In this section, we offer some security proofs and analysis around PUSHTX. This is considered a formal approach to PUSHTX and some of the insights we described in the previous section.

First of all, we notice that there are effectively two signatures in the script execution. One is created by PUSHTX technique, the other one is provided in the unlocking script for P2PKH. The first signature provides data integrity and enables enforcement on the spending transaction, while the second provides authenticity and ensure that only the intended recipient can create the spending transaction[2].

Claim 1: If (r,s) is a valid ECDSA signature with respect to a public key P on the messages m, then it is computationally infeasible to construct m' such that $m' \neq m$ and (r,s) is still valid on m' with respect to P, assuming that the cryptographic hash function used is pre-image resistant and collision resistant.

Proof:

let $z = hash(m)$ and $z' = hash(m')$ for some m'.
Let $u = zs^{-1} \bmod n$, $u' = z's^{-1} \bmod n$, $v = rs^{-1} \bmod n$, and $R = kG$ where $R_x = r \bmod n$.
Also let R' denote the points such that $R' \neq R$ and $R'_x = r \bmod n$.
Note that[3] R' can either be $-R$ or one of the two points, $(x, \pm y)$,
where $x = R_x + n$ if $R_x < n$ and $x = R_x - n$ if $R_x > n$.
So, from the signature verification, we have $[uG + vP]_x = [u'G + vP]_x = r \bmod n$.
There are two cases to consider.
Case 1: $uG + vP = u'G + vP = R$

- $\Rightarrow$ $u = u' \bmod n$
- $\Rightarrow$ $z = z' \bmod n$
- $\Rightarrow$ $m = m'$ under the assumption that the hash function is collision resistant.

Case 2: $uG + vP = R$ and $u'G + vP = R'$

- $\Rightarrow$ $u'G = R' - vP$
- $\Rightarrow$ Given that the message m, the public P, and the value r are all fixed, there are maximum three different values that u' can take, each of which corresponds to a value of R'.

[2] In some use cases where authenticity is not required and it is desired to have anyone being able to create a spending transaction, the script segment P2PKH can be omitted. However, any implications and risks should be analysed and assessed before such approach.

[3] On secp256k1 curve, the group order n is less than the curve modulo p, but they are of the same bit length. Therefore, an equation $x = a \bmod n$ will have maximum two solutions for $x \in [0, p-1]$. The maximum number of solutions can only be achieved when $a \in [0, p-n)$.

⇒ $z' = u's$ also has maximum three values.

⇒ Finding m' such that $hash(m') = z'$ where z' can only be one of the three fixed values is computationally infeasible assuming that the hash function is pre-image resistant.

Therefore, we can conclude that it is computationally infeasible to find m' such that $m' \neq m$ and (r, s) is still valid on m' with respect to P.

Note that Claim 1 implies that PUSHTX technic is secure to use if we assume that double SHA256 is preimage resistant and collision resistant. Moreover, it implies that the data integrity is preserved even when we sacrifice authenticity in ECDSA by making both the private key and the ephemeral key available to the public.

Claim 2: Public key P must be fixed in the locking script.

Reasoning:

Suppose P is not fixed and (r, s) is a valid signature with respect to P on m.
Let $z' = hash(m')$. $u' = z's^{-1}$, and $v = rs^{-1}$
We want to find P' such that $u'G + vP' = R$
$P' = v^{-1}(R - u'G)$
Now (r, s) is valid with respect to P' on m'.
Therefore P must be fixed in the locking script.

Claim 3: k should be fixed in the locking script.

Reasoning:

Suppose (r, s) is a valid signature generated in the locking script with respect to P on m.
Suppose k is not fixed in the locking script and is provided in the unlocking script.
Then an adversary can:

1. intercept the spending transaction, and
2. replace k with k' in the unlocking script.

Then (r', s') generated in the locking script will be a valid signature with respect to P on m.
Transaction will still be valid, but the transaction ID is changed.

Claim 4: Sighash flag should be fixed in the locking script.

Reasoning:

Suppose (r, s) is a valid signature generated in the locking script with respect to P on m.
Suppose sighash flag is not fixed in the locking script and is provided in the unlocking script.

1. Intercept the spending transaction

2. Change the sighash flag
3. Update the message m accordingly to m'.

In some use case, this would invalidate the transaction. E.g., the locking script expects multiple inputs and outputs with sighash flag “ALL”; changing the flag to anything else would invalidate the script execution.
In others, this would change the transaction ID without invalidating the transaction. E.g., the locking script only enforces conditions on the outputs of its spending transaction; adding or removing “ANYONECANPAY” would not invalidate the transaction, but will change the transaction ID.

3 Appendix

All implementations have been tested on regtest on Bitcoin SV v1.0.8. Please be warned that the scripts below are for testing purpose only. Any attempt of using them on mainnet should undergo thorough testing and rigorous reviews.

The result shows that a spending transaction is of size 1415 bytes. The overhead is mainly coming from reversing endianness. A 32-byte string would require 124 bytes of opcodes to reverse its endianness. We need to reverse endianness twice in the locking script. The locking script appears both in the unlocking script and the output in the spending transaction. Therefore, the total overhead from endianness in our implementation is over 500 bytes. We did not use Alt Stack to store G_x and n in our current implementation for simplicity. This would save us about 200 bytes in total.

We present two example locking scripts first and explain what they do. We then present two transactions with one spending the other using the second locking script, demonstrating that the locking script works and can be spent as expected.

Locking Script 1 (LS1) – generateSig and checkSig

```
"aa517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f51
7f517f517f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7e7c7e7c7e7c7
e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2ddbfc9b0
2070b87ce9562a055acbbdcf97e66be799321414136d08c5ed2bf3ba048afe6dcaebafeff
ffffffffffffffffffffffffffff00977621414136d08c5ed2bf3ba048afe6dcaebafefff
ffffffffffffffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6dcaebafe
ffffffffffffffffffffffffffff007c946882766b6b517f517f517f517f517f517f517
f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f
517f517f517f517f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7
c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c
7e7c7e7c7e7c7e7c7e6c0120a0637c7e68827601249301307c7e23022079be667ef9dcbba
c55a06295ce870b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e01417e210279
be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ac"
```

Input: serialised transaction message for signing as in Section 1.2

The locking script takes the message m, and

1. double SHA256 on m to obtain z,
2. reverse endianness of z,
3. add 0x00 to ensure z is not interpreted as a negative number,
4. call `OP_BIN2NUM` to have minimal encoding on z (would take care the case when step 3 introduces redundancy),
5. compute $s = z + G_x \; mod \; n$,
6. convert s to $n - s$ if $s > n/2$,
7. obtain length of s,
8. reverse endianness of s (32 bytes),
9. reverse one more byte if the length of s is greater than 32,
10. compute the total length of a DER signature (`0x24` $+ \; length \; of \; s$),
11. add DER prefix `0x30`,
12. concatenate $r = G_x$,

13. concatenate s,
14. concatenate sighash flag "ALL",
15. push compressed public key G_x, and
16. call `OP_CHECKSIG`.

Note that in step 7 to 9, we have assumed that the length of s is either 32 or 33 bytes. However, s may be shorter than that. Some care should be taken to pad s using `OP_NUM2BIN` if it is too short.

Locking Script 2 (LS2) – constructMsg + LS1 + P2PKH

```
"6e810200029458807c7eaa04ffffffff7c7e7e7e7e0800000000410000007eaa517f517f
517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f5
17f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c
7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7
e7c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2ddbfc9b02070b87ce95
62a055acbbdcf97e66be799321414136d08c5ed2bf3ba048afe6dcaebafeffffffffffff
ffffffffffffffff00977621414136d08c5ed2bf3ba048afe6dcaebafeffffffffffff
ffffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6dcaebafeffffffffff
ffffffffffffffffff007c946882766b6b517f517f517f517f517f517f517f517f517f51
7f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517
f517f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7
c7e7c7e6c0120a0637c7e68827601249301307c7e23022079be667ef9dcbbac55a06295ce
870b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e01417e210279be667ef9dcb
bac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ad76a914751e76e8199196
d454941c45d1b3a323f1433bd688ac"
```

Input: `<Sig> <PK> <Item 1 to 4> <Item 5 and 6> <Item 7>`
The locking script takes a pair of signature and public key, and item 1 to 7 as in the table in Section 1.2 in three `PUSHDATA` operations, and

1. take the previous value and work out the new output value (subtracting a fixed transaction fee),
2. take the previous locking script as the new locking script for the new output,
3. concatenate the new output value and new locking script to obtain the new output,
4. double SHA256 the new output to obtain the hash of outputs (item 9),
5. push sequence number (item 8),
6. concatenate to get message string (item 1 to 9),
7. push locktime and sigahash flag (item 10 and 11),
8. concatenate to obtain the message to be signed m,
9. call LS1 with `OP_CHECKSIGVERIFY`, and
10. call P2PKH to check Sig with respect to PK.

Transaction 0 – genesis transaction

```
{
  "txid":
"88b9d41101a4c064b283f80ca73837d96f974bc3fbe931b35db7bca8370cca34",
  "hash":
"88b9d41101a4c064b283f80ca73837d96f974bc3fbe931b35db7bca8370cca34",
  "version": 1,
  "size": 730,
  "locktime": 0,
```

```
  "vin": [
    {
      "txid":
"52685bdbaae5c76887c23cee699bc48f293192a313c19b9fad4c77b993655df5",
      "vout": 0,
      "scriptSig": {
        "asm":
"3044022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
02201229c3605c61c4133b282cc30ece9e7d5c3693bf2cd1c03a3caadcd9f25900a5[ALL|
FORKID]
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798",
        "hex":
"473044022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f817
9802201229c3605c61c4133b282cc30ece9e7d5c3693bf2cd1c03a3caadcd9f25900a5412
10279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 49.99999388,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_2DUP OP_BIN2NUM 512 OP_SUB 8 OP_NUM2BIN OP_SWAP OP_CAT
OP_HASH256 -2147483647 OP_SWAP OP_CAT OP_CAT OP_CAT OP_CAT
0000000041000000 OP_CAT OP_HASH256 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT 0 OP_CAT OP_BIN2NUM
9817f8165b81f259d928ce2ddbfc9b02070b87ce9562a055acbbdcf97e66be79 OP_ADD
414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffff00 OP_MOD
OP_DUP 414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffff00
2 OP_DIV OP_GREATERTHAN OP_IF
414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffff00
OP_SWAP OP_SUB OP_ENDIF OP_SIZE OP_DUP OP_TOALTSTACK OP_TOALTSTACK 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT OP_FROMALTSTACK 32 OP_GREATERTHAN OP_IF 1 OP_SPLIT OP_ENDIF
```

```
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_FROMALTSTACK 32 OP_GREATERTHAN OP_IF OP_SWAP OP_CAT OP_ENDIF OP_SIZE
OP_DUP 36 OP_ADD 48 OP_SWAP OP_CAT
022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f8179802
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT 65 OP_CAT
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
OP_CHECKSIGVERIFY OP_DUP OP_HASH160
751e76e8199196d454941c45d1b3a323f1433bd6 OP_EQUALVERIFY OP_CHECKSIG",
        "hex":
"6e810200029458807c7eaa04ffffffff7c7e7e7e7e0800000000410000007eaa517f517f
517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f5
17f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c
7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7
e7c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2ddbfc9b02070b87ce95
62a055acbbdcf97e66be799321414136d08c5ed2bf3ba048afe6dcaebafeffffffffffff
ffffffffffffffff00977621414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffff
ffffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6dcaebafeffffffffff
ffffffffffffffffff007c946882766b6b517f517f517f517f517f517f517f517f517f51
7f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517
f517f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7
c7e7c7e6c0120a0637c7e68827601249301307c7e23022079be667ef9dcbbac55a06295ce
870b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e01417e210279be667ef9dcb
bac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ad76a914751e76e8199196
d454941c45d1b3a323f1433bd688ac",
        "type": "nonstandard"
      }
    }
  ],
  "hex":
"0100000001f55d6593b9774cad9f9bc113a39231298fc49b69ee3cc28768c7e5aadb5b68
52000000006a473044022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f
2815b16f8179802201229c3605c61c4133b282cc30ece9e7d5c3693bf2cd1c03a3caadcd9
f25900a541210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f
81798ffffffff019cef052a01000000fd32026e810200029458807c7eaa04ffffffff7c7e
7e7e7e0800000000410000007eaa517f517f517f517f517f517f517f517f517f517f517f5
17f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f51
7f517f7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7
e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e01007e81209817f8
165b81f259d928ce2ddbfc9b02070b87ce9562a055acbbdcf97e66be799321414136d08c5
ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffffff00977621414136d08c5e
d2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffffff005296a06321414136d08
c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffffff007c946882766b6b51
7f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517
f517f517f517f517f517f517f517f517f517f517f517f517f517f6c0120a063517f687c7e7c7e
```

```
7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7
c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e6c0120a0637c7e6882760124930130
7c7e23022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f8179
8027e7c7e7c7e01417e210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959
f2815b16f81798ad76a914751e76e8199196d454941c45d1b3a323f1433bd688ac0000000
0"
}
```

Transaction 1 – spending transaction

```
{
  "txid":
"c700e1d6c995e4c77014536d4431be84d7fb40d3fbef52ed85be2ad06414eac8",
  "hash":
"c700e1d6c995e4c77014536d4431be84d7fb40d3fbef52ed85be2ad06414eac8",
  "version": 1,
  "size": 1415,
  "locktime": 0,
  "vin": [
    {
      "txid":
"88b9d41101a4c064b283f80ca73837d96f974bc3fbe931b35db7bca8370cca34",
      "vout": 0,
      "scriptSig": {
        "asm":
"3044022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
0220388bd5f619c02287145cf0bb3bc440f883b09e35e67a4adcf50635800219ed34[ALL|
FORKID]
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
01000000fb472de1f838d9560dc7b19b1ab62b0c6ed60580779017d3cd32d22bcc051ce13
bb13029ce7b1f559ef5e747fcac439f1455a2ec7c5f09b72290795e7066504434ca0c37a8
bcb75db331e9fbc34b976fd93738a70cf883b264c0a40111d4b98800000000
fd32026e810200029458807c7eaa04ffffffff7c7e7e7e7e0800000000410000007eaa517
f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f
517f517f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7e7c7e7c7e7c7e7e7
c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c
7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2ddbfc9b02070b8
7ce9562a055acbbdcf97e66be799321414136d08c5ed2bf3ba048afe6dcaebafeffffffff
ffffffffffffffffffffff00977621414136d08c5ed2bf3ba048afe6dcaebafeffffffff
ffffffffffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6dcaebafeffffff
ffffffffffffffffffffff007c946882766b6b517f517f517f517f517f517f517f517f5
17f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f51
7f517f517f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7
e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e6c0120a0637c7e68827601249301307c7e23022079be667ef9dcbbac55a06
295ce870b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e01417e210279be667e
f9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ad76a914751e76e81
99196d454941c45d1b3a323f1433bd688ac 9cef052a01000000",
        "hex":
"473044022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f817
980220388bd5f619c02287145cf0bb3bc440f883b09e35e67a4adcf50635800219ed34412
10279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f817984c6801
```

```
000000fb472de1f838d9560dc7b19b1ab62b0c6ed60580779017d3cd32d22bcc051ce13bb
13029ce7b1f559ef5e747fcac439f1455a2ec7c5f09b72290795e7066504434ca0c37a8bc
b75db331e9fbc34b976fd93738a70cf883b264c0a40111d4b988000000004d3502fd32026
e8102000294588 07c7eaa04ffffffff7c7e7e7e7e0800000000410000007eaa517f517f51
7f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517
f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7
c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2ddbfc9b02070b87ce9562
a055acbbdcf97e66be799321414136d08c5ed2bf3ba048afe6dcaebafefffffffffffffff
fffffffffffffff00977621414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffff
ffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6dcaebafefffffffffffff
fffffffffffffffff007c946882766b6b517f517f517f517f517f517f517f517f517f517f
517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f5
17f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c
7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7
e7c7e6c0120a0637c7e68827601249301307c7e23022079be667ef9dcbbac55a06295ce87
0b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e01417e210279be667ef9dcbba
c55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ad76a914751e76e8199196d4
54941c45d1b3a323f1433bd688ac089cef052a01000000"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 49.99998876,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_2DUP OP_BIN2NUM 512 OP_SUB 8 OP_NUM2BIN OP_SWAP OP_CAT
OP_HASH256 -2147483647 OP_SWAP OP_CAT OP_CAT OP_CAT OP_CAT
0000000041000000 OP_CAT OP_HASH256 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT 0 OP_CAT OP_BIN2NUM
9817f8165b81f259d928ce2ddbfc9b02070b87ce9562a055acbbdcf97e66be79 OP_ADD
414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffff00 OP_MOD
OP_DUP 414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffff00
2 OP_DIV OP_GREATERTHAN OP_IF
414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffff00
OP_SWAP OP_SUB OP_ENDIF OP_SIZE OP_DUP OP_TOALTSTACK OP_TOALTSTACK 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
```

```
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1 OP_SPLIT 1
OP_SPLIT OP_FROMALTSTACK 32 OP_GREATERTHAN OP_IF 1 OP_SPLIT OP_ENDIF
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_FROMALTSTACK 32 OP_GREATERTHAN OP_IF OP_SWAP OP_CAT OP_ENDIF OP_SIZE
OP_DUP 36 OP_ADD 48 OP_SWAP OP_CAT
022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f8179802
OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT 65 OP_CAT
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
OP_CHECKSIGVERIFY OP_DUP OP_HASH160
751e76e8199196d454941c45d1b3a323f1433bd6 OP_EQUALVERIFY OP_CHECKSIG",
        "hex":
"6e810200029458807c7eaa04ffffffff7c7e7e7e7e0800000000410000007eaa517f517f
517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f5
17f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7e7c7e7c7e7c7e7c
7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7
e7c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2ddbfc9b02070b87ce95
62a055acbbdcf97e66be79932141 4136d08c5ed2bf3ba048afe6dcaebafeffffffffffff
ffffffffffffffff00977621414136d08c5ed2bf3ba048afe6dcaebafeffffffffffff
ffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6dcaebafeffffffffff
ffffffffffffffffff007c946882766b6b517f517f517f517f517f517f517f517f51
7f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517
f517f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7
c7e7c7e6c0120a0637c7e68827601249301307c7e23022079be667ef9dcbbac55a06295ce
870b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e01417e210279be667ef9dcb
bac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ad76a914751e76e8199196
d454941c45d1b3a323f1433bd688ac",
        "type": "nonstandard"
      }
    }
  ],
  "hex":
"010000000134ca0c37a8bcb75db331e9fbc34b976fd93738a70cf883b264c0a40111d4b9
8800000000fd150347304402207 9be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d
959f2815b16f81798022038 8bd5f619c02287145cf0bb3bc440f883b09e35e67a4adcf506
35800219ed3441210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815
b16f817984c6801000000fb472de1f838d9560dc7b19b1ab62b0c6ed60580779017d3cd32
d22bcc051ce13bb13029ce7b1f559ef5e747fcac439f1455a2ec7c5f09b72290795e70665
04434ca0c37a8bcb75db331e9fbc34b976fd93738a70cf883b264c0a40111d4b988000000
004d3502fd32026e810200029458807c7eaa04ffffffff7c7e7e7e7e0800000000410000 0
07eaa517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f
517f517f517f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7e7c7e7
c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c
```

```
7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2ddbfc9
b02070b87ce9562a055acbbdcf97e66be799321414136d08c5ed2bf3ba048afe6dcaebafe
ffffffffffffffffffffffffffffff00977621414136d08c5ed2bf3ba048afe6dcaebafef
ffffffffffffffffffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6dcaeba
feffffffffffffffffffffffffffff007c946882766b6b517f517f517f517f517f517f5
17f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f51
7f517f517f517f517f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7
e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e7c7e7c7e6c0120a0637c7e68827601249301307c7e23022079be667ef9dcb
bac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e01417e2102
79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ad76a9147
51e76e8199196d454941c45d1b3a323f1433bd688ac089cef052a01000000ffffffff019c
ed052a01000000fd32026e810200029458807c7eaa04ffffffff7c7e7e7e7e08000000004
10000007eaa517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f51
7f517f517f517f517f517f517f517f517f517f517f517f517f517f517f7c7e7c7e7c7c7
e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e
7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e01007e81209817f8165b81f259d928ce2
ddbfc9b02070b87ce9562a055acbbdcf97e66be799321414136d08c5ed2bf3ba048afe6dc
aebafefffffffffffffffffffffffffffff00977621414136d08c5ed2bf3ba048afe6dca
ebafefffffffffffffffffffffffffffff005296a06321414136d08c5ed2bf3ba048afe6
dcaebafefffffffffffffffffffffffffffff007c946882766b6b517f517f517f517f517
f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f517f
517f517f517f517f517f517f517f517f6c0120a063517f687c7e7c7e7c7e7c7e7c7e7c7e7c7e7
c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c7e7c
7e7c7e7c7e7c7e7c7e7c7e7c7e6c0120a0637c7e68827601249301307c7e23022079be667
ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798027e7c7e7c7e0141
7e210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798ad7
6a914751e76e8199196d454941c45d1b3a323f1433bd688ac00000000"
}
```