Bitcoin
Transactions (pre-segwit)

Bassam El Khoury Seguias

January 11, 2020

1 Introduction

Part of philosophy's oddity lies in its unwavering fueling of the desire of each and every one of its practitioners to put an end to it. The path to fulfillment ends in the revelation of a version of the truth that reigns supreme over any version that preceded it and any that would otherwise follow it.

Against this backdrop, many philosophers thoroughly investigated and debated the nature of the universe. In doing so, different truths emerged on a spectrum bounded by two extremes that were vehemently defended by some of humanity's brightest. On one end, a Parmenidean view of a static universe and its absolutely eternal reality. On the other, a Heraclitean representation of a changing universe in an eternal state of flux.

It is not my intent nor is it within the extent of my capabilities to resolve this dichotomy. However, there seems to be substantial empirical evidence that movement is inherently linked to existence. Be it microscopic or macroscopic, a certain notion of change appears to be intimately tied to the very fabric of reality. This flow is particularly noticeable at the level of the various interactions that happen among individuals or groups.

A significant part of these exchanges is succinctly encapsulated in what we refer to as a *transaction*. In its most general setting, a *transaction* refers to *that* which is "driven through", "accomplished" or "settled". It is derived from the pairing of the Latin words *trans* and *agere* which respectively mean "through" and "driving forward". Implicit to this etymology is the notion of a movement, the subject of which could be a physical good or an intangible (e.g., a service, a right, a piece of information) originating at one or multiple sources and destined to one or multiple recipients.

A Bitcoin transaction is no exception as it fundamentally consists of transferring spending control from one entity to another. In this context, control refers to the authority that a given entity benefits from in order to unlock a certain value. As such, a Bitcoin transaction of Satoshi $1,000,000$ from Alice to Bob is an activity that ensures that the control over spending these Satoshis has moved from Alice to Bob who can now spend them (or a portion of them) at will. A Satoshi is the smallest transactable unit of a bitcoin

(the currency, also denoted BTC) and is equal to BTC 10^{-8}. In light of this description, one can define a Bitcoin transaction as a data structure that essentially includes:

- A set of unspent previous Bitcoin transaction outputs commonly known as **UTXOs**. Each one of them contains a specific amount of Satoshis, the control over which has been transferred from a previous entity to the one initiating the current Bitcoin transaction. UTXOs become inputs to the current Bitcoin transaction.
- One or more recipients who will be given spending control over UTXOs.
- An amount specifying the value of Satoshis to be transferred to each recipient.
- Cryptographic signatures and relevant scripts used to verify the authenticity of the sender(s) as well as to codify and observe any spending rule(s) imposed by them.

The fourth point above is of particular importance. It is commonly stated that a signature is applied to a given Bitcoin transaction. However, a Bitcoin transaction is not necessarily characterized by a single signature and can have as many of those as the number of UTXOs it consumes or more (if e.g., a UTXO requires a multisignature). The reason for this is that each UTXO can require a proof of sender(s)'s authenticity as a necessary condition for unlocking the amount it encapsulates. Two design dimensions ensue from this observation:

1. **A choice of a signature message**: Each signature is applied to a **message** which is built on a modified subset of the content of the Bitcoin transaction. Most importantly, the procedure for devising the message is not monolithic and exhibits instead a certain flexibility in choosing what content to take into account. As a result, a message can be a handful of things depending on which constituents of the Bitcoin transaction are passed to Bitcoin's signing algorithm. In order to describe which procedure was followed, a **sighash** byte is specified and appended to the signature.

2. **A codification of the spending conditions**: Aside from requiring a proof of a sender's authenticity, a UTXO may also have other rules that constrain its spending. Conditions are generally encapsulated in a field known as a **locking script** or **scriptPubKey** and expressed using Bitcoin's **Script** programming language. On the other hand, a recipient must provide relevant data in the form of an adequate **unlocking script** also known as **scriptSig**, in order to claim control over a transferred amount.

   The right of the recipient to claim said control in exchange of providing relevant proof as mandated by the sender is enforced through a **smart contract**. In essence, a smart contract is a computer program that verifies and executes the rules dictating the interplay between different parties. It was originally introduced by the American cryptographer, computer scientist and legal scholar Nick Szabo [16], [17]. In the context of Bitcoin, a necessary condition for a transaction to be valid is for the pairing of scriptSig and scriptPubKey to evaluate to **True**.

By virtue of its digital nature, a Bitcoin transaction is internally represented in **raw** form as a byte-stream. The different elements of the transaction data structure are mapped to a sequence of bits through a well-defined **serialization** process. Conversely, one could take a raw transaction and **deserialize** it into its corresponding human-readable format. These two equivalent representations tend to be lengthy, making them rather cumbersome to use whenever referencing a particular Bitcoin transaction. As a result, a more compact identifier known as the transaction identity or **txid** was introduced. It consists of a 32 byte sequence obtained by adequately subjecting the raw representation to specific hashing operations.

The objective of this chapter is to provide an introduction to the mechanics of a Bitcoin transaction predating the **Segregated Witness (SegWit)** activation which will be separately discussed in another chapter. The content is organized as follows:

- Section 2 introduces the **building blocks** of a generic **Bitcoin transaction**. It also describes the serialization process that maps a Bitcoin transaction in human-readable form into its corresponding raw representation and shows how to derive a transaction's txid from its serialized representation.
- Section 3 is a brief introduction to the **Bitcoin Script** used to codify spending conditions imposed on certain UTXOs.
- Section 4 introduces various types of spending conditions that can be expressed in Script and imposed on a UTXO as part of its scriptPubKey. Clearly, this cannot be an exhaustive treatment and is limited to examples that are deemed more relevant than others based on their practicality and usability as of the time of writing.
- Section 5 is dedicated to building relevant python methods to illustrate the serialization and deserialization processes for **Bitcoin transactions** that spend **P2PKH** or **P2SH** outputs. In particular, we consider a special case of a P2SH output that requires **multiple signatures** to be unlocked. The purpose is to provide a more detailed view of the building blocks of some of the most common Bitcoin transactions. The python methods could also assist those with no access to a Bitcoin client to conveniently query the blockchain and interpret raw transactions.
- Section 6 describes two special types of Bitcoin transactions. The first is the **Coinbase** transaction created whenever a new block is mined. Its uniqueness stems from the fact that it does not have any inputs associated with it but generates nevertheless a well-defined amount of bitcoins that the miner can claim as a block reward. The second type of transactions is one that inscribes **data** onto the blockchain by using the special **OP_RETURN** opcode.
- Section 7 introduces the different types of **signature hashes** (i.e., **sighash**) that affect how a message is prepared for signature. Given the seemingly confusing nature of this topic, we build relevant python methods from scratch in order to illustrate the mechanics of constructing messages based on the sighash byte. We then extract the signatures pertaining to these messages from the relevant raw representation and run them through the ECDSA verification algorithm to demonstrate their validity.

- Section 8 concludes with an introduction to **transaction malleability**, its effects, and its prevalence in the legacy Bitcoin transaction structure. The aim is to motivate the need for a mallebility-free construct of which SegWit is a working example.

We assume that the reader is familiar with Bitcoin's keys and address system as introduced in the chapter *"Bitcoin Private key, Public key, and Addresses"*. We also recommend that the reader be familiar with the basics of ECDSA signature as presented in the chapter *"Bitcoin Elliptic Curve Digital Signature Algorithm (ECDSA)"*.

2 Building blocks of a Bitcoin transaction

Common sense dictates that in order to conduct a generic transaction there needs to be, at a minimum, one payer and one payee. In its simplest setting, the payee would receive payment made by the payor in exchange of her goods or services. One could then capture the essence of a generic transaction by specifying a few parameters including:

- The payer's source(s) of funds indicating where payment would come from.
- An assurance that the payer is compliant with any spending condition attached to the selected source(s) of funds. For instance, this could be a proof that these sources are legitimately owned by the payer.
- The payee's account details indicating the new destination of funds.
- A specification of the payment amount destined to the payee.
- Any spending encumbrance to be imposed on the payee.

A Bitcoin transaction is a data structure that encapsulates this information in a handful of fields. They include a **Version** field, **Inputs** and **Outputs** fields respectively referred to as **vin** and **vout**, and a **LockTime** field commonly referred to as **nLockTime**. In what follows, we define each of these fields in more detail.

Version: It can be argued that progress and innovation (be it incremental or groundbreaking) have been largely rooted in Man's desire to improve on the status quo. Underlying this line of argument is a recognition that achieving the *summum bonnum* of anything requires a process of iteration. In particular, a Bitcoin transaction could benefit from a margin of flexibility in redefining its constituents if and when the need arises (e.g., introducing a new consensus rule as was done in BIP 68 [11]). One way that a Bitcoin transaction introduces this flexibility is through its **4 byte long** Version field. Usually, a Bitcoin transaction has it set to the decimal value 1, but higher versions can be used (e.g., version 2 in the case of BIP 68).

nLockTime: This field ensures that the Bitcoin transaction does not get mined until a future time instance or until the blockchain reaches a certain future block height. It is important to note that the **4 byte long** nLockTime field is an unsigned integer that denotes an **absolute** value specifying a block height or a time instance. To distinguish between the two value types, a threshold of 500×10^6 is used:

- If nLockTime is set to a positive value that is less than 500×10^6, it is interpreted as the minimum block height required to be reached on the blockchain for this Bitcoin transaction to be eligible for valid mining.
- If it is set to a value above 500×10^6, it is interpreted as the earliest timestamp (in Unix epoch time) before which the Bitcoin transaction will not be eligible for valid mining.
- If nLockTime is equal to 0 (or less than or equal to the current block height or Unix epoch time), the Bitcoin transaction is eligible for immediate mining.

The "absolute" specifier is a misnomer. It indicates that a block height is measured with respect to the decimal value 0, while a time argument is meant to refer to a Unix Epoch time denoting the number of seconds that have passed since 00:00:00, 1 January 1970. This is to be contrasted with relative time locks that we will introduce a bit later and that enable the Bitcoin transaction initiator to choose a more dynamic reference point.

Importantly, if any party attempts to transmit a Bitcoin transaction whose nLockTime field has not matured yet, it will automatically be rejected by the first node to receive it and not relayed to peer nodes. For nLockTime to be enabled, a transaction must have at least one of its **nSequence** fields (to be discussed in the following paragraph) set to a value below 0xffffffff.

nLockTime (examples)

nLockTime			Description
Hex	Decimal	Type	
0x000BAD3B	765243	Block Height	*Transaction can only be mined after the blockchain achieves a **height of at least 765,243 blocks.***
0x77157B95	1997896597	Time	*Transaction can only be mined after the Median Time Past (MTP) of the last 11 blocks corresponds to a date of **at least Saturday, April 23, 2033 7:16:37 PM (GMT).***
0x535339CF	1397963215	Time	*The Time value corresponds to **Sunday, April 20, 2014 3:06:55 AM (GMT)**. Given that this a date in the past, the transaction can be mined at any time/block height.*

vin: A Bitcoin transaction must specify enough detail about the source of funds it intends to use. It should come as no surprise that any source of funds referenced by a Bitcoin transaction should have either been legitimately generated by the network or transferred from one peer to another. As a result, we get two possible scenarios:

1. New Satoshis are created upon the successful mining of a new block that extends the blockchain with the highest amount of work to date. The creation of new Satoshis takes place as part of a specific type of Bitcoin transaction known as a **Coinbase** transaction. We will revisit it in more detail in section 6.
2. Existing Satoshis were sent to the current sender as part of a previous valid Bitcoin transaction. Any transaction's output that was destined to the current issuer and that has not been spent as of yet is known as an **Unspent**

Transaction Output (UTXO). Each one of these UTXOs contains an amount of Satoshis, the control over which has been transferred from a previous entity to the one initiating the current Bitcoin transaction.

More specifically, a Bitcoin transaction must include the following input information:

- How many UTXOs (sources of funds) it will consume. This is specified in a **variable length input counter**.
- Where each one of these UTXOs is located. This is specified in two subfields, jointly referred to as the UTXO's **outpoint**:
  - > The unique identifier of a previous Bitcoin transaction of which that particular UTXO was an output. This **32 byte** identifier is known as the Bitcoin transaction's **txid** and we will describe it in more detail later on.
  - > Since a Bitcoin transaction may contain many UTXOs, it is imperative to specify the **index** of the particular UTXO being used in the current transaction. A UTXO index is specified as part of a **4byte long** sequence.
- For each UTXO, a proof that all spending conditions associated with it have been met. Most of the time this consists of a signature proving that the sender is the legitimate owner of this UTXO. But **spending conditions** could be more varied as we will see in section 4. This information is encapsulated in a **variable-length** field known as **ScriptSig** or unlocking script. Given its variable length, a scriptSig is always preceded by an adequate stream of bytes indicating its length.
- An additional **4-byte sequence number** field or **nSequence**. Satoshi's original implementation allowed for a degree of flexibility before including a Bitcoin transaction in a block. Each transaction input has its own nSequence field, the maximum value of which is **0xffffffff**. It is believed that this field was originally meant to enable procedures similar to the following:
  - * An entity sends a Bitcoin transaction with a pre-specified nLockTime value. Furthermore, at least one of its nSequence fields is strictly less than 0xffffffff.
  - * The Bitcoin transaction is included in the mempool (i.e., a set of Bitcoin transactions that were initiated but not yet mined as part of a block on the blockchain) and not supposed to be mined until nLockTime is reached.
  - * Prior to reaching the nLockTime value, relevant parties can update the Bitcoin transaction by increasing the appropriate nSequence number(s).
  - * Once all nSequence values become 0xffffffff, the Bitcoin transaction is considered ready to be mined (even if nLockTime has not been reached yet).

  For example, such a procedure could facilitate the following practical scenarios:

  1. Two (or more) entities may need to engage in an on-going series of interactions involving back and forth payments. Relevant parties can upload an initial Bitcoin transaction to the mempool with nSequence equals to 0. Every subsequent interaction would increase nSequence until all parties decide to publish the final state on the blockchain.

2. A transaction may involve multiple signers that don't necessarily sign it simultaneously. In this case, the Bitcoin transaction is added to the mempool with nSequence values of all relevant inputs set to below 0xffffffff. Every time a new signature is secured, the appropriate nSequence gets updated. When all signers have signed and all nSequence fields set to 0xffffffff, the Bitcoin transaction is considered final and ready for inclusion in a block.

This original implementation of nSequence erroneously assumed that miners would always mine a version of a Bitcoin transaction with a higher nSequence, even if that version reflected a miner's fee (i.e., the fee claimed by a miner for including the Bitcoin transaction in a valid block) that is less than one of its predecessors. In reality, miners aim to maximize their profits and may choose an earlier version that is more profitable. Furthermore, this implementation paved the way to a potential Denial of Service (DoS) attack where an attacker would flood the network with as large a number of replacement transactions as she pleases while only incurring a small fee associated with the cost of the newest transaction.

To address these issues, BIP 125 ("Opt-in Full Replace-by-Fee Signaling" (RBF)) [12] was enacted. The proposal countered the risk of a DoS attack by embedding a structural disincentive affecting the fee of replacement transactions. The replacement fee would now need to be higher than the cumulative fees of all its predecessors. More specifically, every replacement transaction must pay a fee greater than or equal to the sum of:

i. The sum of the fees of each of its predecessors and
ii. A fee accounting for the bandwidth to be consumed by the replacement transaction and that must be set above the node's minimum relay fee.

Aside from this remedy, nSequence was repurposed as part of BIP 68 [11] and subjected to new consensus rules that allowed for additional functionality. In particular, it became possible to use **relative time locks** as opposed to their absolute counterparts. As part of BIP 68, nSequence was required to obey the following rules:

- If its leftmost bit (as expressed in big endian notation) is set to 0, i.e., 0x00000000 $\leq$ nSequence $\leq$ 0x7fffffff, then nSequence is used to enable **relative locktime (RLT)**. It also indicates that nLockTime is enabled and RBF is signaled. This most significant bit is also known as the **Disable Flag**.

  The effect is similar to nLockTime in that a transaction gets locked until a future date. However, this date is specified relative to a UTXO-specific time or block height as opposed to a universal time or absolute block height:

  * The UTXO-specific time corresponds to the mining date of the UTXO appearing in the Bitcoin transaction input for that particular nSequence field. The mining date is the **Median-Time-Past (MTP)** of the block in which the UTXO was mined. We will define MTP shortly.
  * The UTXO-specific block height corresponds to the height of the block

in which it was originally mined.

In order to specify whether relative time is measured in actual time or block height, a **Type Flag** is specified. This flag corresponds to the 23rd rightmost bit of nSequence and is interpreted as follows:

* A Type Flag of 1 indicates a time-based lock-time. nSequence's least significant 16 bits turn into a minimum time constraint over the input's UTXO MTP. A value of n prohibits inclusion of the input in blocks prior to the one mined $512 * n$ seconds after the relevant UTXO's mining date.

* A Type Flag of 0 indicates a block-based lock-time. The nSequence value's least significant 16 bits denote the minimum number of blocks above the UTXO's block height for it to be spent legitimately.

– If the Disable Flag is set to 1, the RLT is disabled. However, nSequence could still be used to enable nLockTime or signal RBF. In particular:

  * 0x8fffffff $\leq$ nSequence $\leq$ 0xfffffffd indicates that RLT is not enabled but that RBF and nLockTime are both enabled.

  * nSequence = 0xfffffffe indicates that RLT is not enabled, that RBF is not signaled but that nLockTime is still enabled.

  * nSequence = 0xffffffff indicates that RLT is not enabled, that RBF is not signaled and that nLockTime is not enabled.

nSequence (examples)

nSequence	DF		TF		Relative Lock Time value *(if applicable)*	nLockTime	RBF	RLT	RLT Type	RLT Value
0x00400008	0	0 0 0 0 0 0 0 0	1	0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	Enabled	Enabled	Enabled	Time	512*8 sec
0xFEDC3210	1	1 1 1 1 1 1 0 1	1	0 1 1 1 0 0	0 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0	Enabled	Enabled	Disabled	N/A	
0x00000010	0	0 0 0 0 0 0 0 0	0	0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	Enabled	Enabled	Enabled	Block	16 blocks
0xFFFFFFFF	1	1 1 1 1 1 1 1 1	1	1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	Disabled	Disabled	Disabled	N/A	
0xFFFFFFFE	1	1 1 1 1 1 1 1 1	1	1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0	Enabled	Disabled	Disabled	N/A	
0xFFFFFFFD	1	1 1 1 1 1 1 1 1	1	1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1	Enabled	Enabled	Disabled	N/A	
0x004000AA	0	0 0 0 0 0 0 0 0	1	0 0 0 0 0 0	0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0	Enabled	Enabled	Enabled	Time	512*170 sec

SOURCE: https://en.bitcoinwiki.org/wiki/NSequence

Note that a Bitcoin transaction will be considered invalid if any of its nSequence fields did not age appropriately at the time it gets submitted.

We mentioned that relative time calculation uses MTP. Each block has a timestamp that the miner sets. Given network latencies, consensus rules allow miners a certain flexibility in setting it. With the advent of time locks, this opened the possibility for miners to misrepresent the timestamp and take advantage of time-locked transactions. BIP 113 [13] proposed a remedy by modifying the notion of consensus time and introducing the Median Time Past. It is calculated

by taking the median timestamp of the most recent eleven blocks. By doing so, the possible abuse by any one miner in setting her block's timestamp gets reduced.

vout: A Bitcoin transaction must provide information about the recipient(s) of its funds. In particular, it must specify the following:

>> A **variable length output counter** indicating the total number of outputs.

>> For each output, the amount of Satoshis as specified in an **8 byte long** field.

>> For each output, any spending conditions imposed on the recipient. They are specified in a **variable-length** field known as **ScriptPubKey** or **locking script**. A scriptPubKey is preceded by an adequate stream of bytes to indicate its length.

Here is an example of an actual Bitcoin transaction:

Bitcoin Transaction Example

```
Txid: dbebe45e62370aeab972a9bbbee80f99febe6c904fe49b68efe7cc877a6cfd73
{
  nVersion: 00000001
  vin:[
  {
        prev_out_0:
        {
                txid: 756c1cf676c73b951ecb3b281b375858938b503c2b9b296d9d1cd59e839daea0
                index: 00000000
                scriptSig:
                493046022100b999de2e23127ec2edf16e2f267b4c2df57b9766059369cee85cbc0a
                41be6882022100d09c405f825eec986ca2bf6f35d1267ad7d595042fca4b4f7af3f9
                adfea68d330141040bf69616981e5970c992a0762f441abcadfed9fc4630fa5e1b82
                ab00e81d16905d3820e073e1bd4a9dcfed336f4bf25edc634c2e174989767d299748
                359c2daf
                sequence: ffffffff
        }
  }
  {
        prev_out_1:
        {
                txid: 7ba03bdf67824990fbdd1a48b3fdc42ab3bcddb8b808c2c30e4d3cc4c206be52
                index: 00000001
                scriptSig:
                48304502201193da6f0c1b3f15497415fd75743439374939191331bd7ca1ac183580
                ad4273022100e435bd3c48929d9789810634af47a0461e684dd490132a9c5757af86
                296ce0d70141046cc9eeffe66726abb725d191537f87c023202eb13ede9031d7adb8
                0ecb0ddc9aa380cb2659747b850ea577cf04f01248ca9291976523a94ef0a907e6bb
                15bd55
                sequence: ffffffff
        }
  }]
  vout:[
  {
        value: 3916000 satoshis
        scriptPubKey:
        OP_DUP OP_HASH160 OP_PUSHBYTES_20 e1e1ffc33423807d6914de976738bbdc01477c2d OP_EQUALVERIFY OP_CHECKSIG
  }
  {
        value: 77452 satoshis
        scriptPubKey:
        OP_DUP OP_HASH160 OP_PUSHBYTES_20 19e75cce5ff697a01e14ec3ebcc9a4523e44caf1 OP_EQUALVERIFY OP_CHECKSIG
  }]
  nLockTime: 00000000
}
```

It is displayed in **human-readable** form commonly referred to as **JSON** format (JavaScript Object Notation).

The Version field is equal to **0x00000001** in hexadecimal notation which corresponds to the decimal value **1**.

The vin field consists of two inputs. The first contains a UTXO from a previous Bitcoin transaction with txid (will be explained shortly):

0x756c1cf676c73b951ecb3b281b375858938b503c2b9b296d9d1cd59e839daea0

Since a Bitcoin transaction can have more than one output, it is also important to specify which one of these outputs the UTXO corresponds to. In this case, it is the UTXO whose index (i.e., position among all outputs of the previous Bitcoin transaction) is equal to **0x00000000** i.e., **0** in decimal notation.

The scriptSig field which in this particular case consists of a DER-encoded ECDSA signature and a relevant public key is a 140 byte long sequence:

0x493046022100b999de2e23127ec2edf16e2f267b4c2df57b9766059369cee85c bc0a41be6882022100d09c405f825eec986ca2bf6f35d1267ad7d595042fca4b4f 7af3f9adfea68d330141040bf69616981e5970c992a0762f441abcadfed9fc4630fa 5e1b82ab00e81d16905d3820e073e1bd4a9dcfed336f4bf25edc634c2e17498976 7d299748359c2daf

We will discuss scriptSig fields in more detail in sections 4 and 5.

Lastly, the nSequence field is set to its maximum value of **0xffffffff**.

The second input has a similar structure. Note that by setting the nSequence fields of all the inputs to **0xffffffff**, nLockTime, RBF and RLT become disabled for this Bitcoin transaction. This indicates this transaction's readiness for immediate mining.

The vout field has two outputs. The first corresponds to an amount of **Satoshi 3,916,000** or equivalently BTC 0.03916. Moreover, this output is encumbered with a spending condition dictated by a 25 byte long scriptPubKey. We will discuss the operators that appear in the scriptPubKey field in section 3 and introduce specific examples in section 4. The second output has a similar structure.

Finally, the nLockTime field is set to **0x00000000** which corresponds to decimal value 0. This indicates that the Bitcoin transaction is eligible for immediate mining. Note that even if this value corresponded to a future block height or Unix time, it would not have affected this particular Bitcoin transaction since all its nSequence fields were set to 0xffffffff.

A JSON representation of a Bitcoin transaction makes it easier for humans to interpret it. However, at the level of the network, the Bitcoin transaction is represented as a

sequence of bytes and is referred to as **raw** or **serialized** representation. Without much difficulty, one can map a JSON to its serialized counterpart. To do so, special care must be given to the way certain fields are represented. Specifically, the version, txid, index, nSequence, output value, and nLockTime fields must be represented using **little endian** notation. This means that the least significant byte of the field comes first, followed by the next least significant byte and so on:

We include below a python method sourced from [15] that takes a hex string **x** and changes its endianness from big to little and vice-versa:

```
def change_Endianness(x):
    if (len(x) % 2) == 1:   x += "0"                    # Must have an even # of nibbles
    y = x.decode('hex')                                 # Map from hex to byte (base 256) format
    z = y[::-1]                                         # Read byte representation in reverse order
    return z.encode('hex')                              # Map result back to hex format
```

When we translate this Bitcoin transaction's relevant fields to their little endian representation, we get:

- **Version**: Mapped from 0x00000001 to 0x01000000.
- **Input #1's previous txid**: Mapped from

  0x756c1cf676c73b951ecb3b281b375858938b503c2b9b296d9d1cd59e839daea0

  to

  0xa0ae9d839ed51c9d6d299b2b3c508b935858371b283bcb1e953bc776f61c6c75

- **Input #1's previous output index**: Mapped from 0x00000000 to 0x00000000.
- **Input #1 scriptSig**: Is not affected and remains as is.
- **Input #1 nSequence**: Mapped from 0xffffffff to 0xffffffff.
- **Input #2's previous txid**: Mapped from

  0x7ba03bdf67824990fbdd1a48b3fdc42ab3bcddb8b808c2c30e4d3cc4c206be52

  to

  0x52be06c2c43c4d0ec3c208b8b8ddbcb32ac4fdb3481addfb90498267df3ba07b

- **Input #2's previous output index**: Mapped from 0x00000001 to 0x01000000.
- **Input #2 scriptSig**: Is not affected and remains as is.
- **Input #2 nSequence**: Mapped from 0xffffffff to 0xffffffff.
- **Output #1 value**: Mapped from 0x00000000003bc0e0 (which is the 8 byte hex representation of 3,916,000) to 0xe0c03b0000000000.

- **Output #1 scriptPubKey**: Is not affected and remains the same. We will see some common operational codes (OP codes) in section 3. Items that have the OP prefix and appear in the scriptPubKey have unique single-byte representations. For this Bitcoin transaction, the sequence

  OP_DUP OP_HASH160 OP_PUSHBYTES_20
  e1e1ffc33423807d6914de976738bbdc01477c2d OP_EQUALVERIFY
  OP_CHECKSIG

  is encoded as:

  0x76a914e1e1ffc33423807d6914de976738bbdc01477c2d88ac

- **Output #2 value**: Mapped from 0x0000000000012e8c (which is the 8 byte hex representation of 77,452) to 0x8c2e010000000000.

- **Output #2** scriptPubKey: Is not affected and remains as is. The sequence

  OP_DUP OP_HASH160 OP_PUSHBYTES_20
  19e75cce5ff697a01e14ec3ebcc9a4523e44caf1 OP_EQUALVERIFY OP_CHECKSIG

  is encoded as:

  0x76a91419e75cce5ff697a01e14ec3ebcc9a4523e44caf188ac

- **nLockTime**: Mapped from 0x00000000 to 0x00000000.

Putting it together, we get the following serialized representation:

Bitcoin Transaction Example – Raw format


```
01000000**02**a0ae9d839ed51c9d6d299b2b3c508b935858371b283bcb1e953bc776f61c6c75000000000**8c**493046022100b999
de2e23127ec2edf16e2f267b4c2df57b9766059369cee85cbc0a41be6882022100d09c405f825eec986ca2bf6f35d1267ad7
d595042fca4b4f7af3f9adfea68d330141040bf69616981e5970c992a0762f441abcadfed9fc4630fa5e1b82ab00e81d1690
5d3820e073e1bd4a9dcfed336f4bf25edc634c2e174989767d299748359c2daffffffff52be06c2c43c4d0ec3c208b8b8dd
bcb32ac4fdb3481addfb90498267df3ba07b01000000**8b**48304502201193da6f0c1b3f15497415fd75743439374939191331
bd7ca1ac183580ad4273022100e435bd3c48929d9789810634af47a0461e684dd490132a9c5757af86296ce0d70141046cc9
eeffe66726abb725d191537f87c023202eb13ede9031d7adb80ecb0ddc9aa380cb2659747b850ea577cf04f01248ca929197
6523a94ef0a907e6bb15bd55ffffffff**02**e0c03b0000000000**19**76a914e1e1ffc33423807d6914de976738bbdc01477c2d88
ac8c2e010000000000**19**76a91419e75cce5ff697a01e14ec3ebcc9a4523e44caf188ac00000000
```


The bold black bytes correspond to those that did not appear in the JSON format:

- The first **0x02** byte is an input counter and indicates that there is a total of two inputs. The input counter is of variable length. We will explain the mechanics of variable length encoding in the following paragraph.

- The **0x8c** byte indicates the length of the first scriptSig. The scriptSig is of variable length and in this case corresponds to a decimal value of **140** bytes.
- The length of the second scriptSig is 139 bytes or **0x8b** in hexadecimal.
- The second **0x02** byte is an output counter and indicates that there is a total of two outputs. The output counter is also of variable length.
- Finally, each of the two scriptPubKey fields is 25 byte long or **0x19** in hexadecimal. The scriptPubKey field is also of variable length.

We now describe how variable length encoding works for a Bitcoin transaction:

- If the length is less than or equal to 252 bytes, then the length value is captured in a single byte (e.g., the input and output counters in the previous example).
- If $253 \leq \text{length} < 2^{16}$, we include a prefix of 0xfd followed by a two-byte little endian representation of the actual length.
- If $2^{16} \leq \text{length} < 2^{32}$, we include a prefix of 0xfe followed by a four-byte little endian representation of the actual length.
- If $2^{32} \leq \text{length} < 2^{64}$, we include a prefix of 0xff followed by an eight-byte little endian representation of the actual length.

The table below illustrates variable length encoding for various length values:

Variable length serialization

Length (Decimal value)	Value set lower and upper bounds	Variable length encoding prefix	Variable length allocated bytes	Length (Hex value using big endien)	Length (Hex value using little endien)	Variable length serialization
172	1 <= value < 253	-	1	ac	ac	**ac**
38,627	253 <= value < 2^16	fd	2	96 e3	e3 96	**fd e3 96**
1,073,741,827	2^16 <= value < 2^32	fe	4	40 00 00 03	03 00 00 40	**fe 03 00 00 40**
17,179,869,179	2^32 <= value < 2^64	ff	8	00 00 00 03 ff ff ff fb	fb ff ff ff 03 00 00 00	**ff fb ff ff ff 03 00 00 00**

We now include two python methods from [15] to perform variable length encoding:

- **int2byte(a,b)**: This method converts integer **a** into its byte representation (i.e., base 256) such that the outcome's length is **b** bytes. The output is in hex format.

```
def int2bytes(a, b):
    bit_length = 8*b
    m = 2**bit_length - 1

    if a > m:
        raise Exception(str(a) +
            " cannot be represented with " + str(b)
            + " bytes. Maximum value is " + str(m))

    return ('%0' + str(2 * b) + 'x') % a
```

- **encode_Varint(value)**: This method converts an integer **value** to its varint hexadecimal format.

```python
def encode_Varint(value):
    if value < pow(2, 8) - 3:                       # If value <= 252, no prefix is needed and
        size = 1                                    # 'value' is mapped to a byte-long hex form.
        varint = int2bytes(value, size)

    else:
        if value < pow(2, 16):                      # If 253 <='value' < 2^16, then:
            size = 2                                #  1) Allocate 2 bytes for substring length
            prefix = 253                            #  2) Include a prefix byte of 0xFD

        elif value < pow(2, 32):                    # If 2^16 <='value' < 2^32, then:
            size = 4                                #  1) Allocate 4 bytes for substring length
            prefix = 254                            #  2) Include a prefix byte of 0xFE

        elif value < pow(2, 64):                    # If 2^32 <='value' < 2^64, then:
            size = 8                                #  1) Allocate 8 bytes for substring length
            prefix = 255                            #  2) Include a prefix byte of 0xFF

        else:
            raise Exception("Wrong input data size")

        varint = format(prefix, 'x') +\
            change_Endianness(int2bytes(value, size))

    return varint
```

The table below summarizes the aforementioned serialization procedure:

Bitcoin Transaction serialization procedure

	Field	Field length	Encoding
	nVersion	4 bytes	little endien
vin	vin count	variable	1 byte prefix followed by little endien
	txid_1	32 bytes	little endien
	index_1	4 bytes	little endien
	length of scriptSig_1	variable	1 byte prefix followed by little endien
	scriptSig_1	length of scriptSig_1	as is
	nSequence_1	4 bytes	little endien
	...		
	txid_(vin count)	32 bytes	little endien
	index_(vin count)	4 bytes	little endien
	length of scriptSig_(vin count)	variable	1 byte prefix followed by little endien
	scriptSig_(vin count)	length of scriptSig_(vin count)	as is
	nSequence_(vin count)	4 bytes	little endien
vout	vout count	variable	1 byte prefix followed by little endien
	amount_1	8 bytes	little endien
	length of scriptPubKey_1	variable	1 byte prefix followed by little endien
	scriptPubKey_1	length of scriptPubKey_1	as is
	...		
	amount_(vout count)	8 bytes	little endien
	length of scriptPubKey_(vout count)	variable	1 byte prefix followed by little endien
	scriptPubKey_(vout count)	length of scriptPubKey_(vout count)	as is
	nLockTime	4 bytes	little endien

Before wrapping up this section, we introduce a convenient way of referencing a Bitcoin transaction. We define the transaction id or **txid** to be the little endian representation of the double SHA256 of the serialized transaction. In other words, the raw transaction gets subjected to the SHA256 hash function two consecutive times before the outcome

gets converted to little endian representation.

The collision resistance property of hash functions (refer to the chapter entitled *"Digital Signatures and Other Prerequisites"* for an introduction to hash functions) provides an assurance that the probability of having two distinct Bitcoin transactions sharing the same txid is negligible. As a result, one can treat the txid as a unique identifier for all practical matters. Assuming tx_raw holds the raw representation of a Bitcoin transaction, its txid can be derived as follows:

$$\text{change\_Endianness}(\text{double\_Sha256}(\text{tx\_raw}))$$

For example running this on the previous Bitcoin transaction yields a txid of:

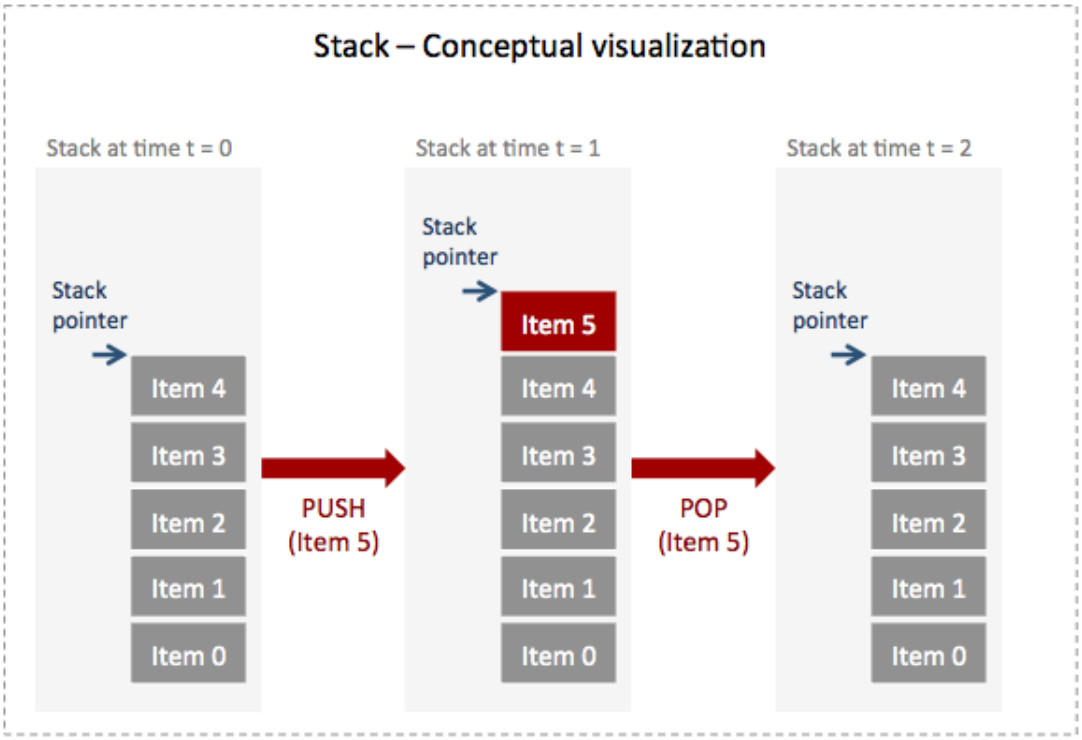
0xdbebe45e62370aeab972a9bbbee80f99febe6c904fe49b68efe7cc877a6cfd73

One can use the following **get_Serialized_Tx** python method to retrieve the raw representation of a given Bitcoin transaction. It takes a txid and a network specification (i.e., "mainnet" or "testnet") as inputs and outputs the raw format of the corresponding Bitcoin transaction. It does so by initiating an API call to adequate platforms.

```
def get_Serialized_Tx(txid, net = "mainnet"):
    assert (net in ["mainnet", "testnet"])
    if (net == "mainnet"):
        tx_raw = requests.get(
            'https://blockchain.info/en/tx/'
            + txid + '?format=hex').text
        assert (txid == change_Endianness(
                        double_Sha256(tx_raw)))          # By definition, txid must be little endian
                                                         # format of the double sha256 of raw tx)
    else:
        str_raw = requests.get(
            'https://testnet-api.smartbit.com.au/v1/blockchain/tx/'
            + txid + '/hex').text
        ind = str_raw.find("hex",                        # Find the 2nd occurence of "hex" in smartbit
                        str_raw.find("hex") + 1)         # hex output
        tx_raw = str_raw[ind+6:-4]
        assert (txid == change_Endianness(
                        double_Sha256(tx_raw)))

    return tx_raw
```

3 Bitcoin Script

Script: A Bitcoin transaction uses a specific language to express **encumbrances** related to spending a certain UTXO. The Script language is relatively simple as it is **stack-based**. A stack is a type of a linear data structure with a well defined order for performing operations. In layman terms, it is an ordered collection of items that one can either add to or remove from. Usually, the last added item is the first one to be removed justifying the "Last In First Out" or LIFO terminology. We commonly refer to the act of adding an item to the stack as **pushing** and to that of removing as **popping**.



Bitcoin's Script rule book is not very complicated and its modus operandi includes the following:

* Expressions are parsed left to right, causing the leftmost factor to be pushed first onto the stack.
* Items on the stack are manipulated through various operational codes known as **opcodes**. Each one of them is internally represented as a single byte. Due to the way they are defined, opcodes are preceded with the **OP** prefix.
* Depending on the opcode in use, the resulting output may either get pushed onto the stack or not.
* An expression is considered valid if after parsing, the top item on the stack evaluates to True.

We first introduce some of the opcodes commonly used to define locking conditions on bitcoin UTXOs. We will not cover the opcode set in its entirety and point interested readers to [1] for a comprehensive list.

- Some of the most simple opcodes do not take any input (i.e., don't operate on any factor) and output a constant integer value that gets pushed onto the stack whenever invoked. For instance each opcode in $\{\mathbf{OP\_N},\ N = 1, ..., 16\}$ pushes a specific integer $1 \leq N \leq 16$ onto the stack. OP_N has a corresponding opcode byte whose decimal representation is given by $80 + N$. For example, OP_15 corresponds to a single byte whose decimal representation is 95 or 0x5f in hex.

- Whenever new data must be pushed onto the stack, the program must be told when said data starts and when it ends. The way this is implemented in Bitcoin

involves the usage of specific **data push** opcodes. Note that data bytes pushed onto the stack are always represented in little endian notation. Broadly speaking, data push opcodes come in two forms:

1. A set of 75 opcodes $\{$**OP_PUSHBYTES**$i,\ i \in \{1, .., 75\}\}$ whose decimal (hex) representations range from 1 (**0x01**) to 75 (**0x4b**). Each one of them indicates that a number of bytes equal to their decimal representation will be pushed next onto the stack. For example, when the program encounters the byte whose hex representation is 0x0e it knows that the subsequent 14 bytes will have to be pushed onto the stack. That is assuming that 0x0e is not itself part of a data stream being currently pushed onto the stack.

2. A set of opcodes used to push data of byte size greater than 75. This set contains three opcodes: **OP_PUSHDATA1**, **OP_PUSHDATA2** and **OP_PUSHDATA4**. Their respective decimal (hex) representations are given by 76 (0x4c), 77 (0x4d), and 78 (0x4e).

   OP_PUSHDATA$i,\ i \in \{1, 2, 4\}$ indicates that the following i bytes will hold the value of the actual number of data bytes to be pushed onto the stack. For example, invoking OP_PUSHDATA2 followed by the two bytes 0xa201 (note that the protocol always uses little endian representation) indicates that the subsequent 418 bytes will be pushed onto the stack. Note that the maximum amount of bytes that can be pushed is equal to 520 and as a result, OP_PUSHDATA4 is not of much use.

   Also note that one could theoretically use OP_PUSHDATA$j,\ j \in \{2, 4\}$ to push any number of data bytes that OP_PUSHDATA$i,\ i \in \{1, 2\},\ i < j$ could otherwise push. In addition, one could use OP_PUSHDATA1 to push any number of bytes between 1 and 75, the same way opcodes 0x01 to 0x4b could. However, these alternatives are considered non-standard and most of the nodes that encounter them will not relay them to their peers.

- In some instances, the topmost element of the stack must be duplicated in order to run specific operations. **OP_DUP** with decimal representation 118 (and hex representation 0x76) does precisely that. It pops the topmost item on the stack, creates a copy of it and then pushes them both onto the stack.

- Another important function needed for various verification tasks is that of comparing two quantities and checking if they are equal. Two opcodes could be used to conduct this comparison:

  1. **OP_EQUAL** with decimal representation 135 (and hex representation 0x87) pops the top 2 elements of the stack and compares them. If they are equal, it pushes a value of 1 onto the stack. If not, it pushes a value of 0.

  2. **OP_EQUALVERIFY** with decimal representation 136 (and hex representation 0x88) pops the top 2 elements of the stack and compares them. If they are equal, then the operations continue normally without pushing anything onto the stack. Otherwise, the program execution fails.

- In addition, Script has a number of built-in opcodes that conduct various cryptographic operations. In particular:

  1. **OP_HASH160** with decimal representation 169 (and hex representation 0xa9) pops the topmost item on the stack, hashes it using SHA256 and then hashes the result using RIPEMD160 before pushing the final output onto the stack. For an introduction to Bitcoin's hashing functions, we refer the reader to the chapter entitled *"Bitcoin - Private key, Public key, and Addresses"*.

  2. **OP_HASH256** with decimal representation 170 (and hex representation 0xaa) pops the topmost item on the stack, runs two SHA256 hashing iterations on it, and then pushes the result onto the stack.

  3. **OP_CHECKSIG** with decimal representation 172 (and hex representation 0xac) does the following:

     3.1 It first creates the message against which a given signature must be verified. The message consists of a hash of various elements of a Bitcoin transaction and we will later see in section 7 how it is formed depending on what part of the Bitcoin transaction gets included.

     3.2 It then pops the two topmost items on the stack, which should consist of a signature followed by a corresponding public key.

     3.3 It uses the public key to verify the validity of the signature on the message (the reader can refer to the post entitled *"Bitcoin - Elliptic Curve Digital Signature Algorithm (ECDSA)"* for an introduction to Bitcoin's signing algorithm).

     3.4 It pushes 1 onto the stack if the signature is valid, and 0 otherwise.

     OP_CHECKSIGVERIFY with decimal representation 173 (0xad in hex) is similar to OP_CHECKSIG except that no value gets pushed onto the stack. If the output is true, the program continues to run. Otherwise, it halts.

  4. **OP_CHECKMULTISIG** with decimal representation 174 (0xae in hex) conducts an iterative version of **OP_CHECKSIG** over m different signatures and n public keys. In this case,

     4.1 Due to a design error, it pops the $(m + n + 3)$ topmost stack items instead of $(m + n + 2)$. This is why in order to effectively invoke it, an additional empty array of bytes (OP_0) gets added onto the stack before execution (we will revisit this when we discuss specific examples of scriptPubKey in section 4).

     4.2 The $(m + n + 2)$ topmost items on the stack should consist of a sequence including signatures $(sig_1, ..., sig_m)$ followed by integer $m \geq 2$ (represented by OP_m), followed by a sequence of public keys $(pk_1, ..., pk_n)$, and finally integer $n \geq m$ (represented by OP_n).

4.3 The opcode takes the first signature sig_1 and verifies it against its associated message and public key. Note that the message is constructed similarly to the case of OP_CHECKSIG which we will explore later in section 7. In order to identify the coresponding public key, the program runs sequentially through the public key set starting at pk_1 until it gets a validation or exhausts them all. Without loss of generality, let pk_r be a corresponding match for some $r \in \{1, ..., n\}$.

4.4 The opcode then repeats the same process with the second signature sig_2. The difference is that it parses the public key set starting immediately after the one that resulted in a match in the previous iteration, i.e., at pk_{r+1}.

4.5 The above step is repeated until all signatures get verified or no more public keys remain.

4.6 If all signatures get verified, then the opcode pushes the value 1 onto the stack. Otherwise, it pushes 0 instead.

Note that the above procedure dictates that signatures must be ordered in the same way as their corresponding public keys. That means that if (sig_{i_1}, pk_{j_1}) and (sig_{i_2}, pk_{j_2}), $i_1, i_2 \in \{1, ..., m\}$, $j_1, j_2 \in \{1, ..., n\}$ are a pair of tuples of matching signatures and public keys, then:

$$\forall\ i_1, i_2 \in \{1, ..., m\},\ i_1 \leq i_2 \Rightarrow j_1 \leq j_2$$

OP_CHECKMULTISIGVERIFY with decimal representation 175 (and hex representation 0xaf) is similar to OP_CHECKMULTISIG except that no value gets pushed onto the stack. If the output is true, the program continues to run. Otherwise, it halts.

- More elaborate spending conditions can be created using Script's built-in **flow control** that includes **if-else** statements. An important observation is that Script does not allow loops (e.g., "for" or "while" loops) and is hence a **Turing incomplete** language. Initially, this may seem as a limitation. However, it is a well-thought design constraint meant to strengthen Bitcoin's underlying security. Indeed, a Turing complete language could pave the way to infinite loops that malicious attackers use to jeopardize the proper functioning of nodes on the network by causing them to get stuck or crash. Despite this design limitation, various complex scripts can still be devised using Script's flow control statements:

  1. **OP_IF** with decimal representation 99 (0x63 in hex) pops the topmost item and checks its value. If it is true (i.e., a positive integer), the condition following OP_IF gets executed. Otherwise, the condition is disregarded.

  2. **OP_ELSE** with decimal representation 103 (0x67 in hex) checks if the preceding OP_IF or OP_ELSE condition was executed. If not, the current condition gets executed. Otherwise, the condition is disregarded.

3. **OP_ENDIF** with decimal representation 104 (0x68 in hex) always concludes an if-else block.

- There are two additional opcodes that merit special attention due to the flexibility they introduce in defining **time-bound** spending conditions. Together with control flow statements, they can be used to devise more elaborate spending conditions as we will see in section 4:

1. **OP_CHECKLOCKTIMEVERIFY** (OP_CLTV) with decimal representation 177 (0xb1 in hex) was introduced in BIP 65 [18] to lock UTXOs and make them unspendable until a future time or block height.

   The option of imposing time-bound spending conditions predates OP_CLTV. We previously saw how a Bitcoin transaction's nLockTime field ensures that it does not get mined until a future time or block height. nLockTime is however limited in effect since it is applied at the Bitcoin transaction level and not at the more granular UTXO level. This limitation is rooted in the fact that nLockTime makes it possible to spend a UTXO in the future but does not guarantee that it remains unspendable until then. Indeed, a UTXO appearing in a Bitcoin transaction with an activated nLockTime could be used in a separate transaction that unlocks it at an earlier time instance or block height. It is the ability to enforce unspendibility at the level of each UTXO that confers to OP_CLTV its advantage over nLockTime.

   It is important to note that the argument fed to OP_CLTV abides by the same rules that apply to nLockTime. More specifically, the argument is a 4 byte long unsigned integer that denotes a block height or a time instance depending on how it compares to the threshold value of $500,000,000$.

   Clearly, a valid Bitcoin transaction must have valid UTXOs. In particular, a time-bound constraint imposed on a Bitcoin transaction must be satisfied by all its UTXOs before the transaction gets validated. As a result, OP_CLTV's argument must be less than or equal to the nLockTime value.

   Being an opcode of the "verify" type, OP_CLTV halts execution if the outcome is False and continues normally without adding any new elements to the stack in case the outcome is True. More specifically, and as described in BIP 65, OP_CLTV reads the topmost stack element and evaluates to False if:

   i. the stack is empty; or
   ii. the top item on the stack is less than 0; or
   iii. the lock-time type (time or block height) of the top item and the nLockTime field are not the same (i.e., one with an argument value below $500,000,000$ and another above); or
   iv. the top stack item is greater than the transaction's nLockTime field; or
   v. the nSequence field of the relevant transaction input is 0xffffffff.

   Once OP_CLTV is executed successfully, its argument stays on the top of the

stack. In some instances, it may need to be dropped to ensure proper subsequent script execution. As a result, it is common to see CLTV scripts paired with the **OP_DROP** opcode whose decimal representation is 117 (0x75 in hex). All OP_DROP does is remove the topmost stack item.

2. **OP_CHECKSEQUENCEVERIFY** (OP_CSV) with decimal value 178 (0xb2 in hex) serves a similar purpose as OP_CLTV in that it locks a UTXO and enforces its unspendibility until a certain future time or block height. It differs from OP_CLTV in that the future time or block-height is relative to the time or block height when the UTXO was mined.

   Similar to nLockTime, nSequence is a transaction level field. This may seem counter-intuitive since as we saw earlier in section 2, nSequence is specified for each input while nLockTime is specified for the overall Bitcoin transaction. However, none of these nSequence fields directly encumbers a UTXO. As a matter of fact, one could reuse the UTXO in a separate transaction that spends it at an earlier time. This led to the introduction of a UTXO-level opcode to enforce relative-time locking on a specific transaction output. OP_CSV fulfills this role as articulated in BIP 112 [10].

   Being an opcode of the "verify" type, CSV halts execution if the outcome is False and continues normally without adding any new elements to the stack in case the outcome is True. More specifically, OP_CSV takes a 32 bit sequence as an argument which can indicate a relative time or block height. Its structure is similar to nSequence previously introduced in section 2. OP_CSV reads the topmost stack element and evaluates to False in case:

   i. the stack is empty; or
   ii. the top item on the stack is less than 0; or
   iii. the top item on the stack has the Disable Flag unset; and
      > the Bitcoin transaction version is less than 2; or
      > the Bitcoin transaction input nSequence's Disable Flag is set; or
      > the relative lock-time type is not the same as that of the corresponding input's nSequence; or
      > the value corresponding to the least significant 16 bits of top stack item is greater than that corresponding to the Bitcoin transaction input's nSequence. Note that the logic for this requirement is similar to that of nLockTime and OP_CLTV i.e., a valid Bitcoin transaction must have valid UTXOs and as a result, a UTXO unlocking time must not be greater than the one specified by the transaction.

   Once OP_CSV is executed successfully, its argument remains on the stack. In some cases it may need to be dropped to ensure proper subsequent execution. As a result, it is common to see CSV scripts paired with **OP_DROP**.

Locktime options (Summary)

APPLICABLE DIMENSIONS			APPLICABLE FIELD OR OPCODE
Location	**Nature**	**Type**	
Transaction	Absolute	Block Height	**nLockTime** with argument vaue < **500 million**
Transaction	Absolute	Time	**nLockTime** with argument value **>= 500 million**
UTXO	Absolute	Block Height	**OP_CLTV** with argument value **< 500 million**
UTXO	Absolute	Time	**OP_CLTV** with argument vaue **>= 500 million**
Transaction	Relative	Block Height	**nSequence** with argument such that **Disable Flag = 0**, and **Type Flag = 0**
Transaction	Relative	Time	**nSequence** with argument such that **Disable Flag = 0**, and **Type Flag = 1**
UTXO	Relative	Block Height	**OP_CSV** with argument such that **Disable Flag = 0**, and **Type Flag = 0**
UTXO	Relative	Time	**OP_CSV** with argument such that **Disable Flag = 0**, and **Type Flag = 1**

- Last but not least, there is one special opcode that whenever invoked, turns a Bitcoin transaction into an invalid one. This is the **OP_RETURN** opcode with decimal representation 106 (0x6a in hex). This opcode is particularly useful for adding data onto the blockchain as we will see later in section 6.

4 Examples of locking scripts (scriptPubKey)

In this section we introduce a number of scriptPubKeys or locking scripts along with their corresponding scriptSigs or unlocking scripts to illustrate how encumbrances are typically encoded. Clearly, this is but a small subset of the universe of possible locking conditions that could be devised in Script. We will go over the following scriptPubKey examples:

1. Pay-To-Public-Key also referred to as P2PK.
2. Pay-To-public-Key-Hash also referred to as P2PKH.
3. Pay-To-Multiple-Signature also referred to as P2MS.
4. Freezing funds using OP_CLTV.
5. Trustless payment for publishing data using OP_CLTV and conditional flow.
6. Escrow functionality with timeout using OP_CSV and conditional flow.
7. Pay-To-Script-Hash also referred to as P2SH.

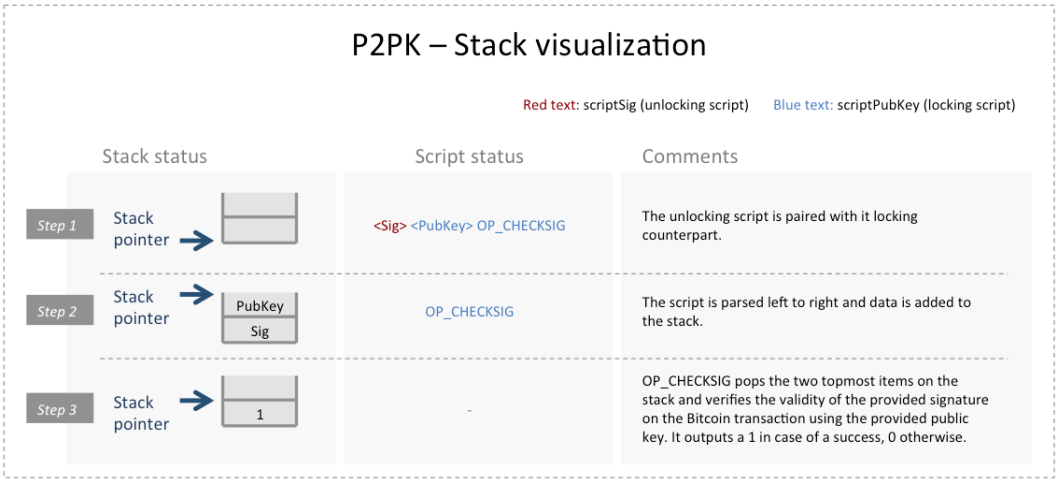
Before going into each of these examples in detail, note that by convention, we enclose in angle brackets (i.e., $< >$) any data that needs to be pushed onto the stack. In other words, brackets correspond to an appropriate data push opcode i.e., OP_PUSHBYTESi ($i \in \{1, .., 75\}$) or OP_PUSHDATAj ($j \in \{1, 2, 4\}$).

1. **P2PK**: In a Pay-To-Public-Key locking script, a UTXO is tied to a particular public key. Only the owner of the private key corresponding to this public key

could unlock its Satoshis and transfer them at will. In Bitcoin's Script, a P2PK scriptPubKey takes the following form:

<PubKey> OP_CHECKSIG

In order to unlock it, the legitimate owner must provide a signature that can be verified using that public key. The unlocking process proceeds as follows:



The first ever Bitcoin transaction that Satoshi Nakamoto initiated to sent BTC 10 to Hal Finney had P2PK scriptPubKeys for both its outputs. Its txid is:

0xf4184fc596403b9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16

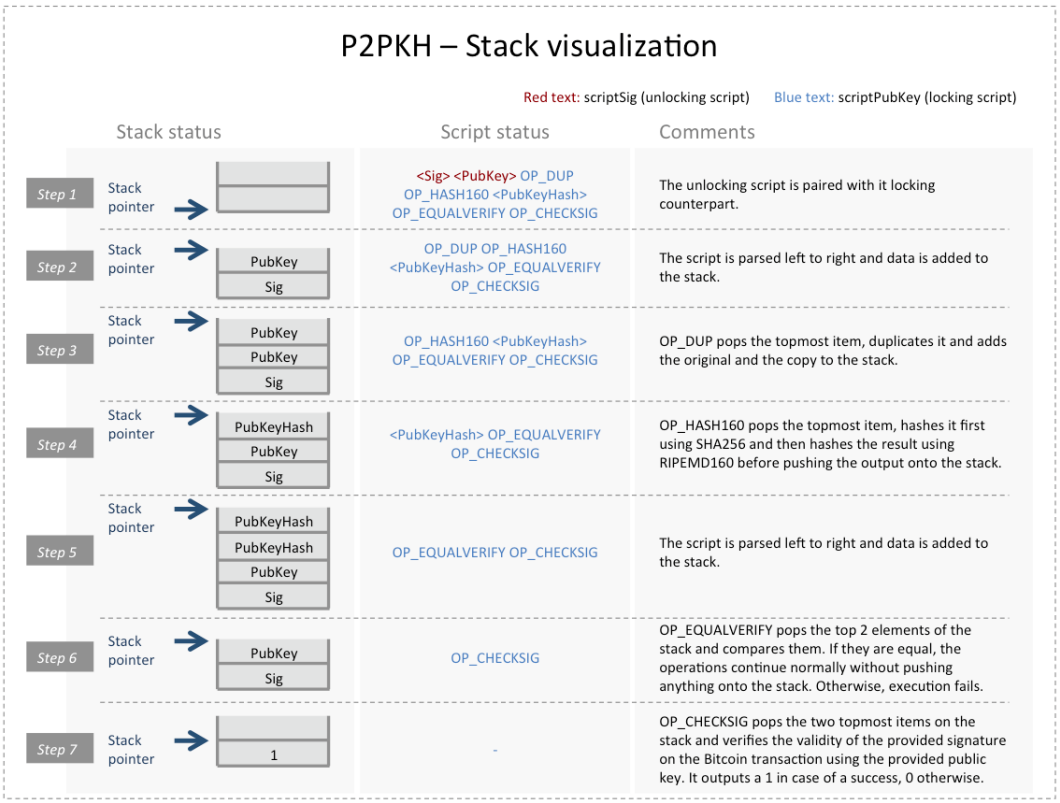
P2PK scriptPubKeys are now considered obsolete. They were used in Bitcoin's early days and have been replaced by the more cost-efficient and secure P2PKH:

- As we will shortly see, P2PKH is more cost-efficient since it uses a 20 byte long hashed construct as opposed to the 65 byte uncompressed or 33 byte compressed public key. A reduction in the overall byte size of a Bitcoin transaction brings its cost down.
- P2PKH is thought to be more secure because its usage of a hashed construct does not expose the public key of the sender prior to broadcasting the Bitcoin transaction. To better appreciate this, note that if the Elliptic Curve Cryptography Discrete Logarithm Problem were to be solved (through e.g., future quantum computations), then one would be able to derive the private key from its public counterpart and claim the relevant Satoshis. On the other hand, the SHA256 hash function is not known to be reducible to a computationally hard problem and is thought to be quantum resistant.

2. **P2PKH**: A P2PKH locking script has a similar purpose as P2PK. A UTXO gets tied to a particular public key and only the owner of the corresponding private key can spend it. The proof of legitimate ownership is however conducted in a different way. In Bitcoin's Script, a P2PKH scriptPubKey takes the following form:

OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

In order to unlock it, the legitimate owner must provide a valid signature and the appropriate public key. The unlocking process proceeds as follows:



The Bitcoin transaction example that we introduced in section 2 has P2PKH scriptPubKeys associated with its two outputs. Its txid is:

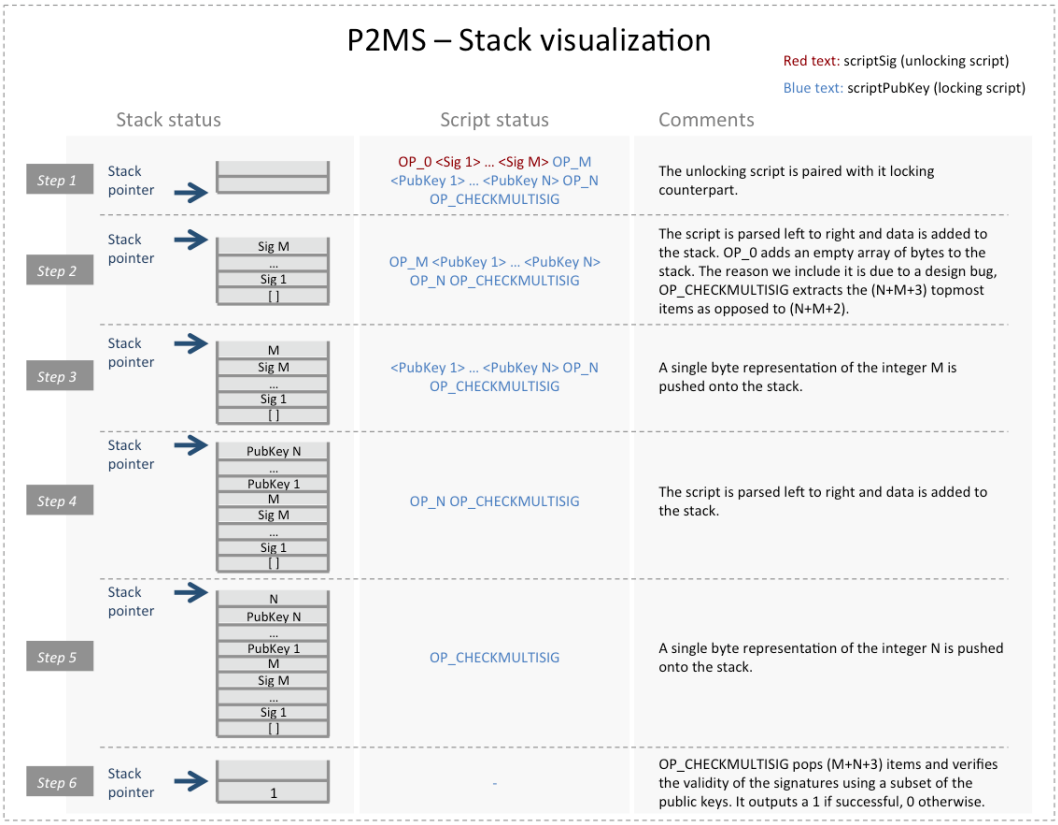
0xdbebe45e62370aeab972a9bbbee80f99febe6c904fe49b68efe7cc877a6cfd73

Note that in a P2PKH locking script, only the hash of the public key is made explicit. This stands in contrast to a P2PK locking script where the actual public key is fully revealed. Recall that it is this difference that confers to a P2PKH scriptPubKey its security and cost advantages over P2PK.

3. **P2MS**: A P2MS locking script can be thought of as a generalization of P2PK. Instead of encumbering the output by one public key, it gets encumbered by at least two public keys. A P2MS scriptPubKey takes the following form:

   OP_M <PubKey 1> ... <PubKey N> OP_N OP_CHECKMULTISIG

   In order to unlock it, M out of N (where $M \leq N$) private keys must each provide a valid signature. The unlocking process proceeds as follows:



Here is the txid of a Bitcoin transaction whose output's locking script is a 2 of 3 P2MS:

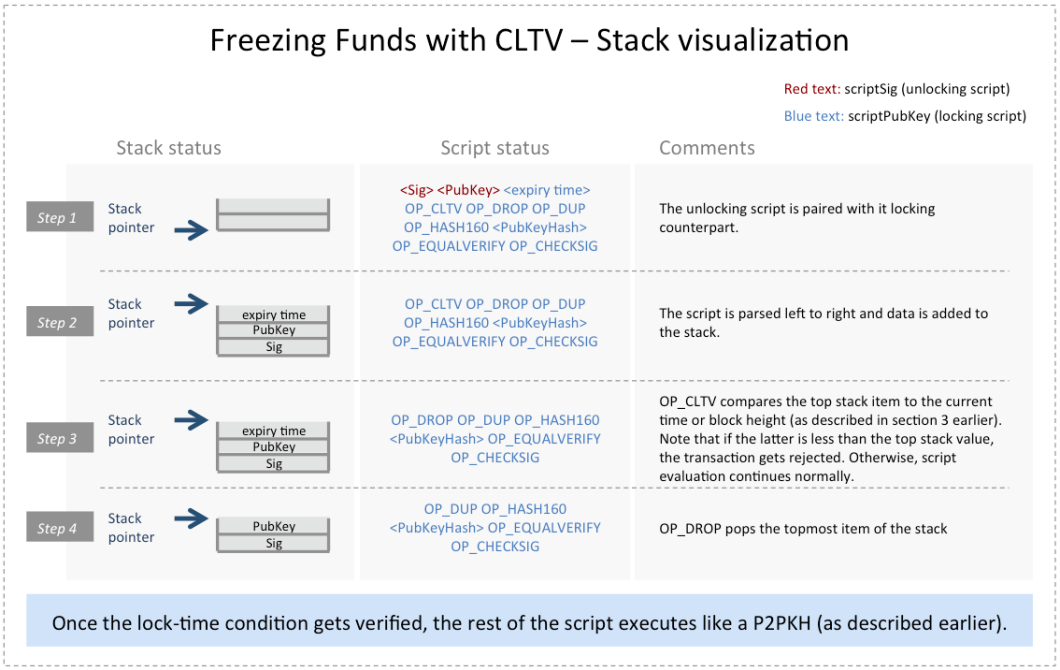
0x581d30e2a73a2db683ac2f15d53590bd0cd72de52555c2722d9d6a78e9fea510

P2MS locking scripts are not used anymore and have been replaced by the shorter and more secure Pay-To-Script-Hash (P2SH) version that we will introduce at the end of this section.

4. **Freezing funds using CLTV**: Bitcoin's Script offers the possibility of making a Bitcoin transaction output unspendable until a future date or block height. To do so, one can make use of OP_CLTV in the following scriptPubKey:

   <expiry time> OP_CLTV OP_DROP OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

   To unlock it, the legitimate owner must first wait for the expiry date to be reached and then provide an adequate signature and public key. The unloking process proceeds as follows:



5. **Trustless payment for publishing data using CLTV and conditional flow**: Purchasing content in physical form (e.g., books, magazines) when the seller and the buyer are co-located is a relatively straightforward process. In big part, this is due to the natural resolution of two distinct trust-related problems:

   (a) Providing the assurance to the buyer that the desired content is available.

   (b) Providing the assurance to the seller that payment will be received in exchange of releasing the content and to the buyer that the content will be accessible upon payment.

   Addressing these concerns in the context of data in digital form when the seller and buyer are not co-located is more challenging. Typical resolution mechanisms involve trusting a "neutral" third party acting as an escrow provider. A question of both practical and philosophical relevance is whether these concerns could be addressed trustlessly. Cryptography and Bitcoin Script help answer affirmatively.

   - **Proving availability of the desired content**: Different solutions have been proposed to address this problem. Peter Todd's **Paypub** protocol displays to the potential buyer a random subset of the digital content that she wishes to purchase. For a large enough subset, this provides acceptable assurance in the availability of the required data. For more details about the protocol, readers can refer to [19].

     Another resolution relies on the notion of a **Zero Knowledge Proof** (ZKP). ZKPs merit a separate post and are excluded from this note. For a brief introduction, the reader can refer to e.g., [6]. For our purposes, we limit ourselves to introducing them as:

     – Cryptographic constructs that allow a **Prover** to demonstrate her possession of a specific piece of knowledge to a **Verifier**.

- In addition, the proof must not leak any information to the Verifier including information pertaining to the possessed knowledge.
- Finally, the Verifier will assert the validity of the proof if and only if the Prover has the claimed knowledge.

The last two points imply that the Verifier will not be able to reproduce the proof. ZKP may seem counter-intuitive, most likely because we are used to mathematical proofs where the reasoning involved in proving or disproving a statement must be made public for it to be independently verified.

Sean Bowe built on an idea of Gregory Maxwell to implement the first ZKP in the context of a contingent payment Bitcoin transaction in 2016. The data purchased by Maxwell was a solution to a 16 by 16 Sudoku solved by Bowe. This ZKP was interactive i.e., involved initial data exchange between the Prover (Bowe) and the Verifier (Maxwell). Given:

- The Sudoku puzzle whose solution is desired,
- A 256-bit long hash value H,
- An encrypted answer E,

The Prover demonstrates to the Verifier that:

- They possess the answer A to the Sudoku puzzle,
- They possess the decryption key Dk which when applied to E outputs A,
- H is the SHA256 hash of Dk.

Following this interactive proof, Maxwell had overwhelming assurance that Bowe possessed the answer and the decryption key Dk. For more information, interested reader can consult [14], [8], and [9].

- **Ensuring that the payment takes place if and only if the desired data is released**: This is the part where Bitcoin Script lends a helping hand and allows one to devise a locking script that releases payment if and only if the desired data is made public. The following scriptPubKey achieves this by making use of conditional flow statements and of OP_CLTV:

```
OP_SHA256 <H> OP_EQUAL
OP_IF
    <Seller Pubkey>
OP_ELSE
    <future block height> OP_CLTV OP_DROP
    <Buyer Pubkey>
OP_ENDIF
OP_CHECKSIG
```

In order to unlock the funds, the seller must provide the correct Dk along with a valid signature. If the seller fails to provide the required Dk, the "OP_ELSE branch goes into effect causing funds to be returned to the buyer

after a pre-defined time if the buyer provided a valid signature. To better understand this, we consider these two cases in more detail:

Trustless Payment for Publishing Data - Stack visualization (Case 1: Seller's redemption)

Red text: scriptSig (unlocking script)
Blue text: scriptPubKey (locking script)

	Stack status	Script status	Comments
Step 1	Stack pointer →	<seller sig> <Dk> OP_SHA256 <H> OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	The unlocking script is paired with it locking counterpart. In this case, the seller provides a valid signature as well as the desired decryption key Dk whose SHA256 is H.
Step 2	Stack pointer → Dk, Seller sig	OP_SHA256 <H> OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	The script is parsed left to right and data is added to the stack.
Step 3	Stack pointer → H, Seller sig	<H> OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	OP_SHA256 pops the topmost element, computes its SHA256 and pushes the result to the stack.
Step 4	Stack pointer → H, H, Seller sig	OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	The script is parsed left to right and data is added to the stack.
Step 5	Stack pointer → 1, Seller sig	OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	OP_EQUAL pops the two topmost items and compares them. It pushes 1 if equal and 0 otherwise. In this case, we assume equality.
Step 6	Stack pointer → Seller Pubkey, Seller sig	OP_CHECKSIG	OP_IF pops the topmost item and if the value is true, the condition following it is executed.. In this case, the seller's pubkey is added.
Step 7	Stack pointer → 1	-	OP_CHECKSIG pops the two topmost items and verifies the the signature using the provided public key. It outputs a 1 in case of a success, 0 otherwise.

Trustless Payment for Publishing Data - Stack visualization (Case 2: Buyer's redemption)

Red text: scriptSig (unlocking script)
Blue text: scriptPubKey (locking script)

	Stack status	Script status	Comments
Step 1	Stack pointer →	<buyer sig> <Nk> OP_SHA256 <H> OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	The unlocking script is paired with it locking counterpart. In this case, the buyer provides a valid signature as well as an invalid decryption key Nk whose SHA256 is different than H.
Step 2	Stack pointer → Nk, buyer sig	OP_SHA256 <H> OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	The script is parsed left to right and data is added to the stack.
Step 3	Stack pointer → W, buyer sig	<H> OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	OP_SHA256 pops the topmost item, computes its SHA256 and pushes the result on the stack.
Step 4	Stack pointer → H, W, buyer sig	OP_EQUAL OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	The script is parsed left to right and data is added to the stack.
Step 5	Stack pointer → 0, buyer sig	OP_IF <Seller Pubkey> OP_ELSE <future block height> OP_CLTV OP_DROP <Buyer Pubkey> OP_ENDIF OP_CHECKSIG	OP_EQUAL pops the two topmost elements and compares them It pushes a 1 if equal and 0 otherwise. In this case, they are not equal.
Step 6	Stack pointer → block height, buyer sig	OP_CLTV OP_DROP <Buyer Pubkey> OP_CHECKSIG	OP_IF pops the top item and if true, condition is executed. In this case, OP_ELSE is executed.
Step 7	Stack pointer → future block height, buyer sig	OP_DROP <Buyer Pubkey> OP_CHECKSIG	OP_CLTV checks if the future block height is reached. If not, the program halts.
Step 8	Stack pointer → buyer sig	<Buyer Pubkey> OP_CHECKSIG	OP_DROP removes the top item on the stack.
Step 9	Stack pointer → buyer Pubkey, buyer sig	OP_CHECKSIG	The buyer's pubkey is added to the stack.
Step 10	pointer → 1	-	OP_CHECKSIG is executed.

We will look at an example of a Bitcoin transaction that implements such a locking script when we introduce P2SH a bit later in this section.

6. **Escrow functionality with timeout using CSV and conditional flow**: BIP112 [10] introduces a scriptPubKey that implements an escrow functionality that expires after a certain amount of relative time (relative to the UTXO creation time):

   OP_IF

   OP_2 <Alice's pubkey> <Bob's pubkey> <Escrow's pubkey> OP_3 OP_CHECKMULTISIG

   OP_ELSE

   <relative expiry term> OP_CHECKSEQUENCEVERIFY OP_DROP <Alice's pubkey> OP_CHECKSIG

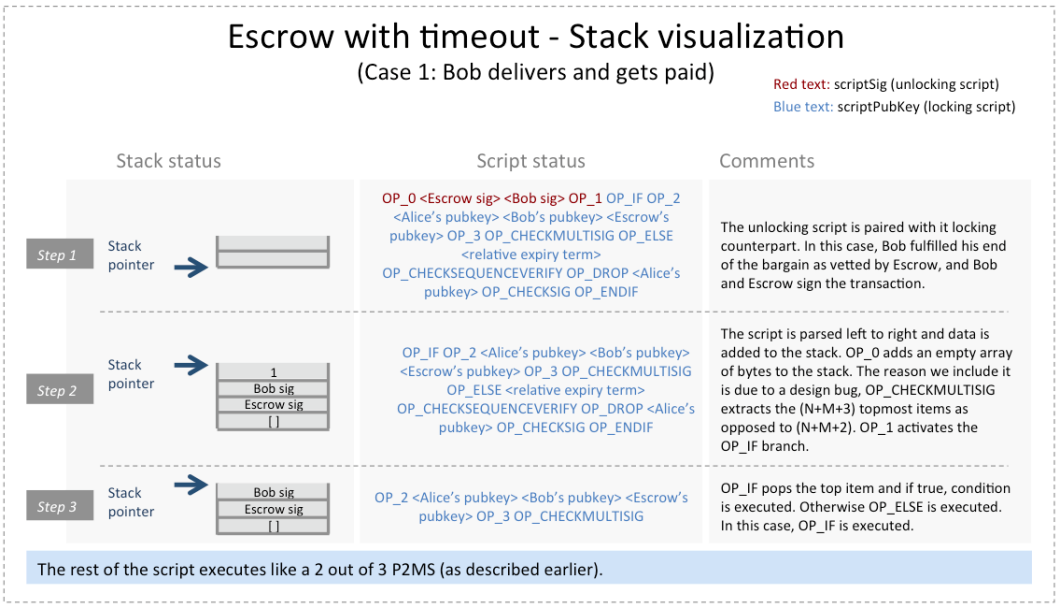
   OP_ENDIF

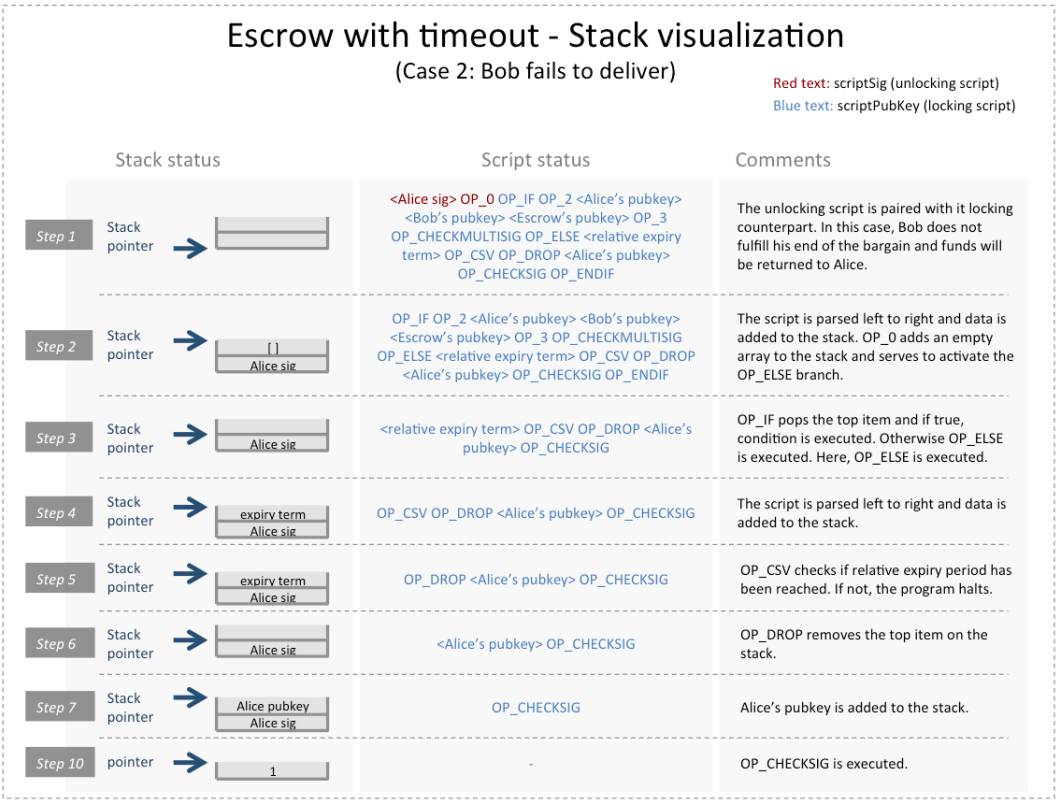
   This locking script involves three parties:

   (a) Alice which could represent e.g., a buyer of a service or good.
   (b) Bob which could represent e.g., a seller of a service or good.
   (c) Escrow, which acts as a third neutral party.

   For a certain period of time, any 2 of the 3 parties have the ability to unlock the funds and release them to the seller. For example, if Alice received the good(s) or service(s) from Bob within the pre-specified time but abstained from releasing the payment, Escrow and Bob can both unlock the transaction that pays Bob. On the other hand, if Bob fails to fulfill his end of the bargain within the specified time, the funds go back to Alice.

   The unlocking process proceeds as follows:





7. **P2SH:** Pay to Script Hash Bitcoin transactions accommodate a wide range of locking scripts. This class of transactions was introduced and formalized in BIP 16 [5]. Instead of asking the sender to supply all the spending conditions at the time of the transaction's creation, the new structure reduces this burden to a single encumbrance. Namely, a 20 byte hash that the receiver is required to reproduce at the time of redemption. The hash pre-image is a script with one or more spending conditions and is commonly referred to as **redeemScript**. More specifically, P2SH transactions involve the following steps:

   - The receiver(s) specify the spending conditions and generate an appropriate redeemScript.

   - The redeemScript gets mapped to a relevant Bitcoin address starting with the numeral 3. The mapping process was outlined in the chapter entitled *"Bitcoin Private Key, Public Key, and Addresses"*. As a reminder:
     - The redeemScript is serialized with all opcodes converted to binary.
     - The serialized output gets subjected to a HASH160 operation (i.e., a SHA256 followed by a RIPEMD160) resulting in a 20 byte hash.
     - The hash is prefixed with a byte whose hex representation is 0x05.
     - The resulting 21 byte string is appended with a 4 byte checksum.
     - The resulting 25 byte string is converted to Base58 to form the address.

   - The address is communicated to the sender who can subsequently issue a Bitcoin transaction with a single encumbrance, namely the requirement that the redeemer be able to reproduce the 20 byte hash of the redeemScript.

- When the redeemer reproduces the appropriate redeemScript, the latter gets evaluated to ensure proper observance of all its spending conditions.

The new structure minimizes the burden on the sender by shifting the need to supply the spending conditions from the sender to the redeemer. In Bitcoin's Script, a P2SH scriptPubKey takes the following form:

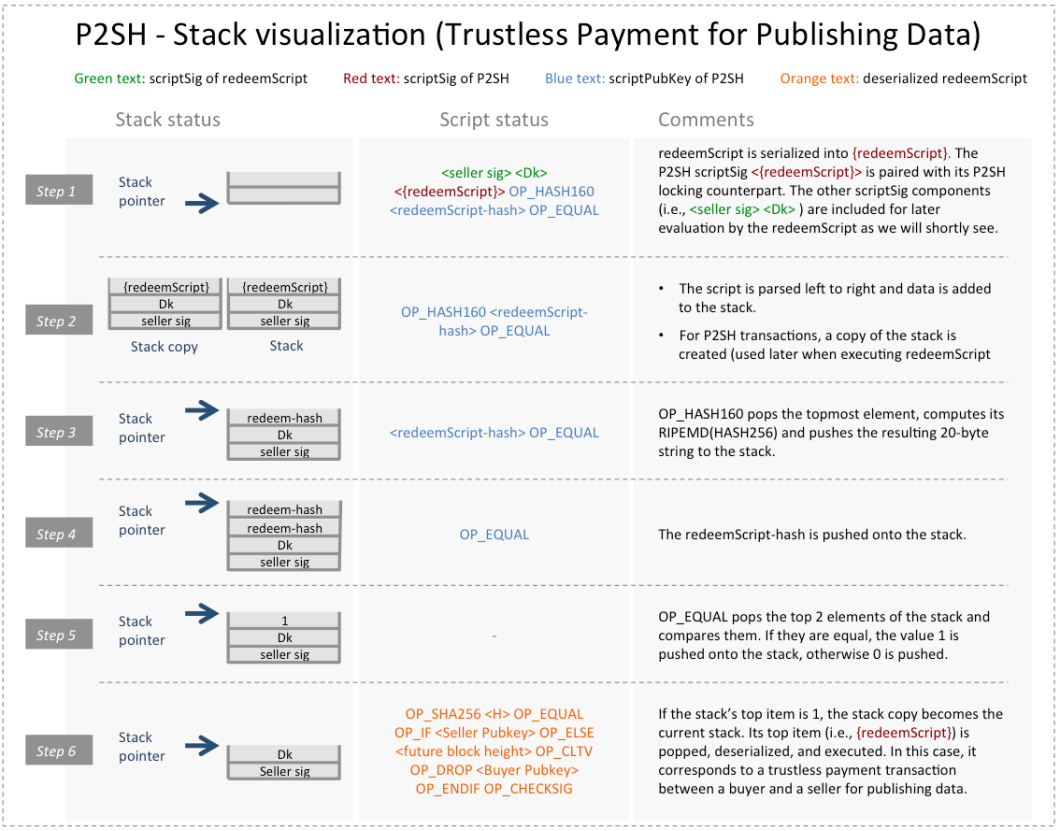
OP_HASH160 <20-byte-redeemScript-hash> OP_EQUAL

To unlock it, the owner(s) must provide a valid redeemScript in addition to any information required by the redeemScript's locking conditions. We illustrate this process for two distinct P2SH transactions:

(a) One where the redeemScript correpsonds to a seller unlocking funds in the context of a trustless payment for publishing data.

(b) One where the redeemScript correpsonds to an M-of-N multisignature.

Example 1: P2SH - Trustless Payment for Publishing Data:



For an example, one can consider the actual Bowe-Maxwell contingent payment Bitcoin transaction mentioned earlier. Its txid is:

0x200554139d1e3fe6e499f6ffb0b6e01e706eb8c897293a7f6a26d25e39623fae

It has a single input whose UTXO is locked using a P2SH script. The redeemScript can be read as:

OP_SHA256 OP_PUSHBYTES_32
5a917fe0e9a08004ea16bd682d656f8780b33af30c1e6d1b483652cecbb9d290
OP_EQUAL
OP_IF
OP_PUSHBYTES_33
0219b65599338687c784f5b78a23cac164a9094e94af6ec49532132da22d74e422
OP_ELSE
OP_PUSHBYTES_3 931b06 OP_CLTV OP_DROP
OP_PUSHBYTES_33
02cff5fe1dae742d57cf2be42ae607f28ae0e4837019ca4b8b1bcdc96bcf4af9ee
OP_ENDIF
OP_CHECKSIG

We see that:

- The hash H of the decryption key Dk is:

  0x5a917fe0e9a08004ea16bd682d656f8780b33af30c1e6d1b483652cecbb9d290

- The seller's public key is:

  0x0219b65599338687c784f5b78a23cac164a9094e94af6ec49532132da22d74e422

- The future block height is three bytes long and given by **0x931b06** in little endien notation. This corresponds to a decimal value of $400,275$.

- The buyer's public key is:

  0x02cff5fe1dae742d57cf2be42ae607f28ae0e4837019ca4b8b1bcdc96bcf4af9ee

By serializing this redeemScript (i.e., converting all opcodes to their corresponding byte-representation) we find that {redeemScript} is:

0xa8205a917fe0e9a08004ea16bd682d656f8780b33af30c1e6d1b483652cecbb9d290876
3210219b65599338687c784f5b78a23cac164a9094e94af6ec49532132da22d74e42267
03931b06b1752102cff5fe1dae742d57cf2be42ae607f28ae0e4837019ca4b8b1bcdc96b
cf4af9ee68ac

Subjecting this string to SHA256 followed by RIPEMD160 we obtain the following 20-byte hash:

0xecc23533aa4b1c12421c05bcd11abe181b3f4515

This result matches the redeemScript hash that appears in the P2SH locking

script. After this validation, the actual redeemScript gets executed. Note that the scriptSig associated with this redeemScript can be read from the Bitcoin transaction as:

- Seller's signature: 0x3044022060402771d8339b3bd09028c574912f09395f94a8 aa6e58b4d056605efcdc1c2802206cc94da4c9aeecee402ad5c740a5027cb9312a 33566b13fdc35f07086491ed6b01

- Decryption key Dk :

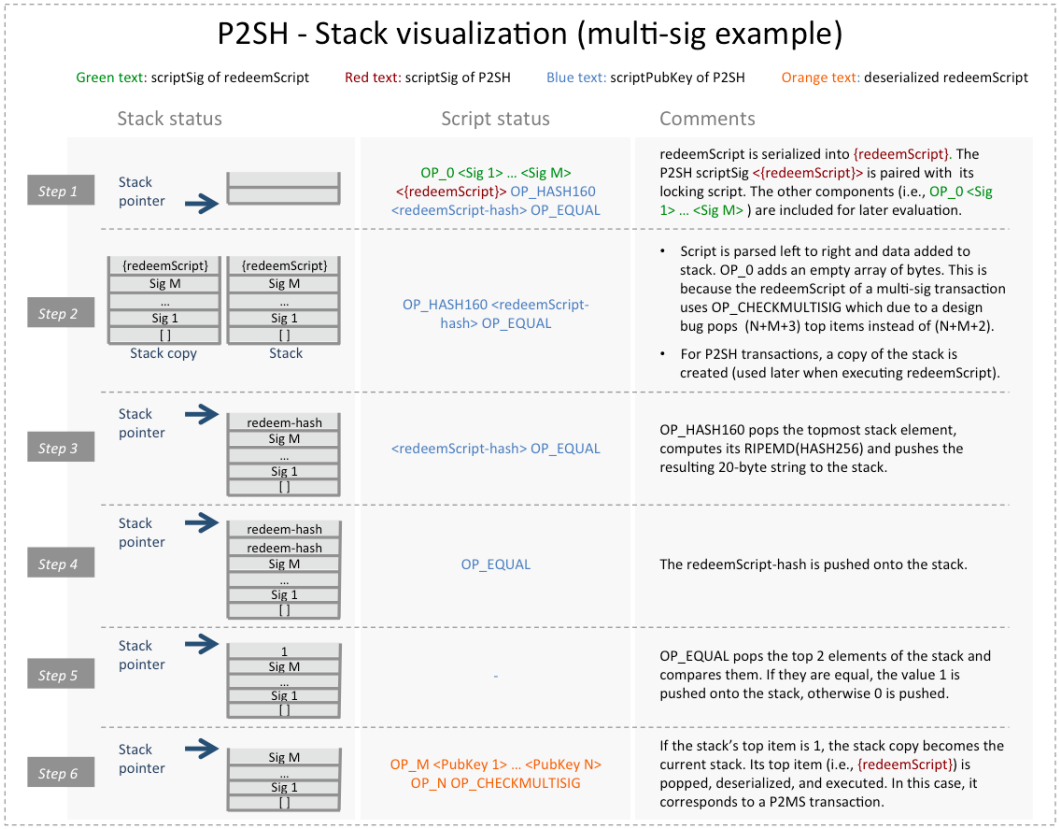
  0xbcf548de9614da26651fec6b48a696b0afc123a2b2d96449635ec92f58a5d882

Finally, note that the SHA256 of Dk evaluates to:

0x5a917fe0e9a08004ea16bd682d656f8780b33af30c1e6d1b483652cecbb9d290

which matches the value H provided in the redeemScript. The "OP_IF" branch of the locking script is then executed, the signature provided by the seller matched against his public key and upon verification, the funds transferred to the seller.

Example 2: P2SH - Multisignature example:



Note that {redeemScript} is pushed onto the stack using a PUSHDATA opcode. We have previously seen in section 3 that the maximum amount of bytes that a PUSHDATA operator can accommodate is 520 bytes. As a result, and in order to

ensure backward compatibility, {redeemScript} must not exceed 520 bytes. For the specific case of an M-of-N multisignature P2SH Bitcoin transaction where each public key is 33 bytes long in compressed format, this translates to a maximum N of 15 public keys. To see why, note that the {redeemScript} of an M-of-15 multisignature will be 513 bytes long, while that of an M-of-16 will be 547 bytes long because:

- OP_M consumes 1 byte
- Each public key requires 34 bytes (1 byte prefix $\in$ {0x02, 0x03}, and the 33 bytes of content). 15 public keys result in a total of 510 bytes, while 16 public keys result in 544 bytes $>$ 520.
- OP_N consumes 1 byte
- OP_CHECKMULTISIG consumes 1 byte.

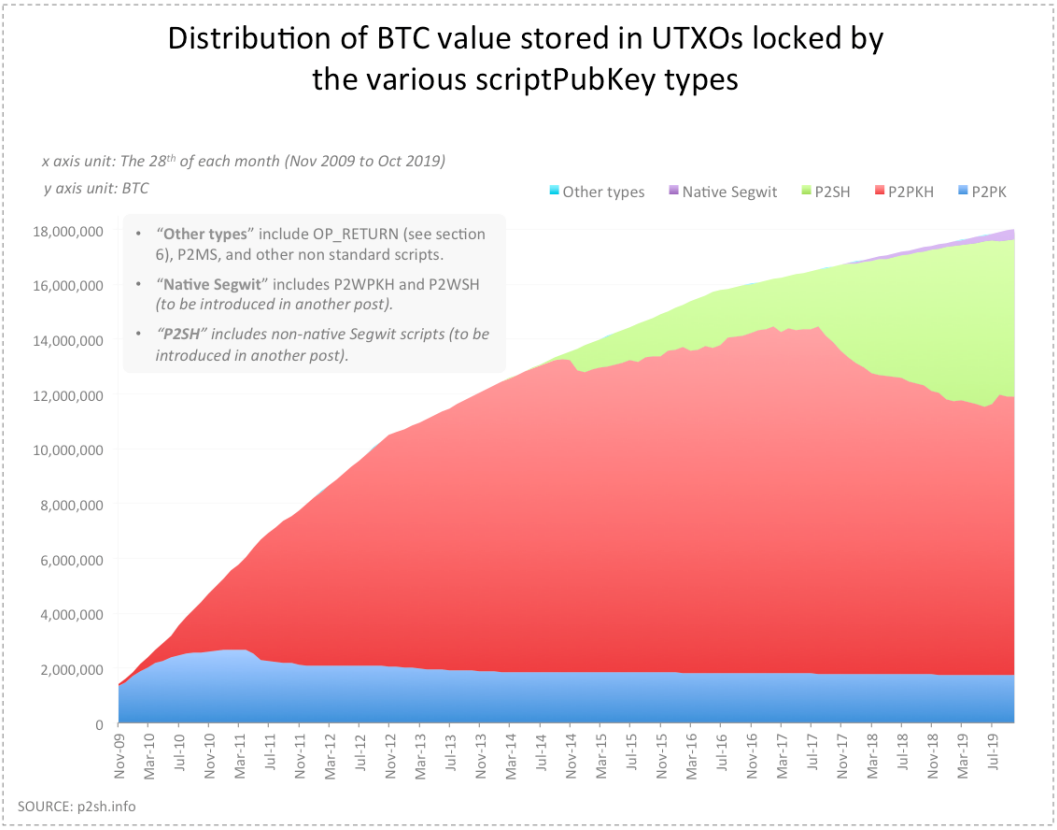
As an example, one can refer to the 2-of-3 multisignature P2SH Bitcoin transaction with txid:

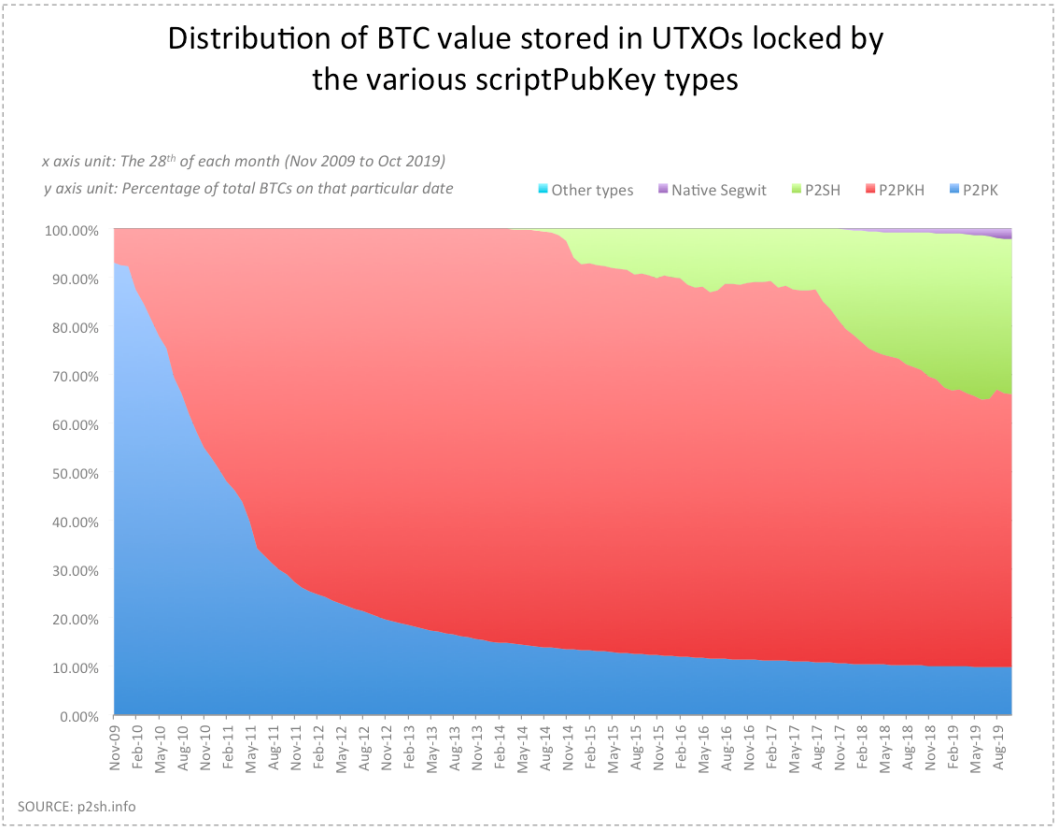
0xeeab3ef6cbea5f812b1bb8b8270a163b781eb7cde10ae5a7d8a3f452a57dca93

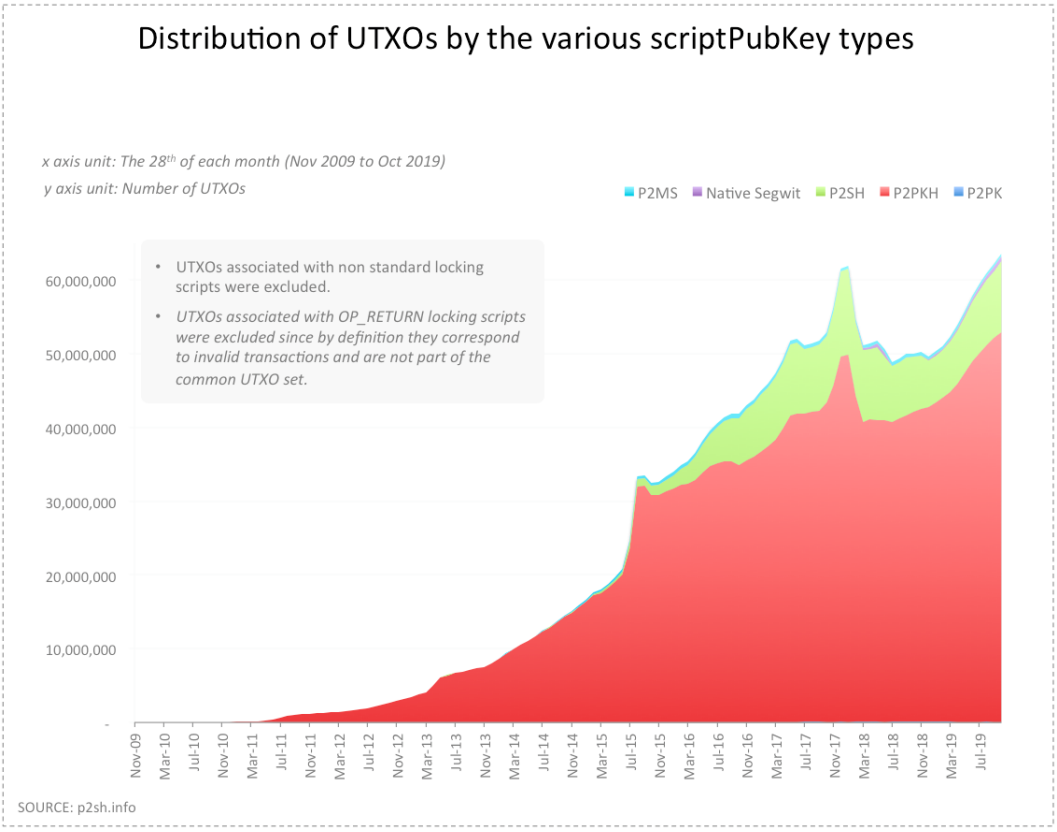
Irrespective of the length and complexity of the redeemScript, the scriptPubKey of a P2SH Bitcoin transaction is always 23 byte long (1 byte for OP_HASH160, 1 byte for an appropriate PUSHDATA opcode preceding the 20 byte redeemscript hash to be pushed onto the stack, 1 byte for OP_EQUAL). As a result, a P2SH Bitcoin transaction benefits from a clear cost advantage. In addition, one could argue that it also offers a security advantage steming from not having to publish the details of the redeemScript (including e.g., public keys) at the time of issuing a Bitcoin transaction.

We conclude this section by reproducing some statistics sourced from [2]. The following charts display two sets of information:

1. The distribution over time of bitcoins stored in UTXOs locked by the various scriptPubKey types
2. The evolution over time of the total number of UTXOs associated with the various scriptPubKey types.


Distribution of BTC value stored in UTXOs locked by the various scriptPubKey types
x axis unit: The 28th of each month (Nov 2009 to Oct 2019)
y axis unit: BTC
Other types
Native Segwit
P2SH
P2PKH
P2PK
"Other types" include OP_RETURN (see section 6), P2MS, and other non standard scripts.
"Native Segwit" includes P2WPKH and P2WSH (to be introduced in another post).
"P2SH" includes non-native Segwit scripts (to be introduced in another post).
18,000,000
16,000,000
14,000,000
12,000,000
10,000,000
8,000,000
6,000,000
4,000,000
2,000,000
0
Nov-09
Mar-10
Jul-10
Nov-10
Mar-11
Jul-11
Nov-11
Mar-12
Jul-12
Nov-12
Mar-13
Jul-13
Nov-13
Mar-14
Jul-14
Nov-14
Mar-15
Jul-15
Nov-15
Mar-16
Jul-16
Nov-16
Mar-17
Jul-17
Nov-17
Mar-18
Jul-18
Nov-18
Mar-19
Jul-19
SOURCE: p2sh.info


Distribution of BTC value stored in UTXOs locked by the various scriptPubKey types
x axis unit: The 28th of each month (Nov 2009 to Oct 2019)
y axis unit: Percentage of total BTCs on that particular date
Other types
Native Segwit
P2SH
P2PKH
P2PK
100.00%
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
Nov-09
Feb-10
May-10
Aug-10
Nov-10
Feb-11
May-11
Aug-11
Nov-11
Feb-12
May-12
Aug-12
Nov-12
Feb-13
May-13
Aug-13
Nov-13
Feb-14
May-14
Aug-14
Nov-14
Feb-15
May-15
Aug-15
Nov-15
Feb-16
May-16
Aug-16
Nov-16
Feb-17
May-17
Aug-17
Nov-17
Feb-18
May-18
Aug-18
Nov-18
Feb-19
May-19
Aug-19
SOURCE: p2sh.info



5 A closer look at transactions spending P2PKH or P2SH-MS outputs

In this section, we look more closely at two of the most common types of Bitcoin transactions: 1) Transactions that spend a P2PKH output, and 2) Transactions that spend a P2SH-MS output.

In order to better appreciate their underlying mechanisms, we dedicate the bulk of this section to building python scripts that help, among other things, to serialize and deserialize them:

- First, we introduce a set of three helper methods to parse different hexadecimal strings. They include method **parse_Hexstr**, method **parse_Elem** and method **parse_Varint**.
- Next, we introduce the **scriptPubKey_Type** method that checks if a given locking script is of the P2PKH type, P2SH type, or neither.
- We then build the **parse_ScriptSig_P2PKH** and **parse_ScriptSig_P2MS** methods that respectively check if an unlocking script corresponds to a P2PKH or to a P2SH-MS scriptPubKeys and in the process, extract relevant information. We limit ourselves to these two types and do not cover other cases.
- Lastly, we define the **BTC_TX** class that allows us to instantiate a Bitcoin transaction object inclusive of all its attributes. Within it, we define a number of methods including **serialize** and **deserialize**.

We point out that the code in this section is meant for educational purposes only. It does not cover the full spectrum of allowable Bitcoin transactions and is hence limited in scope. Moreover, the code has not been optimized for performance or size but rather for ease of illustrating the various building blocks of a Bitcoin transaction.

First, we define a subset of useful opcodes and their corresponding byte representation:

```
OP_0 = '00'
OP_PUSHDATA1 = '4c'
OP_PUSHDATA2 = '4d'
OP_PUSHDATA4 = '4e'
OP_1 = '51'
OP_DUP = '76'
OP_HASH160 = 'a9'
OP_EQUAL = '87'
OP_EQUALVERIFY = '88'
OP_CHECKSIG = 'ac'
OP_CHECKMULTISIG = 'ae'
```

Next, we introduce a set of three helper methods to parse various strings and elements. These methods are slightly modified versions of the ones presented in [15]:

- Method **parse_Hexstr(str_hex, index, length)** takes a hexadecimal string **str_hex**, an **index** value denoting where to start the parsing and the **length** of the sub-string to be extracted. It returns the extracted sub-string and an updated index pointing to the end of the sub-string.

```
def parse_Hexstr(str_hex, index, length):
    return str_hex[index:index+length], index+length
```

- Method **parse_Elem(tx, length)** takes a Bitcoin transaction object **tx** and extracts a sub-component of a given **length** from tx's serialized representation. It also updates the **offset** attribute of tx.

```
def parse_Elem(tx, length):
    element = tx.hex[tx.offset:tx.offset + length * 2]

    tx.offset += length * 2
    return element
```

- Method **parse_Varint(tx)** takes a Bitcoin transaction object **tx** and extracts a sub-component of variable length as per the varint encoding rules described in section 2. It returns two outputs. The first is the string of bytes containing the length value inclusive of the adequate prefix. The second output is the same as the first but exclusive of the prefix.

```python
def parse_Varint(tx):
    data = tx.hex[tx.offset:]                          # Move 'offset' of tx
    assert (len(data) > 0)

    varl_len = int(data[:2], 16)                       # Convert 1st byte todecimal
    assert (varl_len <= 255)

    if   varl_len <= 252:   storage_len = 1
    elif varl_len == 253:   storage_len = 3            # prefix byte and next 2 bytes
    elif varl_len == 254:   storage_len = 5            # prefix byte and next 4 bytes
    elif varl_len == 255:   storage_len = 9            # prefix byte and next 8 bytes
    else:
        raise Exception("Wrong input data size")

    varint = data[:storage_len * 2]

    if (storage_len == 1): length = varint             # Assign to 'length' the number
    else: length = data[2:storage_len * 2]             # of data bytes to read

    tx.offset += storage_len * 2                       # Update the tx's offset

    return varint, length
```

Before moving onto building the **BTC_TX** class, we introduce three methods to parse and extract relevant information from scriptPubKey and scriptSig fields.

Method **scriptPubKey_Type(hexstr)** takes a hexadecimal string and compares its structure to that of a P2PKH and P2SH scriptPubKeys. It outputs a string describing whether it is either of these types or neither:

```python
def scriptPubKey_Type(hexstr):
    if ((hexstr[:6] == OP_DUP + OP_HASH160 + '14')
        and (hexstr [46:] == OP_EQUALVERIFY +\
        OP_CHECKSIG) and (len(hexstr) == 50)):
        return "P2PKH"

    elif ((hexstr[:4] == OP_HASH160 + '14')
        and (hexstr [44:] == OP_EQUAL
        and (len(hexstr) == 46))):
        return "P2SH"

    else: return "Other type"
```

The following method **parse_ScriptSig_P2PKH(hexstr)** takes a hexadecimal string and parses it against the expected scriptSig of a P2PKH locking script. We know from section 4 that it should be of the form <Sig><PubKey> . The method returns a two-tuple consisting of the signature and the public key (both inclusive of their respective leading length bytes). In case the input is not a relevant scriptSig, the method outputs two empty arrays:

```
def parse_ScriptSig_P2PKH(hexstr):
    # Step 1: Extract signature-relevant data
    if (len(hexstr) < 2):
        return [],[]

    siglen = int(hexstr[:2], 16)*2
    sig, index = parse_Hexstr(hexstr, 0, siglen + 2)

    # Step 2: Extract public-key-relevant data
    if (index < len(hexstr) - 2):
        pubklen = 2 * int(hexstr[index:index + 2], 16)
        pubk, index = parse_Hexstr(
                          hexstr, index, pubklen + 2)
        if (pubk[2:4] in ['02', '03', '04'] and
                          (index == len(hexstr))):
            return [sig], [pubk]

    return [],[]
```

In Step 1, the input string's length is tested first. In case it is not long enough to accommodate a byte (i.e., 2 nibbles) containing the length of the signature, then the scriptSig cannot correspond to a P2PKH scriptPubKey. Otherwise, it proceeds to extract the length of the signature (expressed in units of nibbles) and store it in variable **siglen**. It then invokes the previously introduced **parse_Hexstr** method to extract the DER-encoded signature (inclusive of its leading length byte) and store it in variable **sig**.

In Step 2, the method conducts another test on the string's length. If the remaining part of the string is not long enough to accommodate a byte containing the length of the public key, the method returns two empty arrays. Otherwise, the method proceeds to extract the length of the public key (expressed in units of nibbles) and store it in variable **pubklen**. It then invokes the **parse_Hexstr** method one more time to extract the public key (inclusive of its leading length byte) and store it in variable **pubk**. Subsequently, the method conducts a check to ensure that the first byte following the public key's length byte correspond to an acceptable prefix (i.e., '0x02', '0x03', '0x04').

Finally, the method verifies that the end of the string has been reached.

Method **parse_ScriptSig_P2MS(hexstr)** takes a hexadecimal string and parses it against the scriptSig of a P2SH-MS scriptPubKey of the form:

OP_0 <Sig_1>..<Sig_M><{redeemScript}>

{redeemScript} is the serialized version of the redeemScript. In this case it is given by:

OP_M <PubKey_1>..<PubKey_N> OP_N OP_CHECKMULTISIG

The method returns a 4-tuple consisting of:

1. A string indicating whether the P2SH redeemScript is of type multisignature.
2. An array of the provided signatures.
3. The serialized redeemScript.
4. An array of the public keys.

```
def parse_ScriptSig_P2MS(hexstr):
    signature, pubkey =[], []
    delimiter = ''

    if (len(hexstr)<3): return "",[],"",[]

    op_0, index = parse_Hexstr(hexstr, 0, 2)
    if (op_0 != OP_0): return "-Not_MS",[],"",[]

    else:
        # Step 1: Extract signature-relevant data
        while (delimiter not in [OP_PUSHDATA1, OP_PUSHDATA2,
                OP_PUSHDATA4] and (index < len(hexstr) - 2)):
            siglen = 2 * int(hexstr[index : index + 2],16)
            sig, index = parse_Hexstr(hexstr,index,siglen + 2)
            signature.append(sig)
            delimiter = hexstr[index: index + 2]

        # Step 2: Extract the redeemScript data
        assert(index < len(hexstr) - 2)
        delimiter, index = parse_Hexstr(hexstr, index, 2)
        assert(delimiter in [OP_PUSHDATA1, OP_PUSHDATA2,
                                                OP_PUSHDATA4])
        index = parse_Hexstr(hexstr, index,
            2**(1+int(delimiter,16)-int(OP_PUSHDATA1,16)))[1]
        redeemscript = hexstr[index:]

        # Step 3: Extract pubkeys from redeemScript
        op_m, index = parse_Hexstr(hexstr, index, 2)
        assert ((1 <= int(op_m,16) - int(OP_1, 16) + 1 <= 15)
            and len(signature) == int(op_m,16) - int(OP_1,16)+1)

        assert(index < len(hexstr) - 2)
        pubklen = 2 * int(hexstr[index : index + 2], 16)
        while (hexstr[index+2 : index+4] in ['02','03','04']):
            pubk,index = parse_Hexstr(hexstr,index,pubklen+2)
            pubkey.append(pubk)
            assert (index <= len(hexstr) - 4)
            pubklen = 2 * int(hexstr[index : index + 2], 16)

        op_n, index = parse_Hexstr(hexstr, index, 2)
        assert ((op_m <= op_n)
            and (1 <= int(op_n,16) - int(OP_1, 16) + 1 <= 15)
            and (len(pubkey)==int(op_n, 16)-int(OP_1, 16)+1))

        op_msig, index = parse_Hexstr(hexstr, index, 2)
        assert (op_msig == OP_CHECKMULTISIG
            and (len(hexstr) == index))

        return "-MS", signature, redeemscript, pubkey
```

The method declares a **signature** array and a **pubkey** array to hold the M signatures and N public keys. It also declares a **delimeter** to mark the start of redeemScript in scriptSig.

It first checks that the length of the input is at least 2 nibbles for otherwise, it would not accommodate the leading OP_0 that marks the start of a multisignature scriptSig. It then calls **parse_Hexstr** and compares the extracted byte to OP_0.

In case of equality, the method proceeds to Step 1 to extract the signatures. It runs a while loop conditional on the value of **delimiter**. Recall that {redeemScript} is pushed onto the stack using a PUSHDATA opcode. As a result, after parsing each signature, the 2 nibble long delimiter is updated and tested against any of the PUSHDATA opcodes. In case of a match, the while loop exits marking the end of the signature sequence and the beginning of redeemScript. To avoid an infinite loop (i.e., if the delimiter never assumes an OP_PUSHDATA value) the method additionally checks that the current value of the index is not out of bound with respect to the string.

Step 2 extracts the redeemScript. The method first checks that the remaining part of the input is at least 2 nibbles long to contain the delimiter. It then checks if the first byte is a PUSHDATA opcode before invoking the **parse_Hexstr** method to adjust the index value. We have seen that with OP_PUSHDATAi, $i \in \{1, 2, 4\}$, the following i byte(s) contain the length of redeemScript. Hence the redeemScript content starts at the $(i+1)^{st}$ byte after the delimiter. The formula used to adjust the index is:

2**(1+int(delimiter,16) - int(OP_PUSHDATA1,16))

To justify it, recall that OP_PUSHDATA1, OP_PUSHDATA2, and OP_PUSHDATA4 have respective decimal representations 76, 77, and 78. Now observe that:

- OP_PUSHDATA1, yields $2^{(1+76-76)} = 2$, moving index by 2 nibbles as expected.
- OP_PUSHDATA2, yields $2^{(1+77-76)} = 4$, moving index by 4 nibbles as expected.
- OP_PUSHDATA4, yields $2^{(1+78-76)} = 8$, moving index by 8 nibbles as expected.

Step 3 parses redeemScript to extract the public keys. It extracts the first byte which should correspond to OP_M. It checks that the number of signatures equals M $\leq$ 15. Technically this limit applies when all keys are in compressed form (as discussed in section 4 earlier). Uncompressed keys dictate a lower value. The method then executes a while loop to extract the keys. After each extraction, it checks that the remaining string is long enough to accommodate at least 4 nibbles for OP_N and OP_CHECKMULTISIG.

Upon exiting the while loop, the method extracts the OP_N byte, checks that 1 $\leq$ M $\leq$ N $\leq$ 15 and that the number of public keys is N. Finally it verifies that the last byte corresponds to OP_CHECKMULTISIG.

Next, we build the **BTC_TX** class that allows the instantiation of a Bitcoin transaction object:

```
class BTC_TX:

    def __init__(self, net = ""):
        # Transaction input attributes
        self.inputs = None                      # Total number of inputs to current tx
        self.prev_tx_id = []                    # Array of previous txids
        self.prev_out_index = []                # Array of previous tx outputs indices
        self.scriptSig = []                     # Array of scriptSig of each tx input
        self.nSequence = []                     # Array of nSequence variables

        # Transaction output attributes
        self.outputs = None                     # Total number of outputs
        self.value = []                         # Array of values to each destination address
        self.scriptPubKey = []                  # Array of output script for each output

        # Other attributes
        self.version = None                     # Holds the version specifier
        self.nLockTime = None                   # Holds the locktime variable

        self.hex = ""                           # Holds the raw hex string of current tx
        self.offset = 0                         # Specifies index in raw hex string for
                                                # -- parsing purposes
        self.net = net                          # Specifies whether transaction is on
                                                # -- "mainnet"or "testnet"
```

This class has a total of nine methods:

- Method **deserialize(tx_serialized_hex)** takes a raw Bitcoin transaction and extracts all its attributes to be displayed later in a human readable format:

```
def deserialize(self, tx_serialized_hex):
    self.hex = tx_serialized_hex

    self.version = int(
        change_Endianness(parse_Elem(self, 4)), 16)

    self.inputs = int(
        change_Endianness(parse_Varint(self)[1]), 16)

    '''Input parameters deserialization'''
    in_count = 0
    while (in_count < self.inputs):
        self.prev_tx_id.append(
            change_Endianness(parse_Elem(self, 32)))

        self.prev_out_index.append(int(
            change_Endianness(parse_Elem(self, 4)), 16))

        scriptSiglen = int(
            change_Endianness(parse_Varint(self)[1]), 16)

        self.scriptSig.append(
            parse_Elem(self, scriptSiglen))

        self.nSequence.append(int(
            change_Endianness(parse_Elem(self, 4)), 16))

        in_count += 1

    ''''Output parameters deserialization'''
    self.outputs = int(
        change_Endianness(parse_Varint(self)[1]), 16)

    out_count = 0;
    while (out_count < self.outputs):
        self.value.append(int(
            change_Endianness(parse_Elem(self, 8)), 16))

        scriptPubKeylen = int(
            change_Endianness(parse_Varint(self)[1]), 16)

        self.scriptPubKey.append(
            parse_Elem(self, scriptPubKeylen))

        out_count += 1

    ''''Remaining parameters deserialization'''
    self.nLockTime = int(
        change_Endianness(parse_Elem(self, 4)), 16)

    assert(self.offset == len(self.hex))
    self.offset = 0

    return self
```

  The deserialization procedure introduced in section 2 is applied as follows:

  - The **parse_Elem(tx, length)** method extracts the first four bytes of the raw transaction. These bytes are passed to the **change_Endienness(x)** method (introduced in section 2) to convert them to big endian. The equivalent decimal representation is then stored in the **version** field. Note that the **parse_Elem(tx, length)** method updates the **offset** attribute of the Bitcoin transaction instance, ensuring as such proper parsing of the raw input.

  - The **inputs count** is a variable length field that is first parsed using the

parse_Varint(tx) method. The second output of the latter is the desired count value encoded in little endian. The **change_Endienness(x)** method subsequently converts it to big endian, and its decimal representation is stored in variable **inputs**. Here too, note that the **parse_Varint(tx)** method correctly updates the **offset** attribute of the transaction instance in order to allow proper parsing of the raw input.

- The method then iterates through the inputs, and for each one of them:
  * Extracts the 32 byte long txid, converts it to big endian and appends the result to the **prev_tx_id** array.
  * Extracts the 4 byte index of the appropriate output appearing in the transaction whose txid has just been extracted. The result is converted to big endian and its decimal value stored in the **prev_out_index** array.
  * Extracts the scriptSig length associated with the current input. Given its varint nature, the **parse_Varint(tx)** method is invoked and the desired result converted to big endian. The corresponding decimal value is stored in the variable **scriptSiglen**.
  * Uses **scriptSiglen** in order to appropriately parse the raw transaction, extract the **scriptSig** field, and append it to the **scriptSig** array.
  * Extracts the 4 byte nSequence number by invoking the **parse_Elem(tx, length)** method, converts the result to big endian and append its decimal representation to the **nSequence** array.
- The Bitcoin transaction **outputs count**, like its input counterpart is of type varint. The **parse_Varint(tx)** method extracts the count encoded in little endian. The result is converted to big endian and its decimal representation stored in variable **outputs**.
- The method then iterates over all outputs and for each one of them:
  * Extracts the 8 byte long Satoshi value, converts it to big endian and appends its decimal representation to the **value** array.
  * Extracts the scriptPubKey length associated with the current output. Given its varint nature, the **parse_Varint(tx)** method is invoked and the desired result converted to big endian. The corresponding decimal value is then stored in the variable **scriptPubKeylen**.
  * Uses **scriptPubKeylen** in order to appropriately parse the raw Bitcoin transaction, extract the **scriptPubKey** field, and append it to the **scriptPubKey** array.
- The method finally extracts the 4 byte **nLocktime** value, converts it to big

endian and stores its decimal representation.

- Since both methods **parse_Elem(tx, length)** and **parse_Varint(tx)** adjust the **offset** attribute of the transaction instance, successful deserialization should end with an offset value equal to the length of the raw transaction.

- Method **serialize()** acts on a Bitcoin transaction instance and outputs its raw hexadecimal representation after properly serializing its various attributes. The logical flow of the method is straightforward since it performs the reverse operations of the **deserialize()** method. Note that it makes various calls to the following two methods introduced earlier in section 2:
  - Method **int2bytes(a,b)** that converts integer **a** into its byte (base 256) representation such that the length of the byte representation is **b** bytes.
  - Method **encode_Varint(value)** that converts an integer **value** to its varint hexadecimal format expressed in little endian.

```
def serialize(self):
    serialized_tx = change_Endianness(
        int2bytes(self.version, 4))

    serialized_tx += encode_Varint(self.inputs)

    '''Input parameters serialization'''
    for i in range(self.inputs):
        serialized_tx += change_Endianness(
            self.prev_tx_id[i])

        serialized_tx += change_Endianness(
            int2bytes(self.prev_out_index[i], 4))

        serialized_tx += encode_Varint(
            len(self.scriptSig[i]) / 2)

        serialized_tx += self.scriptSig[i]

        serialized_tx += change_Endianness(
            int2bytes(self.nSequence[i], 4))


    ''''Output parameters serialization'''
    serialized_tx += encode_Varint(self.outputs)

    for i in range(self.outputs):
        serialized_tx += change_Endianness(
            int2bytes(self.value[i], 8))

        serialized_tx += encode_Varint(
            len(self.scriptPubKey[i]) / 2)

        serialized_tx += self.scriptPubKey[i]

    ''''Other parameters serialization'''
    serialized_tx += change_Endianness(
        int2bytes(self.nLockTime, 4))


    return serialized_tx
```

- **get_Prev_Tx_Deserialized()** acts on a Bitcoin transaction instance and returns

an array **prev_tx** of Bitcoin transaction objects. Each object is a deserialized version of a relevant previous Bitcoin transaction that has one of its outputs feeding into the current transaction.

```
def get_Prev_Tx_Deserialized(self):
    prev_tx = []

    for i in range(self.inputs):
        tx = BTC_TX(self.net)

        prev_tx_raw = get_Serialized_Tx(
            self.prev_tx_id[i], self.net)

        prev_tx.append(tx.deserialize(prev_tx_raw))

    return prev_tx
```

For each input in the current Bitcoin transaction, a new Bitcoin transaction object is instantiated. The **get_Serialized_Tx(txid, net)** method is then invoked on the txid referenced by the current input and the result stored in **prev_tx_raw**. The latter is then fed to the **deserialize()** method and the outcome appended to the **prev_tx** array.

- **get_Prev_Out_Type()** acts on a Bitcoin transaction instance and returns an array **prev_out_type** of string elements. Each string element corresponds to the type of transaction output (i.e., P2PKH, P2SH-Not_MS, P2SH-MS) being unlocked by the current input.

```
def get_Prev_Out_Type(self):
    prev_out_type = []
    prev_tx = self.get_Prev_Tx_Deserialized()

    for i in range(self.inputs):
        prev_scriptPubKey = prev_tx[i].scriptPubKey[
            self.prev_out_index[i]]

        prev_out_type.append(scriptPubKey_Type(
            prev_scriptPubKey))

        if (prev_out_type[i] == "P2SH"):
            prev_out_type[i] += parse_ScriptSig_P2MS(
                self.scriptSig[i])[0]

    return prev_out_type
```

It starts by calling the **get_Prev_Tx_Deserialized()** method and storing the resulting array in **prev_tx**. It then retrieves the scriptPubKey of the appropriate output in a previous Bitcoin transaction that the current input's scriptSig is meant to unlock. This is performed by noticing that input i of the current Bitcoin transaction is associated with previous transaction **prev_tx[i]** and that the index of the relevant output is **prev_out_index[i]**. As a result, the relevant locking script **prev_scriptPubKey** can be retrieved as follows:

prev_tx[i].scriptPubKey[self.prev_out_index[i]]]

The latter is then passed to method **scriptPubKey_Type(hexstr)** which returns a string indicating whether the scriptPubKey is of type "P2PKH",

"P2SH", or "Other type". If it is "P2SH", **parse_ScriptSig_P2MS(hexstr)** gets invoked and the first of its outputs (which can either be "-MS" or "-Not_MS") is appended to **prev_out_type[i]**.

- **decomp_ScriptSig()** acts on a Bitcoin transaction instance and returns an array of dictionaries **decomp_scriptSig**. The method decomposes the scriptSig of each input of a Bitcoin transaction into its components including signatures, public keys, and redeemScript if applicable.

```
def decomp_ScriptSig(self):
    decomp_scriptSig = []
    prev_out_type = self.get_Prev_Out_Type()

    for i in range(self.inputs):
        if (prev_out_type[i] == "P2PKH"):
            signature, pubkey = parse_ScriptSig_P2PKH(
                self.scriptSig[i])

            decomp_scriptSig.append(
                {"Signature": signature,
                "Public Key": pubkey})

        elif (prev_out_type[i] == "P2SH-MS"):
            signature, redeemscript, pubkey =\
                parse_ScriptSig_P2MS(self.scriptSig[i])[1:]

            decomp_scriptSig.append(
                {"Signature": signature,
                "Redeem Script": redeemscript,
                "Public Key": pubkey})

        else:
            decomp_scriptSig.append(
                {"Signature": '',
                 "Other data": ''})

    return decomp_scriptSig
```

It starts by invoking **get_Prev_Out_Type()** and then iterates over the Bitcoin transaction's inputs. Depending on the type of each input, it performs the following:

  - Invokes the **parse_ScriptSig_P2PKH(hexstr) method** on the input's **scriptSig** attribute in case it corresponds to a P2PKH locking script. It then stores the relevant signature and public key in a dictionary and appends it to the **decomp_scriptSig** array.

  - Invokes the **parse_ScriptSig_P2MS(hexstr) method** on the input's **scriptSig** attribute in case it corresponds to a P2SH-MS locking script. It then stores the relevant signature(s), redeemScript and public key(s) in a dictionary and appends it to the **decomp_scriptSig** array.

  - Otherwise, the method appends an empty dictionary to the **decomp_scriptSig** array.

- **display()** shows the attributes of the Bitcoin transaction in human readable form.

```
def display(self):
    print "{"

    print '\t"version": ' + str(self.version)

    print '\n\t"inputs":[ '
    for i in range(self.inputs):
        print "\t{"
        print '\t\t"prev_out":\n'
        print "\t\t{"
        print '\t\t\t"txid": ' + self.prev_tx_id[i]
        print '\t\t\t"index": ' + str(self.prev_out_index[i])
        print '\n\t\t\t\t"scriptSig":'
        print '\t\t\t\t', '\n\t\t\t\t'.join(textwrap.wrap(
            self.scriptSig[i], 68, break_long_words=True))

        decomp_scriptSig = self.decomp_ScriptSig()
        for key in decomp_scriptSig[i].keys():
            print '\n\t\t\t\t\t', key, ":"
            print '\t\t\t\t\t', '\n\t\t\t\t\t'.join(
                textwrap.wrap(str(decomp_scriptSig[i][key]),
                60, break_long_words=True))

        prev_out_type = self.get_Prev_Out_Type()
        print '\n\t\t\t"type": ' + prev_out_type[i]
        print "\t\t}\n"

        print '\t\t"sequence": ' + str(self.nSequence[i])
        print "\t}"

    print"\t]"

    print '\n\t"out":[ '
    for i in range(self.outputs):
        print "\t{"
        print '\t\t"value": ' + str(self.value[i]) + " satoshis"
        print '\t\t"scriptPubKey": ' + self.scriptPubKey[i]
        print "\t}"
    print"\t]"

    print '\n\t"nLockTime": ' + str(self.nLockTime)

    print "}"
```

The following three methods will be used in section 7 to construct the appropriate message to be signed based on the sighash value:

- **reset_Inputs()** acts on a Bitcoin transaction instance, sets the **input counter** to 0 and all other relevant input attributes to empty arrays.

```
def reset_Inputs(self):
    self.inputs = 0
    self.prev_tx_id = []
    self.prev_out_index = []
    self.scriptSig = []
    self.nSequence = []
```

- **reset_Outputs()** acts on a Bitcoin transaction instance, sets the **output counter** to 0 and all other relevant output attributes to empty arrays.

```
def reset_Outputs(self):
    self.outputs = 0
    self.value = []
    self.scriptPubKey = []
```

- **set_ScriptSig(index, hex_str)** acts on a Bitcoin transaction instance and sets

the **scriptSig** attribute of input #**index** to **hex_str**.

```
def set_ScriptSig(self, index, hex_str):
    self.scriptSig[index] = hex_str
    return self
```

To display the deserialized form of a Bitcoin transaction specified by its txid, we can proceed as follows:

```
tx = BTC_TX("mainnet")

txid = "039b145453739a8d58198eb9caa63eb9db9a8bb\
08cbd0977237e05082561a4a5"

tx_hex = get_Serialized_Tx(txid, "mainnet")

print "The raw hexadecimal transaction is: "
print '\n'.join(textwrap.wrap(tx_hex, 100, break_long_words=True));

print "\nThe deserialized transaction is: "
tx.deserialize(tx_hex)
tx.display()
```

For example, when applied to the Bitcoin transaction with txid:

0x039b145453739a8d58198eb9caa63eb9db9a8bb08cbd0977237e05082561a4a5

We get the following:

```
The raw hexadecimal transaction is:
0100000003f5d37b475141216a2682d7b1ca2339ddfce85bde721568498cb0db4665f0ed1901000000fc004730440220568e
2dce1f11b2db59de929fd1fead6b67585ea309fb6204f2451a4e19b4cc1702200ab5c60127dde33532718fef6a7773625b0d
4a03b441eee75e2f80de6e18ee6a014730440220408a2599e5daace10d148bbd1771c286b7adc60d9d5e1302babb9eb7c76f
f2c002204bbc3d9143cc98a74bfa2736d42e2d209926fb07fc65074a3e62e21f327362f5014c69522103ea74dfdbc1aa1689
c00b5319bb8ccbd3074cded17efc4e5046a0be026a5736ce210340614f30da78f213fbe4f0eab3efd187d57f8d2d6e0880ff
026bfe5d65fde7c62103583b06c1f8510f4a8da8c8babb867676722b323787d5c57974cfe31c8cf420df53aeffffffff320c
cdd3bded6eb3bb317a437fc6f611254da70a2fc1fb8b9a19bec1bd67bf9a01000000fc00473044022016402 77f58817c09db
76532f2af0f430e5bcb25b1b3d94dc9dcb823c73889e81022060cd7589983c8cf84958b366a2cbdf09b0343b630ff0f13326
6e2be87a2bc50e014730440220434bfb409312a376efac9f81e9ab99ba1a5c4274998eee634635971193c6f6a802200bf63b
a78274a666df7be21cc96bf1fd813b85bf1fc61774cc7822743507c8f7014c69522103b323ad56963c69076ee2479ed92e5b
092c579c9bae59bca564386e642347bf9c21023ef9051578518d3b2576637c958021121d2ea6af046bd0f64c5d63b020ff6a
9b2103c33a9ba944fd4636704e56d7cc0301978c6736facebbf90e0b8e692dc60f9d5b53aeffffffff1c74bf4636506268b
c3b38476dc126c9190c73c9bee5332bf47a7d413b7631501000000fdfd000048304502210 09f7d7c44eab7bfd3acd874f1c8
93152089cb0f9937798b3b6fdcc47521cc5434022044249ee3516f3108af9f6dfa7be0275f1abfa3531094ff80d4af8d20d0
6437610147304402 20355e92861b63fcf58c8e6028075d1559ffbd74d154ae90a11aea56e2e3e05958022075965c83ced37b
b942644772f2ddea466aeddaf4990620a441cce884b781c752014c69522102aba4b3bfdea7e4b709edcc56e609845d7eeba9
ffb0a813e9edd5b8729c57804f21033ad2088afe60726c746dac2f2dcfd747a47cd685122ca06635af860c00f690e321025d
5e665937e296a137c01afedd9a0c76e8e0f3f6ac3466a260d5d8e50baadb9c53aeffffffff0350185b00000000001976a914
410d9e4a1cf587d1707b402a986812780d16b62088ac6f5edd040000000017a91414328fd3a99e5666514cd5167d2cc4ebed
15c1a087d13e1c070000000017a91486744da8f363950f7c90378bdb1cc54336d158868700000000
```

```
The deserialized transaction is:
{
        "version": 1

        "inputs":[
        {
                "prev_out":

                {
                        "txid": 19edf06546dbb08c49681572de5be8fcdd3923cab1d782266a214151477bd3f5
                        "index": 1

                                "scriptSig":
                                004730440220568e2dce1f11b2db59de929fd1fead6b67585ea309fb6204f2451a4e
                                19b4cc1702200ab5c60127dde33532718fef6a7773625b0d4a03b441eee75e2f80de
                                6e18ee6a014730440220408a2599e5daace10d148bbd1771c286b7adc60d9d5e1302
                                babb9eb7c76ff2c002204bbc3d9143cc98a74bfa2736d42e2d209926fb07fc65074a
                                3e62e21f327362f5014c69522103ea74dfdbc1aa1689c00b5319bb8ccbd3074cded1
                                7efc4e5046a0be026a5736ce210340614f30da78f213fbe4f0eab3efd187d57f8d2d
                                6e0880ff026bfe5d65fde7c62103583b06c1f8510f4a8da8c8babb867676722b3237
                                87d5c57974cfe31c8cf420df53ae

                                        Redeem Script :
                                        522103ea74dfdbc1aa1689c00b5319bb8ccbd3074cded17efc4e5046a0be
                                        026a5736ce210340614f30da78f213fbe4f0eab3efd187d57f8d2d6e0880
                                        ff026bfe5d65fde7c62103583b06c1f8510f4a8da8c8babb867676722b32
                                        3787d5c57974cfe31c8cf420df53ae

                                        Public Key :
                                        [u'2103ea74dfdbc1aa1689c00b5319bb8ccbd3074cded17efc4e5046a0b
                                        e026a5736ce', u'210340614f30da78f213fbe4f0eab3efd187d57f8d2d
                                        6e0880ff026bfe5d65fde7c6', u'2103583b06c1f8510f4a8da8c8babb8
                                        67676722b323787d5c57974cfe31c8cf420df']

                                        Signature :
                                        [u'4730440220568e2dce1f11b2db59de929fd1fead6b67585ea309fb620
                                        4f2451a4e19b4cc1702200ab5c60127dde33532718fef6a7773625b0d4a0
                                        3b441eee75e2f80de6e18ee6a01', u'4730440220408a2599e5daace10d
                                        148bbd1771c286b7adc60d9d5e1302babb9eb7c76ff2c002204bbc3d9143
                                        cc98a74bfa2736d42e2d209926fb07fc65074a3e62e21f327362f501']

                        "type": P2SH-MS
                }

                "sequence": 4294967295
        }
        {
                "prev_out":

                {
                        "txid": 9abf67bdc1be199a8bfbc12f0aa74d2511f6c67f437a31bbb36eedbdd3cd0c32
                        "index": 1

                                "scriptSig":
                                0047304402201640277f58817c09db76532f2af0f430e5bcb25b1b3d94dc9dcb823c
                                73889e81022060cd7589983c8cf84958b366a2cbdf09b0343b630ff0f133266e2be8
                                7a2bc50e014730440220434bfb409312a376efac9f81e9ab99ba1a5c4274998eee63
                                4635971193c6f6a802200bf63ba78274a666df7be21cc96bf1fd813b85bf1fc61774
                                cc7822743507c8f7014c69522103b323ad56963c69076ee2479ed92e5b092c579c9b
                                ae59bca564386e642347bf9c21023ef9051578518d3b2576637c958021121d2ea6af
                                046bd0f64c5d63b020ff6a9b2103c33a9ba944fd4636704e56d7cc0301978c6736fa
                                cebbf90e0b8e692dc60f9d5b53ae

                                        Redeem Script :
                                        522103b323ad56963c69076ee2479ed92e5b092c579c9bae59bca564386e
                                        642347bf9c21023ef9051578518d3b2576637c958021121d2ea6af046bd0
                                        f64c5d63b020ff6a9b2103c33a9ba944fd4636704e56d7cc0301978c6736
                                        facebbf90e0b8e692dc60f9d5b53ae

                                        Public Key :
                                        [u'2103b323ad56963c69076ee2479ed92e5b092c579c9bae59bca564386
                                        e642347bf9c', u'21023ef9051578518d3b2576637c958021121d2ea6af
                                        046bd0f64c5d63b020ff6a9b', u'2103c33a9ba944fd4636704e56d7cc0
                                        301978c6736facebbf90e0b8e692dc60f9d5b']

                                        Signature :
                                        [u'47304402201640277f58817c09db76532f2af0f430e5bcb25b1b3d94d
                                        c9dcb823c73889e81022060cd7589983c8cf84958b366a2cbdf09b0343b6
                                        30ff0f133266e2be87a2bc50e01', u'4730440220434bfb409312a376ef
                                        ac9f81e9ab99ba1a5c4274998eee634635971193c6f6a802200bf63ba782
                                        74a666df7be21cc96bf1fd813b85bf1fc61774cc7822743507c8f701']

                        "type": P2SH-MS
                }

                "sequence": 4294967295
        }
```

```
        {
                "prev_out":

                {
                        "txid": 1563b713d4a747bf3253ee9b3cc790916c12dc7684b3c38b26066563f44bc7f1
                        "index": 1

                                "scriptSig":
                                004830450221009f7d7c44eab7bfd3acd874f1c893152089cb0f9937798b3b6fdcc4
                                7521cc5434022044249ee3516f3108af9f6dfa7be0275f1abfa3531094ff80d4af8d
                                20d064376101473044022035 5e92861b63fcf58c8e6028075d1559ffbd74d154ae90
                                a11aea56e2e3e05958022075965c83ced37bb942644772f2ddea466aeddaf4990620
                                a441cce884b781c752014c69522102aba4b3bfdea7e4b709edcc56e609845d7eeba9
                                ffb0a813e9edd5b8729c57804f21033ad2088afe60726c746dac2f2dcfd747a47cd6
                                85122ca06635af860c00f690e321025d5e665937e296a137c01afedd9a0c76e8e0f3
                                f6ac3466a260d5d8e50baadb9c53ae

                                        Redeem Script :
                                        522102aba4b3bfdea7e4b709edcc56e609845d7eeba9ffb0a813e9edd5b8
                                        729c57804f21033ad2088afe60726c746dac2f2dcfd747a47cd685122ca0
                                        6635af860c00f690e321025d5e665937e296a137c01afedd9a0c76e8e0f3
                                        f6ac3466a260d5d8e50baadb9c53ae

                                        Public Key :
                                        [u'2102aba4b3bfdea7e4b709edcc56e609845d7eeba9ffb0a813e9edd5b
                                        8729c57804f', u'21033ad2088afe60726c746dac2f2dcfd747a47cd685
                                        122ca06635af860c00f690e3', u'21025d5e665937e296a137c01afedd9
                                        a0c76e8e0f3f6ac3466a260d5d8e50baadb9c']

                                        Signature :
                                        [u'4830450221009f7d7c44eab7bfd3acd874f1c893152089cb0f9937798
                                        b3b6fdcc47521cc5434022044249ee3516f3108af9f6dfa7be0275f1abfa
                                        3531094ff80d4af8d20d064376101', u'4730440220355e92861b63fcf5
                                        8c8e6028075d1559ffbd74d154ae90a11aea56e2e3e05958022075965c83
                                        ced37bb942644772f2ddea466aeddaf4990620a441cce884b781c75201']

                        "type": P2SH-MS
                }

                "sequence": 4294967295
        }
        ]

        "out":[
        {
                "value": 5970000 satoshis
                "scriptPubKey": 76a914410d9e4a1cf587d1707b402a986812780d16b62088ac
        }
        {
                "value": 81616495 satoshis
                "scriptPubKey": a91414328fd3a99e5666514cd5167d2cc4ebed15c1a087
        }
        {
                "value": 119291601 satoshis
                "scriptPubKey": a91486744da8f363950f7c90378bdb1cc54336d1588687
        }
        ]

        "nLockTime": 0
}
```


This is a transaction with three outputs and three inputs, each of which unlocks a scriptPubKey of type P2SH-MS.

- The **version** field consists of the first 4 bytes 0x01000000 which when transformed to big endian yield **0x00000001** or **1** in decimal.
- The subsequent byte(s) correspond(s) to the input counter of type varint. In this case, the first byte is 0x03 which is less than 0xfd. As a result, varint decoding rules dictate that the input count is **0x03** i.e., **3** in decimal.
- The next 32 bytes correspond to the **txid** of the **first previous transaction** referenced by the current Bitcoin transaction:

0xf5d37b475141216a2682d7b1ca2339ddfce85bde721568498cb0db4665f0ed19

In big endian, it becomes:

0x19edf06546dbb08c49681572de5be8fcdd3923cab1d782266a21415 1477bd3f5

- The next 4 bytes correspond to the **index** of the **first previous output** which is 0x01000000 in little endian or **0x00000001** in big endian. Recalling that indexing starts at 0, this particular index corresponds to the second output of the previous Bitcoin transaction just referenced. By retrieving this output, one can see that its serialized scriptPubKey is:

  0xa914704bc33d78d213a430a982c5a4c1fd8a87959b5b87

  When deserialized, the scriptPubKey becomes:

  OP_HASH160 OP_PUSHBYTES_20
  704bc33d78d213a430a982c5a4c1fd8a87959b5b OP_EQUAL

  It is of type P2SH with a redeemScript hash given by:

  0x704bc33d78d213a430a982c5a4c1fd8a87959b5b

- The subsequent byte(s) correspond(s) to the length of the scriptSig associated with the first input. It is of type varint with first byte **0xfc**. Since it is less than 0xfd, varint decoding rules dictate that the length of scriptSig is equal to **0xfc** i.e., **252** bytes.

- The following 252 bytes correspond to the actual **scriptSig** associated with **input #1**. The scriptSig is meant to unlock the aforementioned P2SH output. In this case, it is a 2-of-3 P2SH multisignature. Recall from section 4 that this particular scriptSig must begin with OP_O, followed by two signatures and then the serialized version of the redeemScript:

  - The first byte of the scriptSig is **0x00** which corresponds to OP_0.
  - The following byte is **0x47** (i.e., OP_PUSHBYTES_71), indicating that the first signature to be pushed onto the stack is 71 bytes long. It is given by:

    0x30440220568e2dce1f11b2db59de929fd1fead6b67585ea309fb6204f245
    1a4e19b4cc1702200ab5c60127dde33532718fef6a7773625b0d4a03b44
    1eee75e2f80de6e18ee6a01

  - The following byte is also **0x47** (i.e., OP_PUSHBYTES_71). The second signature is hence 71 bytes long and given by:

    0x30440220408a2599e5daace10d148bbd1771c286b7adc60d9d5e1302ba
    bb9eb7c76ff2c002204bbc3d9143cc98a74bfa2736d42e2d209926fb07fc
    65074a3e62e21f327362f501

  - The subsequent byte is a PUSHDATA opcode that specifies how many bytes

to allocate to the length of the redeemScript. In this case, it is **0x4c** (i.e., OP_PUSHDATA1). As a result, the byte following it will be the length of the redeemScript.

- The next byte is **0x69** indicating a redeemscript length of **105** bytes.
- The following 105 bytes are the serialized redeemScript associated with this first input:

  0x522103ea74dfdbc1aa1689c00b5319bb8ccbd3074cded17efc4e5046a0be
  026a5736ce210340614f30da78f213fbe4f0eab3efd187d57f8d2d6e0880
  ff026bfe5d65fde7c62103583b06c1f8510f4a8da8c8babb867676722b32
  3787d5c57974cfe31c8cf420df53ae

  In deserialized form it becomes:

  OP_2 OP_PUSHBYTES_33
  03ea74dfdbc1aa1689c00b5319bb8ccbd3074cded17efc4e5046a0be026a5736ce
  OP_PUSHBYTES_33
  0340614f30da78f213fbe4f0eab3efd187d57f8d2d6e0880ff026bfe5d65fde7c6
  OP_PUSHBYTES_33
  03583b06c1f8510f4a8da8c8babb867676722b323787d5c57974cfe31c8cf420df
  OP_3 OP_CHECKMULTISIG

  This is a redeemScript of type P2MS with 2 of 3 multisignatures. Note that the RIPEMD160 of the serialized redeemScript is equal to the hash appearing in the aforementioned scriptPubKey:

  0x704bc33d78d213a430a982c5a4c1fd8a87959b5b

- The subsequent 8 bytes correspond to the **sequence number** associated with **input #1** and given by 0xffffffff in little endian or **4,294,967,295** in decimal. Recall from section 2 that this means that none of RLT, RBF or nLockTime are enabled.

- **Input #2** is deserialized in a similar way to Input #1 and unlocks a 2-of-3 P2SH-MS scriptPubKey. Its corresponding portion in the raw transaction is:

  0x320ccdd3bded6eb3bb317a437fc6f611254da70a2fc1fb8b9a19bec1bd67bf9a010000
  00fc0047304402201640277f58817c09db76532f2af0f430e5bcb25b1b3d94dc9dcb82
  3c73889e81022060cd7589983c8cf84958b366a2cbdf09b0343b630ff0f133266e2be8
  7a2bc50e014730440220434bfb409312a376efac9f81e9ab99ba1a5c4274998eee6346
  35971193c6f6a802200bf63ba78274a666df7be21cc96bf1fd813b85bf1fc61774cc7822
  743507c8f7014c69522103b323ad56963c69076ee2479ed92e5b092c579c9bae59bca5
  64386e642347bf9c21023ef9051578518d3b2576637c958021121d2ea6af046bd0f64c5
  d63b020ff6a9b2103c33a9ba944fd4636704e56d7cc0301978c6736facebbf90e0b8e69
  2dc60f9d5b53aeffffffff

- **Input #3**'s deserialization follows the same process and unlocks a 2-of-3 P2SH-MS scriptPubKey. Its corresponding portion in the raw transaction is:

0x1c74bf4636506268bc3b38476dc126c9190c73c9bee5332bf47a7d413b7631501000000
fdfd00004830450221009f7d7c44eab7bfd3acd874f1c893152089cb0f9937798b3b6fdcc
47521cc5434022044249ee3516f3108af9f6dfa7be0275f1abfa3531094ff80d4af8d20d064
376101473044022035 5e92861b63fcf58c8e6028075d1559ffbd74d154ae90a11aea56e2e
3e05958022075965c83ced37bb942644772f2ddea466aeddaf4990620a441cce884b781c
752014c69522102aba4b3bfdea7e4b709edcc56e609845d7eeba9ffb0a813e9edd5b8729
c57804f21033ad2088afe60726c746dac2f2dcfd747a47cd685122ca06635af860c00f690e
321025d5e665937e296a137c01afedd9a0c76e8e0f3f6ac3466a260d5d8e50baadb9c53ae
ffffffff

Note however that the length of scriptSig is given by the varint sequence 0xfdfd00 (highlighted in bold). The first byte 0xfd indicates that the following 2 bytes are the little-endian encoded length of scriptSig. These are 0xfd00, i.e., **253**.

- The next byte(s) correspond(s) to the output counter which is of type varint. The first byte is **0x03** which is less than 0xfd. As a result, varint decoding rules dictate that the output count is **0x03** i.e., **3**.
- The subsequent 8 bytes correspond to **Output #1's value** encoded in little endian and given by 0x50185b0000000000. This corresponds to Satoshi **597,0000**.
- The next byte(s) correspond(s) to the length of **Output #1's scriptPubKey**. The first byte is **0x19** which is less than 0xfd. As a result, varint decoding rules dictate that the length of this scriptPubKey is **0x19** i.e., **25** bytes.
- The next 25 bytes are **Output #1's scriptPubKey** in serialized form:

  0x76a914410d9e4a1cf587d1707b402a986812780d16b62088ac

  When deserialized, this yields the following P2PKH scriptPubKey:

  OP_DUP OP_HASH160 OP_PUSHBYTES_20
  410d9e4a1cf587d1707b402a986812780d16b620 OP_EQUALVERIFY
  OP_CHECKSIG

- The subsequent 8 bytes correspond to **Output #2's value** encoded in little endian and given by 0x6f5edd0400000000. This corresponds to Satoshi **81,616,495**.
- The next byte **0x17** is the varint length of the **scriptPubKey**.
- The following 23 bytes are **Output #2's scriptPubKey** in serialized form:

  0xa91414328fd3a99e5666514cd5167d2cc4ebed15c1a087

  When deserialized, this yields the following P2SH scriptPubKey:

  OP_HASH160 OP_PUSHBYTES_20 14328fd3a99e5666514cd5167d2cc4ebed15c1a0
  OP_EQUAL

- **Output #3** is deserialized in a similar way to Output #2. It imposes a P2SH locking condition and its corresponding portion in the raw transaction is:

  0xd13e1c070000000017a91486744da8f363950f7c90378bdb1cc54336d1588687

- The last 8 bytes correspond to the **nLockTime** value expressed in little endian as 0x00000000 i.e., **0** in decimal.

6 Coinbase and data inscription Bitcoin transaction

Coinbase transaction : This is a special type of Bitcoin transactions. Its creation is tied to the successful mining of a new block and its purpose is to unlock new bitcoins to the miners. These bitcoins do not result from a typical transfer of spending control from a payer to a payee. Instead, they follow a well-defined issuance schedule that limits all bitcoins that can ever exist to a hard cap of 21 million units expected to be reached in the year 2140. It is this finite maximal amount that confers upon Bitcoin its scarcity and makes it the epitome of sound money as defined by Austrian school economists.

Every four years, the issuance scheme halves the amount of new bitcoins per block, also known as the **block subsidy**. The next halving event will happen in May 2020 when the block subsidy will be reduced from BTC 12.5 to 6.25.

The anatomy of a coinbase transaction is not that different from that of a common Bitcoin transaction. The difference consists of the following:

- The coinbase transaction must have exactly one input.
- The txid corresponding to that input must be the all zero 32 byte long string:

  0x0000000000000000000000000000000000000000000000000000000000000000

- The 4 byte previous index field must be set to the maximum value of 0xffffffff.
- The scriptSig associated with this single input can be any arbitrary string. The rationale is that this scriptSig is not meant to unlock any previous output. However, there are two constraints on its structure:

  1. Its length must not be less than 2 bytes and not more than 100 bytes.
  2. It must start with a push of the height of the block associated with the coinbase transaction. This constraint was introduced in BIP 34 [4] and we will explain its logic later in this section.

- The sum of the values of the coinbase transaction's outputs must not surpass the sum of the appropriate block subsidy and the fees of all non-coinbase transactions included in the relevant block and owed to the miner. This upper-bound sum is also known as the **total miner's reward**.

- The output of any coinbase transaction has a maturation period of 101 block confirmations during which it cannot be spent. To justify it, recall that blockchain forks occur regularly on the Bitcoin network as explained in the chapter entitled *"To fork or not to fork: the blockchain's propensity to converge"*. If a block becomes orphaned as a result of a fork, any Bitcoin transaction that unlocks UTXOs tied to that block's coinbase transaction becomes obsolete. The maturation constraint aims to render the probability of such an occurrence negligibly small.

As an example, consider block #400,000's coinbase transaction with txid:

0xa8d0c0184dde994a09ec054286f1ce581bebf46446a512166eae7628734ea0a5

Its raw representation is given by:

0x01000000010000000000000000000000000000000000000000000000000000000000000000000
ffffffff3f03801a060004cc2acf560433c30f37085d4a39ad543b0c000a425720537570706f7274
20384d200a666973686572206a696e78696e092f425720506f6f6c2ffffffff012fd8ff9600000000
1976a914721afdf638d570285d02d3076d8be6a03ee0794d88ac00000000

- The **version** field is 0x01000000 expressed in little endian notation. This corresponds to the decimal value **1**.
- The following byte is the input counter which for a coinbase transaction must necessarily be set to 1. As expected, this byte is **0x01**.
- The next 32 bytes correspond to the only previous txid associated with this coinbase transaction. This is set to the 32 byte zero string:

  0x0000000000000000000000000000000000000000000000000000000000000000

- The next 4 bytes correspond to the previous output index which for a coinbase transaction is set to **0xffffffff**.
- The subsequent byte(s) correspond(s) to the length of the scriptSig associated with the input. It is of type varint with first byte **0x3f**. Since it is less than 0xfd, varint decoding rules dictate that the length of scriptSig is **0x3f** i.e., **63** bytes.
- The following 63 bytes contain the actual **scriptSig**:
  - Recall that the activation of BIP 34 mandated that the scriptSig of a coinbase transaction starts with the block height of its corresponding block. More specifically, it must start with a push opcode of a number of bytes that contain the height of the relevant block. In this case, the first byte is **0x03** indicating that the next **3** bytes will hold the block height value.
  - The following 3 bytes contain the block height encoded in little endian as 0x801a06 i.e., **400,000** in decimal.

  - The remaining 59 bytes (i.e., 63 - 1 - 3) contain the arbitrary string of the coinbase transaction's scriptSig. In this case it is:

    0x0004cc2acf560433c30f37085d4a39ad543b0c000a425720537570706f727420384d200a666973686572206a696e78696e092f425720506f6f6c2f

- The following 8 bytes correspond to the **sequence number** associated with the input and given by **0xffffffff** in little endian or **4,294,967,295** in decimal.
- The next byte(s) correspond(s) to the output counter of type varint. In this case, the first byte is **0x01** which is less than 0xfd. As a result, varint decoding rules show that there is only **1** output.
- The subsequent 8 bytes correspond to the **output's value** encoded in little endian and given by 0x2fd8ff9600000000 i.e., Satoshi **2,533,349,423**.
- The following byte(s) correspond(s) to the the length of the **output's locking script**. It is of type varint with first byte equal to **0x19**. Since this value is less than 0xfd, varint decoding rules imply a scriptPubKey length of **25** bytes.
- The next 25 bytes are the **output's scriptPubKey** in serialized form:

  0x76a914721afdf638d570285d02d3076d8be6a03ee0794d88ac

  When deserialized, this yields the following P2PKH scriptPubKey:

  OP_DUP OP_HASH160 OP_PUSHBYTES_20 721afdf638d570285d02d3076d8be6a03ee0794d OP_EQUALVERIFY OP_CHECKSIG

- The last 8 bytes correspond to the **nLockTime** value expressed in little endian as **0x00000000** i.e., **0** in decimal.

Prior to BIP 34's scriptSig block height constraint, it was possible for miners to create two identical coinbase transactions (i.e., with the same txid) corresponding to two distinct blocks. All that was required was to ensure that the two transactions had matching scriptSig and matching output attributes (i.e., amount and scriptPubKey). This was the case for e.g., blocks #91812 and #91842 that shared the coinbase transaction with txid:

0xd5d27987d2a3dfc724e359870c6644b40e497bdc0589a033220fe15429d88599

Its raw representations is given by:

0x01000000010000000000000000000000000000000000000000000000000000000000000000
0000ffffffff060456720e1b00ffffffff0100f2052a010000004341046896ecfc449cb8560594e
b7f413f199deb9b4e5d947a142e7dc7d2de0b811b8e204833ea2a2fd9d4c7b153a8ca76
61d0a0b7fc981df1f42f55d64b26b3da1e9cac00000000

By deserializing it, one can see that the scriptSig is 6 bytes long. Moreover, there is a single output whose value is 0x00f2052a01000000 in little endian representation or BTC 50 in decimal. Furthermore, the scriptPubKey is of type P2PK with recipient public key given by:

0x046896ecfc449cb8560594eb7f413f199deb9b4e5d947a142e7dc7d2de0b811b8e2048
33ea2a2fd9d4c7b153a8ca7661d0a0b7fc981df1f42f55d64b26b3da1e9c

The problem with having two Bitcoin transactions share the same txid is that once a UTXO in one of them gets unlocked, its identical instance in the other transaction becomes obsolete. For a coinbase transaction where each UTXO encapsulates newly mined bitcoins, this would lead to the permanent loss of bitcoins. Given that there is only one coinbase transaction per block, adding the block height to its serialized representation ensures the uniqueness of the raw content and as a result, that of its txid.

The rather arbitrary nature of the scriptSig field paved the way to inscribing various messages on the blockchain by encoding them in the coinbase transaction's scriptSig. As an example, consider the first ever coinbase transaction associated with the genesis block. Its txid is:

0x4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b

And in raw form, it is given by:

0x01000000010000000000000000000000000000000000000000000000000000000000000000
0000ffffffff**4d**04ffff001d0104455468652054696d65732030332f4a616e2f323030392043
68616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f7574
20666f722062616e6b73ffffffff0100f2052a01000000434104678afdb0fe5548271967f1a6
7130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de
5c384df7ba0b8d578a4c702b6bf11d5fac00000000

The scriptSig is 77 bytes long (hexadecimal representation of the scriptSig's length byte is highlighted in bold in the raw transaction above). These bytes are:

0x04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e63
656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420666f72
2062616e6b73

When converted to ASCII characters, one gets the following famous message inscribed by Satoshi Nakamoto:

ÿÿ EThe Times 03/Jan/2009 Chancellor on the brink of second bailout for banks

OP_RETURN transactions : An alternative method of writing data to the Bitcoin's blockchain is to use the **OP_RETURN** opcode with decimal representation 106 (0x6a in hex). Data up to 80 bytes could be stored on the blockchain by adding an

output to a Bitcoin transaction with a locking script of the form:

OP_RETURN < Data >

This scriptPubKey is commonly referred to as **NULL DATA**. In addition to the 80-byte data size constraint, OP_RETURN introduces an architectural constraint that limits the number of NULL DATA type outputs to only one per Bitcoin transaction.

The main advantage of OP_RETURN is that the NULL DATA scriptPubKey is provably unspendable. In other words, it is impossible to unlock it. This is due to the operational nature of OP_RETURN which immediately terminates the execution of the script and marks it as invalid. As a result, NULL DATA type outputs are excluded from the UTXO set, reducing as such the burden on the Bitcoin network.

To illustrate the mechanism of OP_RETURN, consider the Bitcoin transaction with txid [3]:

0xafeb330a84dc62571491132242787f4dadac48dd8d1b3affdfdbb3529db15cb9

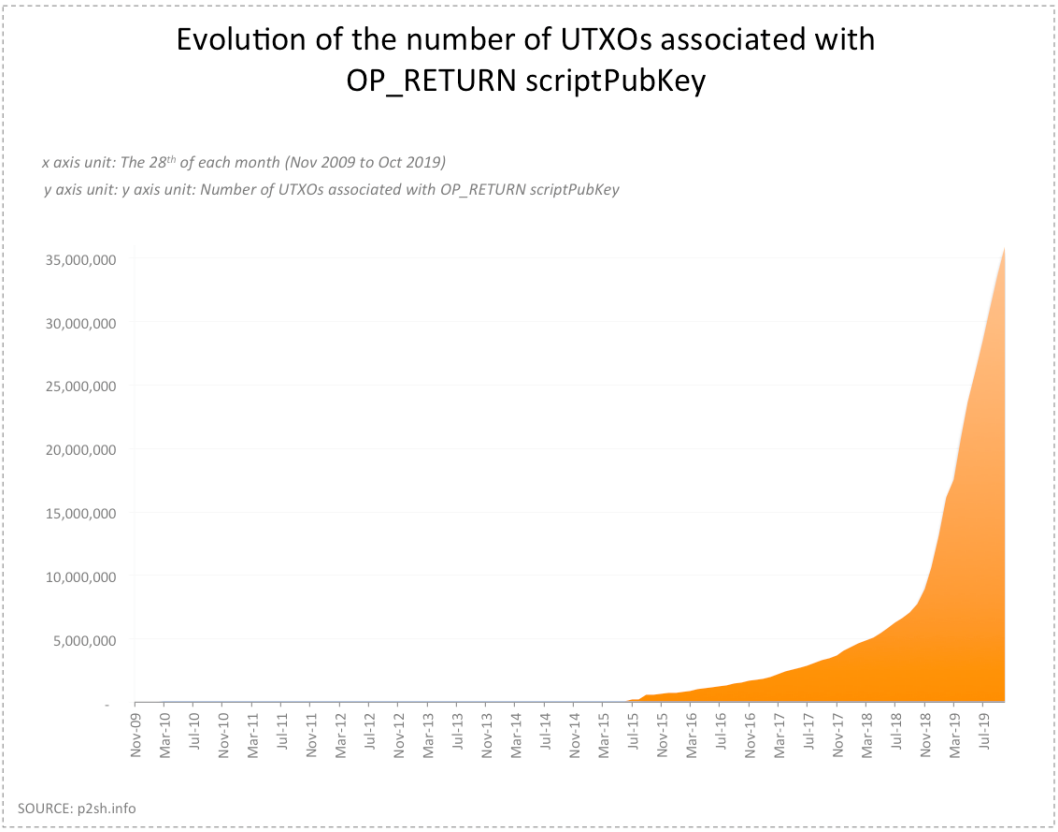
```
The serialized transaction is:
0100000001db6e9a25a86cbf0d92707fe4875038c6076918ba05bb971d598997328ca317cc010000008a47304402206fa411
bac6966f22187cd25e047382656555f9cba3854973b3c4f2404a0f3bab0220796404d317a0aa90a19d16571825d187b7a462
339ddbafd5c16bd764c5602d890141045901f6367ea950a5665335065342b952c5d5d60607b3cdc6c69a03df1a6b915aa02e
b5e07095a2548a98dcdd84d875c6a3e130bafadfd45e694a3474e71405a4ffffffff02000000000000000000106a0e43686169
6e206973206261636b211098020000000000001976a914b8268ce4d481413c4e848ff353cd16104291c45b88ac00000000


The deserialized transaction is:
{
        "version": 1

        "inputs":[
        {
                "prev_out":

                {
                        "txid": cc17a38c329789591d97bb05ba186907c6385087e47f70920dbf6ca8259a6edb
                        "index": 1

                                "scriptSig":
                                47304402206fa411bac6966f22187cd25e047382656555f9cba3854973b3c4f2404a
                                0f3bab0220796404d317a0aa90a19d16571825d187b7a462339ddbafd5c16bd764c5
                                602d890141045901f6367ea950a5665335065342b952c5d5d60607b3cdc6c69a03df
                                1a6b915aa02eb5e07095a2548a98dcdd84d875c6a3e130bafadfd45e694a3474e714
                                05a4

                                        Public Key :
                                        [u'41045901f6367ea950a5665335065342b952c5d5d60607b3cdc6c69a0
                                        3df1a6b915aa02eb5e07095a2548a98dcdd84d875c6a3e130bafadfd45e6
                                        94a3474e71405a4']

                                        Signature :
                                        [u'47304402206fa411bac6966f22187cd25e047382656555f9cba385497
                                        3b3c4f2404a0f3bab0220796404d317a0aa90a19d16571825d187b7a4623
                                        39ddbafd5c16bd764c5602d8901']

                        "type": P2PKH
                }

                "sequence": 4294967295
        }
        ]

        "out":[
        {
                "value": 0 satoshis
                "scriptPubKey": 6a0e436861696e206973206261636b21
        }
        {
                "value": 170000 satoshis
                "scriptPubKey": 76a914b8268ce4d481413c4e848ff353cd16104291c45b88ac
        }
        ]

        "nLockTime": 0
}
```

The first output has a value of Satoshi 0 and a scriptPubKey given by:

0x6a0e436861696e206973206261636b21

The first byte **0x6a** corresponds to OP_RETURN while the second **0x0e** corresponds to OP_PUSHBYTES_14. The following 14 bytes **0x436861696e206973206261636b21** correspond to the actual data committed to the blockchain. When converted from hexadecimal to UTF8, the result becomes **"Chain is back!"**.

It is worth noting that the usage of OP_RETURN has been increasing dramatically recently. The graph below shows the number of outputs associated with an OP_RETURN scriptPubKey from 2009 to 2019.



In [7], the authors analyze the usage of OP_RETURN in Bitcoin transactions up to block $453,200$ (i.e., until the 15th of February 2017) including an identification of the various protocols and classification of their relevant application domains. The usage of OP_RETURN has been a point of contention in the Bitcoin community. Some perceive it as an unwelcome diversion of the Bitcoin network away from its objective of serving as a currency transfer infrastructure. Others counter by stating that OP_RETURN still requires adequate payment to miners in order to inscribe data on the blockchain.

7 Sighash types and Bitcoin transaction signatures

In section 4, we described a subset of all possible spending conditions that can be imposed on an output of a Bitcoin transaction. Such conditions commonly include the production of an appropriate signature. Recall that signatures are applied to specific messages and generated using an appropriate private key (refer to the post entitled *"Bitcoin Elliptic Curve Digital Signature Algorithm (ECDSA)"*). Consequently, it becomes crucial to specify what message must be considered to unlock a given UTXO.

It turns out that the formation of such messages is not monolithic. Instead, it depends on what elements of the Bitcoin transaction that counts the UTXO as one of its inputs are selected for inclusion. For example, one may decide to include information about every single input. Alternatively, one may choose to limit such input information to that associated with the specific UTXO. Similarly, one may decide to include all output information or instead, limit the inclusion to a subset of such outputs.

One must not equate this flexibility with complete freedom in deciding what to include or exclude. As a matter of fact, message formation options are limited to a set of six possibilities. In order to verify the validity of a signature, nodes need to know which model was used. To that effect, a particular byte known as **sighash** is appended to the end of every UTXO signature. The set of possible sighash bytes comprises:

- **SIGHASH_ALL** (0x01 in hex)
- **SIGHASH_NONE** (0x02 in hex)
- **SIGHASH_SINGLE** (0x03 in hex)
- **SIGHASH_ALL_ANYONECANPAY** (0x81 in hex)
- **SIGHASH_NONE_ANYONECANPAY** (0x82 in hex)
- **SIGHASH_SINGLE_ANYONECANPAY** (0x83 in hex)

The objective of this section is to discuss the details of these options. In particular, we illustrate how to create UTXO-specific messages based on a sighash type and use these procedures to generate signatures for a testnet Bitcoin transaction. In addition, we write a python code (applicable only to a pre-segwit Bitcoin transaction with UTXOs of type P2PKH or P2SH-MS) to build UTXO-specific messages and verify their signatures.

Throughout this section, we consider a generic Bitcoin transaction with these attributes:

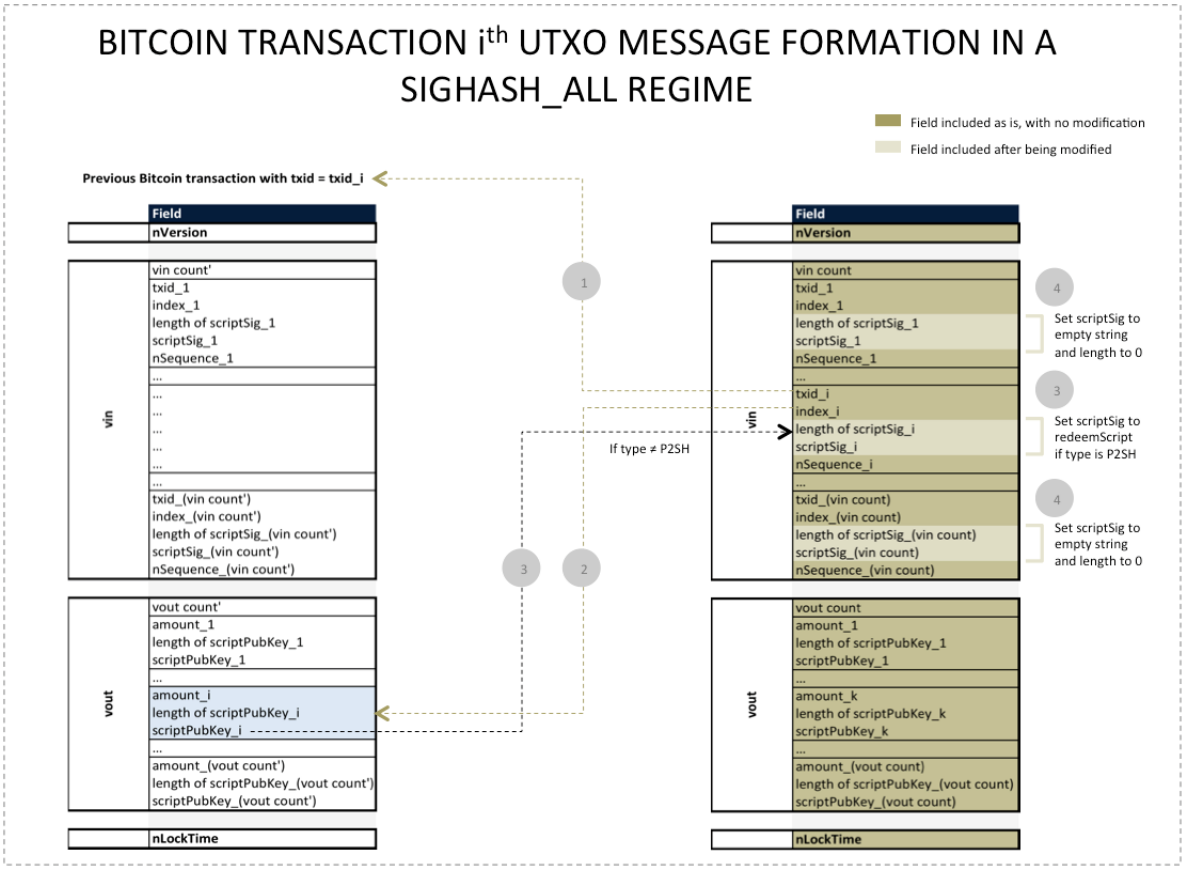
- A version field specified by **nVersion**.
- A total of **vin count** inputs, where the ith input corresponds to the UTXO located at **index_i** of a previous transaction with **txid_i**.
- A total of **vout count** outputs, where the kth output ($1 \leq k \leq$ *vout count*) encapsulates an amount of Satoshis **amount_k** subjected to **scriptPubKey_k**.
- A locktime value specified by **nLockTime**.

Without loss of generality, we limit ourselves to producing relevant messages associated with the ith UTXO of such a Bitcoin transaction.

The case of a SIGHASH_ALL byte: This is the most common of all types. All the elements of the Bitcoin transaction must be part of the message. As a result, a sender requires that her UTXO be spent alongside a pre-defined set of inputs and destined to a pre-defined set of recipients with pre-defined amounts.

However, some of the components of the Bitcoin transaction must be modified prior to inclusion. In particular, the **scriptSig** fields of all the inputs will have to be altered. This is because **scriptSig_s** ($1 \leq s \leq vin\ count$) would hold the signature associated with the sth UTXO and it would be logically challenging to expect these fields to know it in advance. More specifically, the modification is conducted as follows:

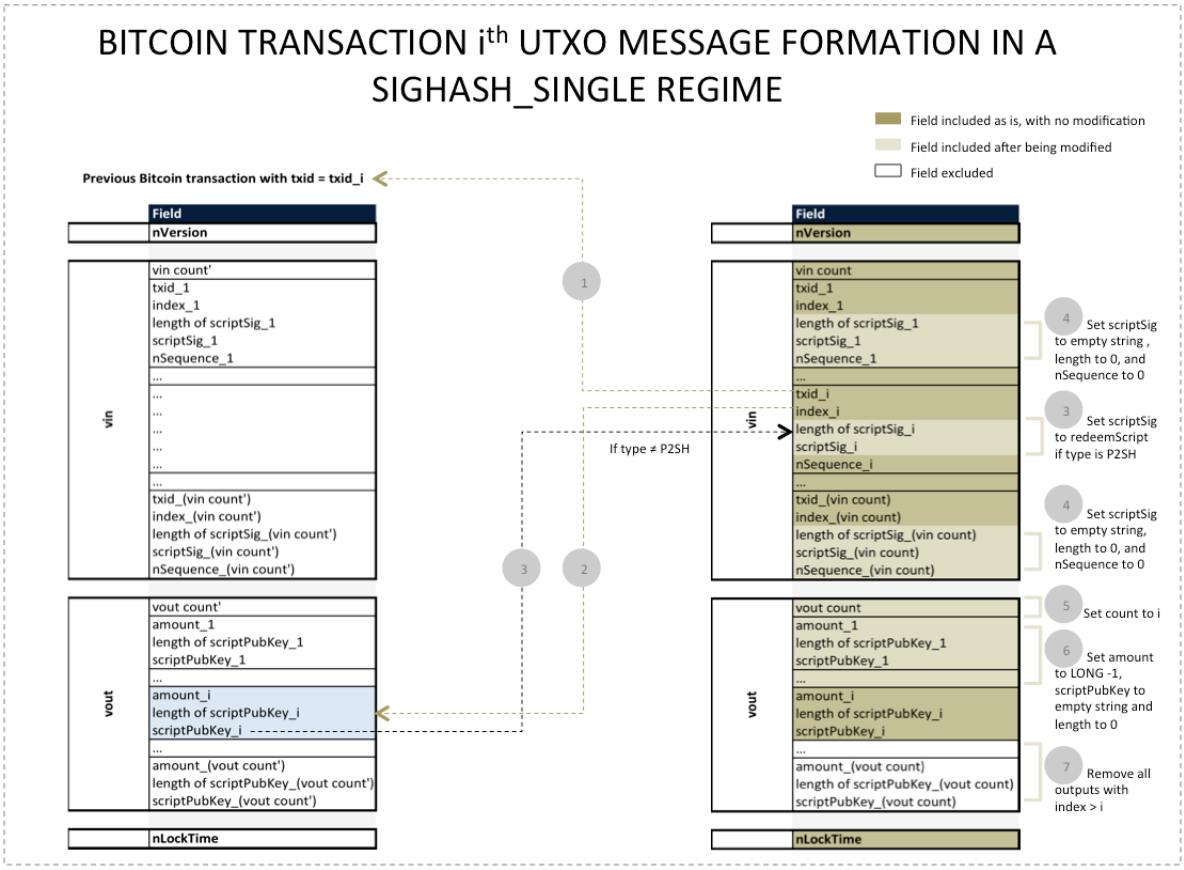
1. Retrieve the previous transaction containing the ith UTXO as an output.
2. Retrieve the index of this UTXO in the previous transaction.
3. If UTXO is of type P2SH, replace scriptSig_i with P2SH's **redeemScript**. Else, replace it with **scriptPubKey_i**. Update length of scriptSig_i. If scriptPubKey_i contains OP_CODESEPARATOR (legacy from an older Bitcoin client version), further modifications would be needed. The scriptSig of a P2PKH and redeemScript of a P2SH-MS UTXO are OP_CODESEPARATOR free.
4. Set **scriptSig_s** to empty string and its **length** to 0 ($1 \leq s \leq vin\ count,\ s \neq i$).



The modified content is subsequently serialized as outlined in section 2. The sighash byte 0x01 is extended to the 4 byte sequence 0x01000000 and appended to the serialized content. The desired message is formed by taking a double SHA256 of the outcome.

The case of a SIGHASH_SINGLE byte: All inputs are included but any output with index $> i$ is excluded (ideally, there should be at least as many outputs as inputs). Any output with index $< i$ will have its amount set to -1 (i.e., 0xffffffffffffffff in hex) and its scriptPubKey to the empty string. De facto, a sender requires that his UTXO be spent alongside a pre-defined set of inputs as long as a given amount is sent to a specific recipient without caring about others. The process is as follows:

1. Retrieve the previous transaction containing the ith UTXO as an output.
2. Retrieve the index of this UTXO in the previous transaction.
3. If UTXO is of type P2SH, replace scriptSig_i with P2SH's **redeemScript**. Else, replace it with **scriptPubKey_i**. Update length of scriptSig_i. If scriptPubKey_i contains OP_CODESEPARATOR (legacy from an older Bitcoin client version), further modifications would be needed. The scriptSig of a P2PKH and redeemScript of a P2SH-MS UTXO are OP_CODESEPARATOR free.
4. Set **scriptSig_s** to an empty string, **length of scriptSig_s** to 0 and **nSequence_s** to 0 ($1 \leq s \leq vin\ count,\ s \neq i$)
5. Update **vout count** to the value i.
6. Change **amount_s** to -1, replace **scriptPubKey_s** with an empty string and set **length of scriptPubKey_s** to 0 ($1 \leq s < i$).
7. Remove all transaction outputs with index greater than i.



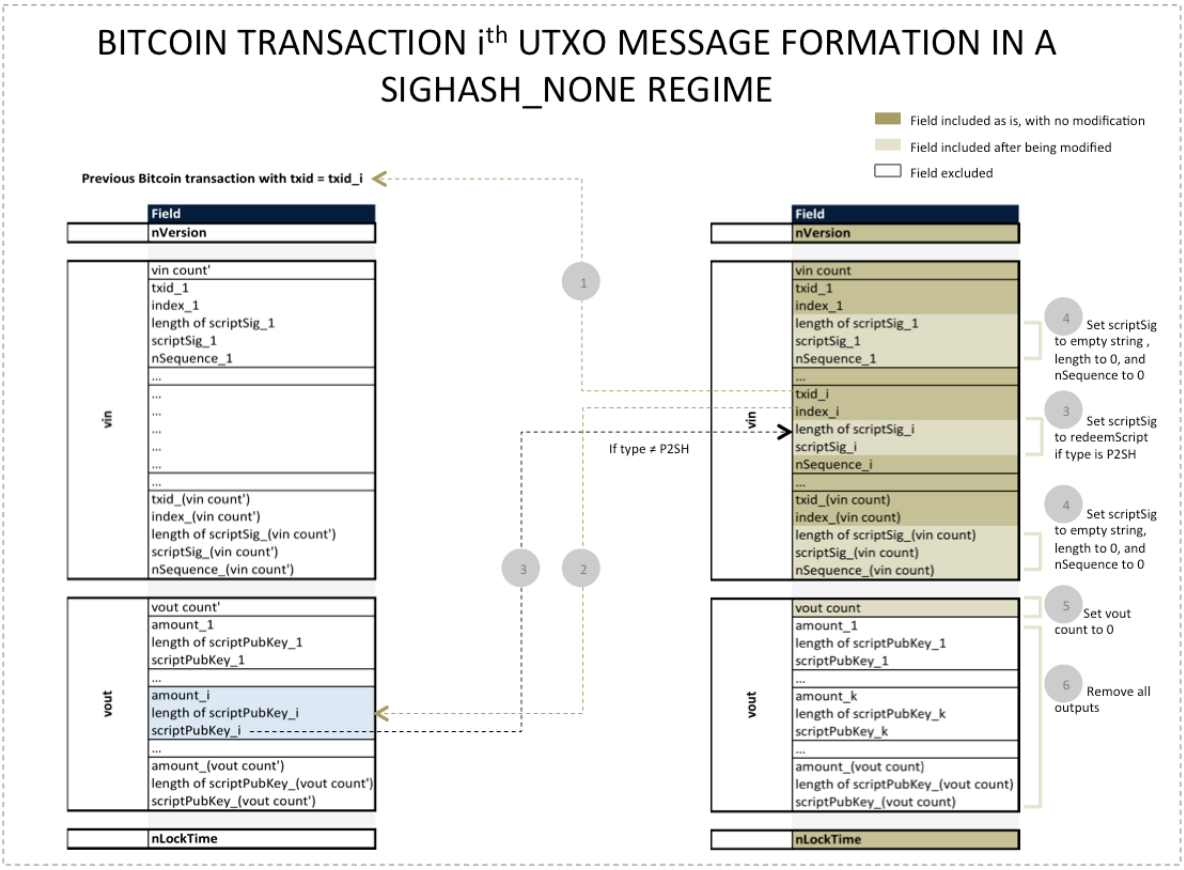
The modified content is subsequently serialized as outlined in section 2. The sighash byte 0x03 is extended to the 4 byte sequence 0x03000000 and appended to the serialized content. The desired message is formed by taking a double SHA256 of the outcome.

The case of a SIGHASH_NONE byte: All inputs are included but none of the outputs are. In other words, a sender requires that her UTXO be spent alongside a pre-defined set of inputs without caring about who receives it.

This type of signature is deemed insecure if used in a Bitcoin transaction with a single input because anyone could snatch it. However, it may be useful in the context of a Bitcoin transaction with many inputs. For example, consider a scenario where one of the inputs corresponds to an active investor. All the other inputs are associated with passive investors who agree to commit their funds as long as all of them participate. Moreover, they do not need to know their funds' final destination in advance because the investment will be determined at a later stage by their active colleague. Consequently, the passive subset sign their UTXOs with SIGHASH_NONE and trust that their active colleague will sign his with SIGHASH_ALL upon sealing his portfolio.

The process is straightforward with all transaction outputs removed. More specifically:

1. Retrieve the previous transaction containing the ith UTXO as an output.
2. Retrieve the index of this UTXO in the previous transaction.
3. If UTXO is of type P2SH, replace scriptSig_i with P2SH's **redeemScript**. Else, replace it with **scriptPubKey_i**. Update length of scriptSig_i. If scriptPubKey_i contains OP_CODESEPARATOR (legacy from an older Bitcoin client version), further modifications would be needed. The scriptSig of a P2PKH and redeemScript of a P2SH-MS UTXO are OP_CODESEPARATOR free.
4. Set **scriptSig_s** to an empty string, **length of scriptSig_s** to 0 and **nSequence_s** to 0 ($1 \leq s \leq vin\ count,\ s \neq i$)
5. Set **vout count** to 0.
6. Remove all transaction outputs.



The modified content is subsequently serialized as outlined in section 2. The sighash byte 0x02 is extended to the 4 byte sequence 0x02000000 and appended to the serialized content. The desired message is formed by taking a double SHA256 of the outcome.

All the previous sighash bytes mandated the inclusion of all inputs in order to form the message. It turns out that the sender could alternatively limit the subset of inputs to a singleton with the one corresponding to the relevant UTXO. This flexibility introduces three more sighash bytes as described below.

The case of a SIGHASH_ALL_ANYONECANPAY byte: This is similar to SIGHASH_ALL in that all outputs are included in the message. It however differs from it by limiting its inputs to input i. In this case, a sender requires that her UTXO be part of a Bitcoin transaction with a pre-defined set of recipients and pre-defined amounts destined to each of one of them. However, she does not care about who else may be funding the Bitcoin transaction. This could be useful for crowdfunding campaigns where recipients are known in advance but funders do not necessarily care about who else may be contributing.

The message formation process mimics that of SIGHASH_ALL with the additional steps of setting **vin count** to 1 and removing all inputs other than input i. More specifically, the modification is conducted as follows:

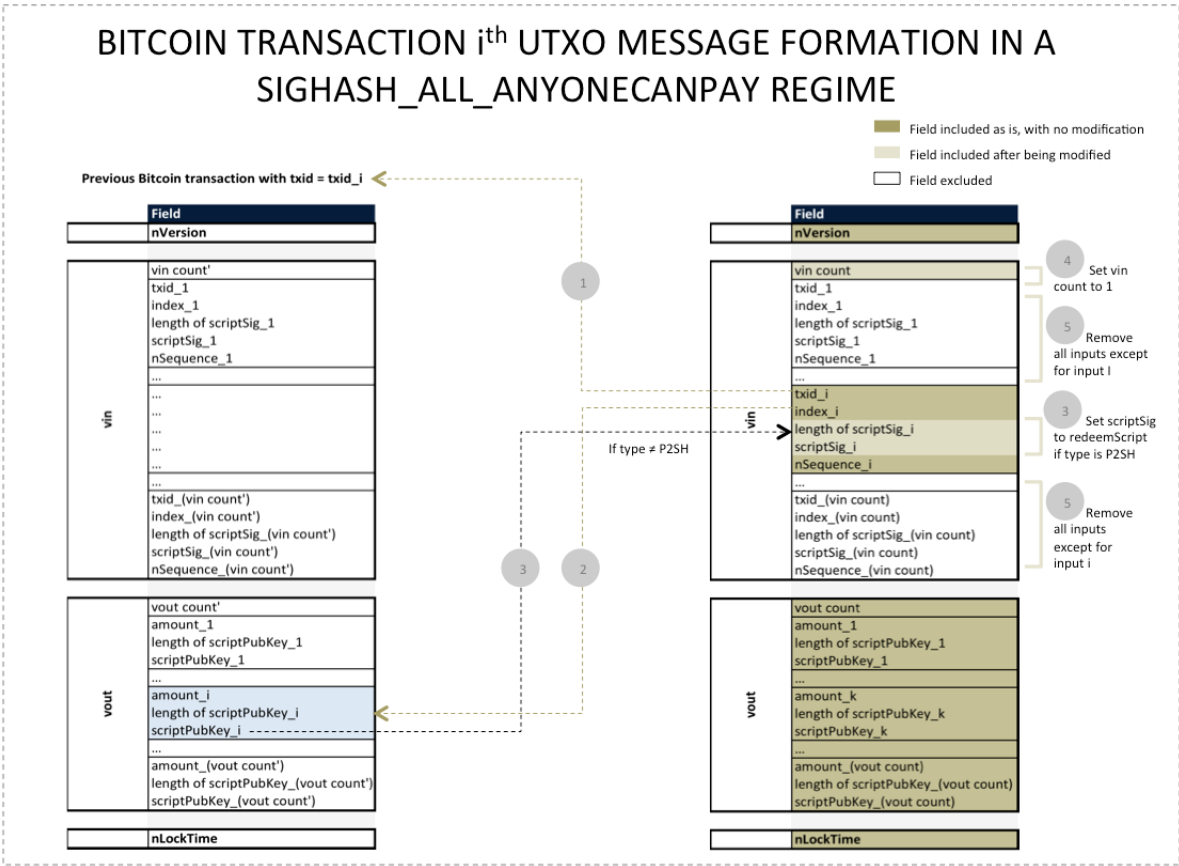
1. Retrieve the previous transaction containing the ith UTXO as an output.

2. Retrieve the index of this UTXO in the previous transaction.

3. If UTXO is of type P2SH, replace scriptSig_i with P2SH's **redeemScript**. Else, replace it with **scriptPubKey_i**. Update length of scriptSig_i. If scriptPubKey_i contains OP_CODESEPARATOR (legacy from an older Bitcoin client version), further modifications would be needed. The scriptSig of a P2PKH and redeemScript of a P2SH-MS UTXO are OP_CODESEPARATOR free.

4. Set **vin count** to 1.

5. Remove all inputs except for input i.



The modified content is subsequently serialized as outlined in section 2. The sighash byte 0x81 is extended to the 4 byte sequence 0x81000000 and appended to the serialized content. The desired message is formed by taking a double SHA256 of the outcome.

The case of a SIGHASH_SINGLE_ANYONECANPAY byte: This is similar to SIGHASH_SINGLE but limits its input set to the singleton with input i. In this regime, a sender remains indifferent vis-a-vis other senders or recipients as long as a pre-defined amount gets sent to one specific recipient. The modification is conducted as follows:

1. Retrieve the previous transaction containing the ith UTXO as an output.

2. Retrieve the index of this UTXO in the previous transaction.

3. If UTXO is of type P2SH, replace scriptSig_i with P2SH's **redeemScript**. Else, replace it with **scriptPubKey_i**. Update length of scriptSig_i. If scriptPubKey_i contains OP_CODESEPARATOR (legacy from an older Bitcoin client version), further modifications would be needed. The scriptSig of a P2PKH and redeemScript of a P2SH-MS UTXO are OP_CODESEPARATOR free.

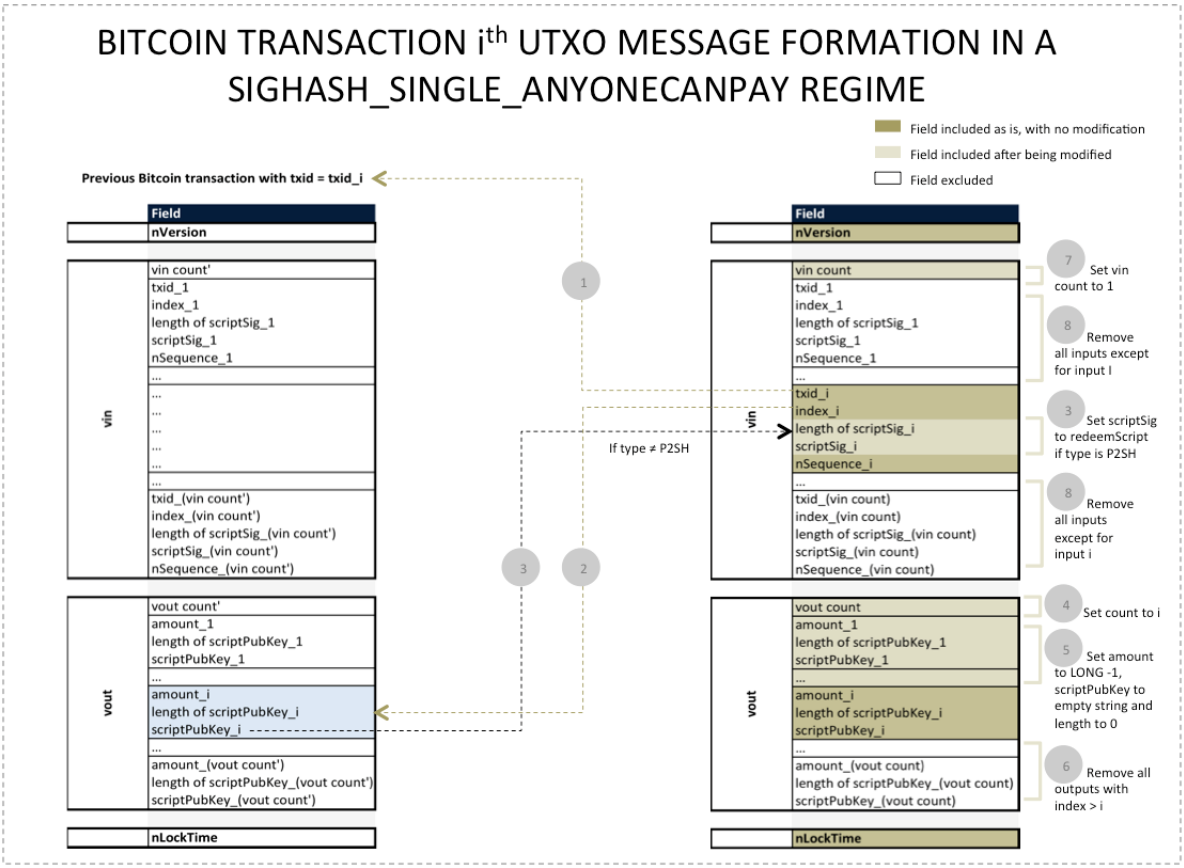
4. Update **vout count** to the value i.

5. Change **amount_s** to -1, replace **scriptPubKey_s** with an empty string and set **length of scriptPubKey_s** to 0 ($1 \leq s < i$).

6. Remove all transaction outputs with index greater than i.

7. Set **vin count** to 1.

8. Remove all inputs except for input i.

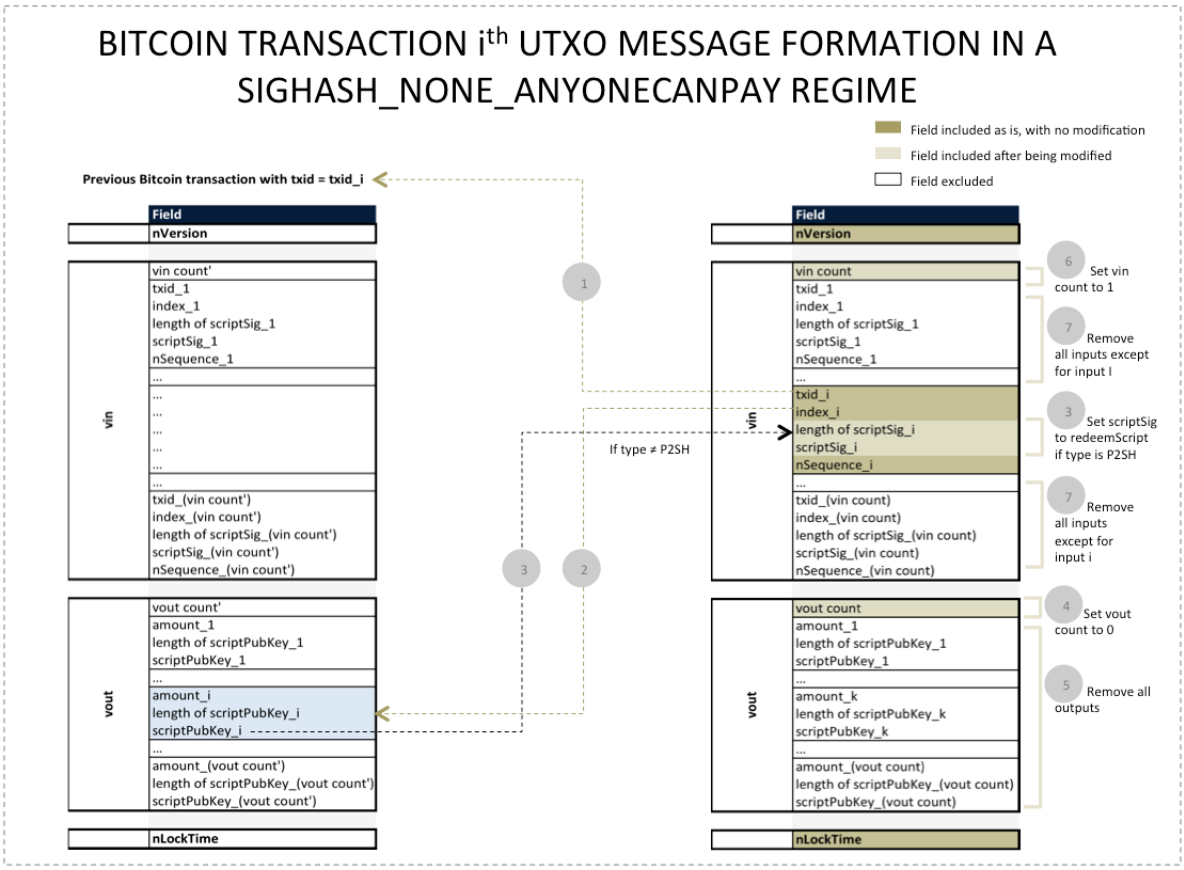


The modified content is subsequently serialized as outlined in section 2. The sighash byte 0x83 is extended to the 4 byte sequence 0x83000000 and appended to the serialized content. The desired message is formed by taking a double SHA256 of the outcome.

The case of a SIGHASH_NONE_ANYONECANPAY byte: This is similar to SIGHASH_NONE but limits its input set to the singleton with input i. In this case, a sender remains indifferent vis-a-vis other senders or recipients.

It is not recommended to use this type of signature on a Bitcoin transaction with a single input because anyone could claim it. However, it may be useful in the context of a multi-input Bitcoin transaction. For example, consider a scenario similar to the one described earlier for SIGHASH_NONE. The difference now is that passive investors do not care about who else may be funding the investment. Recall that the active investor is later expected to sign his UTXO with a SIGHASH_ALL byte. The modification is conducted as follows:

1. Retrieve the previous transaction containing the ith UTXO as an output.
2. Retrieve the index of this UTXO in the previous transaction.
3. If UTXO is of type P2SH, replace scriptSig_i with P2SH's **redeemScript**. Else, replace it with **scriptPubKey_i**. Update length of scriptSig_i. If scriptPubKey_i contains OP_CODESEPARATOR (legacy from an older Bitcoin client version), further modifications would be needed. The scriptSig of a P2PKH and redeemScript of a P2SH-MS UTXO are OP_CODESEPARATOR free.
4. Set **vout count** to 0.
5. Remove all transaction outputs.
6. Set **vin count** to 1.
7. Remove all inputs except for input i.



The modified content is subsequently serialized as outlined in section 2. The sighash byte 0x82 is extended to the 4 byte sequence 0x82000000 and appended to the serialized content. The desired message is formed by taking a double SHA256 of the outcome.

Illustrative example of signing a new Bitcoin transaction: We now use the aforementioned procedures to sign the inputs of a testnet Bitcoin transaction using different sighash types. The Bitcoin transaction will have four inputs and two outputs. All four inputs correspond to UTXOs of type P2PKH. The first input's UTXO will be signed with SIGHASH_ALL, the second with SIGHASH_SINGLE, the third with SIGHASH_NONE and the fourth with SIGHASH_ALL_ANYONECANPAY.

- Details of the first UTXO:
  - This UTXO corresponds to the first output of the testnet Bitcoin transaction whose txid is:

    0x663325d63f76fec38b153718dd1ede1bd150c76c51214f0fa0b386d59fb5bc7b

  - Its owner's private and public keys are respectively:

    0xae53c7e504f8e281e9b45ae3f71f837296196d02182c8d01fa924bd857c97de9

    0x0427a5ebe8b8af49e2afd490eb5ce05469b535bb433ea0bd176eee4977252e8a
    05961a9f6a58c16dcd2be136c9fe73bc35689673bd8e5e3cb21a68c1e7367fc7e3

  - Its nSequence number is set to 0xffffffff.

- Details of the second UTXO:
  - This UTXO corresponds to the first output of the testnet Bitcoin transaction whose txid is:

    0x0f704d306d84c4626f76faead68b40aff5bd005f57128e13ce822bf7f549b718

  - Its owner has the same private and public key as the first UTXO
  - Its nSequence number is set to 0xffffffff.

- Details of the third UTXO:
  - This UTXO corresponds to the first output of the testnet Bitcoin transaction whose txid is:

    0x88b0a5cba88aa22d271d1de3157d4bc2683bfc6b9feecc0bee800cc1926cae17

  - Its owner's private and public keys are respectively:

0x82ab03119c793609e9e2b5a477647d02c97a14f2004e53d1c89fc6ec1b34280a

0x043391bcbfdcc5f9a3c9bcb9b1b4f3296f55ca46c6207d6c3da8eba4e2ec8b19
96c445397a1ac33f610669c307e5853980d7883b036f9865d5cbbdb7564ed25894

  - Its nSequence number is set to 0xffffffff.

- Details of the fourth UTXO:
  - This UTXO corresponds to the first output of the testnet Bitcoin transaction whose txid is:

    0x4fd95f9ef04a4a6fb21fa5618513d95f905e2c676973fbb41d074ac53f9a7e89

  - Its owner's private and public keys are respectively:

    0xed79ab140cbc162770cec536cc927421e8e5ca143018a8197f664c76b91d7922

    0x04ae8d673db4f5d11cf9f39eaa801dce86babff2f9cadf504260f01e7c017855
    e3fdf1a5fad9a3413a4bf81a9e4b8ef47fda1a08cc6e58605779513e295dd5797b

  - Its nSequence number is set to 0xffffffff.

- Details of the first output:
  - It encapsulates an amount of Satoshi 6,062,628.
  - It has a P2PKH scriptPubKey given by:

    0x76a9149f9a7abd600c0caa03983a77c8c3df8e062cb2fa88ac

- Details of the second output:
  - It encapsulates an amount of Satoshi 19,000.
  - It has a P2PKH scriptPubKey given by:

    0x76a9142cd2c0ac48f6383c72ca45a66fffa37a26b38d7288ac.

- The Bitcoin transaction has version 1 and locktime set to 0.

The following code creates a SIGHASH_ALL message for the first UTXO and then signs it. Note that the method **ecdsa_Sign** used below was originally introduced in the post entitled *"Bitcoin Elliptic Curve Digital Signature Algorithm (ECDSA)"*:

```
tx = BTC_TX("testnet")                          # Create a testnet tx object using
                                                # the BTC_TX class defined earlier
tx.inputs = 4
tx.prev_tx_id = ["663325d63f76fec38b153718dd1ede1bd150c76c51214f0fa0b386d59fb5bc7b",
                 "0f704d306d84c4626f76faead68b40aff5bd005f57128e13ce822bf7f549b718",
                 "88b0a5cba88aa22d271d1de3157d4bc2683bfc6b9feecc0bee800cc1926cae17",
                 "4fd95f9ef04a4a6fb21fa5618513d95f905e2c676973fbb41d074ac53f9a7e89"]

tx.prev_out_index = [0, 0, 0, 0]

tx.scriptSig = ["76a914ac8b668dd659bec9b86ce26fad2fa823484cb18588ac", "", "", ""]

tx.nSequence = [4294967295, 4294967295, 4294967295, 4294967295]

tx.outputs = 2
tx.value = [6062628, 19000]
tx.scriptPubKey = ["76a9149f9a7abd600c0caa03983a77c8c3df8e062cb2fa88ac",
                   "76a9142cd2c0ac48f6383c72ca45a66fffa37a26b38d7288ac"]

tx.version = 1
tx.nLockTime = 0

tx_raw = tx.serialize() + '01000000'            # Append SIGHASH_ALL 4 bytes


#Transform result to base256
tx_raw_256 = tx_raw.decode('hex')

#Run SHA256
'''Note that the final message must be a double SHA256 of the serialized output.
In what follows, tx_raw_sha256 runs a single SH256 instance while the second
iteration is done by the ecdsa_Sign method itself.
'''
tx_raw_sha256 = hashlib.sha256(tx_raw_256).digest()

priv_key_1 = int("ae53c7e504f8e281e9b45ae3f71f837296196d02182c8d01fa924bd857c97de9", 16)

r, s = ecdsa_Sign(priv_key_1, tx_raw_sha256)
```

This resulted in a signature (r,s) with:

- r ≡ 0x1f8dd6a75fdd21b36c46b9f6281ddd4339862194c8c4d6931e80761987435a8e
- s ≡ 0x49f1c316242ff34feb3e767d00e02cc9ac5545b1849fd6cebb34d095aefd4ce3

The signature is subsequently encoded in DER format as introduced in the post entitled *"Bitcoin Elliptic Curve Digital Signature Algorithm (ECDSA)"* and appended with the SIGHASH_ALL byte. The outcome is:

0x47304402201f8dd6a75fdd21b36c46b9f6281ddd4339862194c8c4d6931e8076
1987435a8e022049f1c316242ff34feb3e767d00e02cc9ac5545b1849fd6cebb
34d095aefd4ce301

Since this UTXO is of type P2PKH, its correpsonding scriptSig is obtained by concatenating the signature and the UTXO owner's public key as follows:

0x47304402201f8dd6a75fdd21b36c46b9f6281ddd4339862194c8c4d6931e8076
1987435a8e022049f1c316242ff34feb3e767d00e02cc9ac5545b1849fd6cebb
34d095aefd4ce301410427a5ebe8b8af49e2afd490eb5ce05469b535bb433ea0
bd176eee4977252e8a05961a9f6a58c16dcd2be136c9fe73bc35689673bd8e5e
3cb21a68c1e7367fc7e3

The following code creates a SIGHASH_SINGLE message for the second UTXO and then signs it:

```
tx = BTC_TX("testnet")                          # Create a testnet tx object using
                                                # the BTC_TX class defined earlier
tx.inputs = 4
tx.prev_tx_id = ["663325d63f76fec38b153718dd1ede1bd150c76c51214f0fa0b386d59fb5bc7b",
                 "0f704d306d84c4626f76faead68b40aff5bd005f57128e13ce822bf7f549b718",
                 "88b0a5cba88aa22d271d1de3157d4bc2683bfc6b9feecc0bee800cc1926cae17",
                 "4fd95f9ef04a4a6fb21fa5618513d95f905e2c676973fbb41d074ac53f9a7e89"]

tx.prev_out_index = [0, 0, 0, 0]

tx.scriptSig = ["", "76a914ac8b668dd659bec9b86ce26fad2fa823484cb18588ac", "", ""]

tx.nSequence = [0, 4294967295, 0, 0]

tx.outputs = 2
tx.value = [18446744073709551615, 19000]
tx.scriptPubKey = ["", "76a9142cd2c0ac48f6383c72ca45a66fffa37a26b38d7288ac"]


tx.version = 1
tx.nLockTime = 0

tx.display()

tx_raw = tx.serialize() + '03000000'            # Append SIGHASH_SINGLE 4 bytes


#Transform to base256 and take sha256
tx_raw_256 = tx_raw.decode('hex')

# Run SHA256
''' Note that the final message must be a double SHA256 of the serialized output.
In what follows, tx_raw_sha256 runs a single SHA256 instance while the second
iteration is done by the ecdsa_Sign method itself.
'''
tx_raw_sha256 = hashlib.sha256(tx_raw_256).digest()

priv_key_2 = int("ae53c7e504f8e281e9b45ae3f71f837296196d02182c8d01fa924bd857c97de9", 16)

r, s = ecdsa_Sign(priv_key_2, tx_raw_sha256)
```

This resulted in a signature (r,s) with:

- r ≡ 0x3c62c8893223e9a8abff7a983e2f569806d5f1f1cb954f25fbeedd204f1c67f3
- s ≡ 0x1d0eaecec9e280bca7d5cc395277e4b8be07d1de34980d4860de202f3db59f81

The signature is subsequently encoded in DER format and appended with the SIGHASH_SINGLE byte. The outcome is :

0x47304402203c62c8893223e9a8abff7a983e2f569806d5f1f1cb954f25fbeedd
204f1c67f302201d0eaecec9e280bca7d5cc395277e4b8be07d1de34980d48
60de202f3db59f8103

Since this UTXO is of type P2PKH, its correpsonding scriptSig is obtained by concatenating the signature and the UTXO owner's public key as follows:

0x47304402203c62c8893223e9a8abff7a983e2f569806d5f1f1cb954f25fbeedd
204f1c67f302201d0eaecec9e280bca7d5cc395277e4b8be07d1de34980d48
60de202f3db59f8103410427a5ebe8b8af49e2afd490eb5ce05469b535bb433e
a0bd176eee4977252e8a05961a9f6a58c16dcd2be136c9fe73bc35689673bd8e
5e3cb21a68c1e7367fc7e3

The following code creates a SIGHASH_NONE message for the third UTXO and then signs it:

```
tx = BTC_TX("testnet")                          # Create a testnet tx object using
                                                # the BTC_TX class defined earlier
tx.inputs = 4
tx.prev_tx_id = ["663325d63f76fec38b153718dd1ede1bd150c76c51214f0fa0b386d59fb5bc7b",
                 "0f704d306d84c4626f76faead68b40aff5bd005f57128e13ce822bf7f549b718",
                 "88b0a5cba88aa22d271d1de3157d4bc2683bfc6b9feecc0bee800cc1926cae17",
                 "4fd95f9ef04a4a6fb21fa5618513d95f905e2c676973fbb41d074ac53f9a7e89"]

tx.prev_out_index = [0, 0, 0, 0]

tx.scriptSig = ["", "", "76a9146553774650fcc12b918ce2e17606103c5329592788ac", ""]

tx.nSequence = [0, 0, 4294967295, 0]

tx.outputs = 0
tx.value = []
tx.scriptPubKey = []

tx.version = 1
tx.nLockTime = 0

tx_raw = tx.serialize() + '02000000'          # Append SIGHASH_NONE 4 bytes


#Transform result to base256
tx_raw_256 = tx_raw.decode('hex')

#Run SHA256
'''Note that the final message must be a double SHA256 of the serialized output.
In what follows, tx_raw_sha256 runs a single SH256 instance while the second
iteration is done by the ecdsa_Sign method itself.
'''
tx_raw_sha256 = hashlib.sha256(tx_raw_256).digest()

priv_key_3 = int("82ab03119c793609e9e2b5a477647d02c97a14f2004e53d1c89fc6ec1b34280a", 16)

r, s = ecdsa_Sign(priv_key_3, tx_raw_sha256)
```

This resulted in a signature (r,s) with:

- r ≡ 0x8623da9c40faf27cc4d3c6514f74c9ef65fba189079d2d189ba5f2400fba9b35
- s ≡ 0x1801ba05f916906ba1483a4109616a75efcc28d6976708b8d2fdd702440493ed

The signature is subsequently encoded in DER format and appended with the SIGHASH_NONE byte. The outcome is:

0x4830450221008623da9c40faf27cc4d3c6514f74c9ef65fba189079d2d189ba5
f2400fba9b3502201801ba05f916906ba1483a4109616a75efcc28d6976708b8
d2fdd702440493ed02

Since this UTXO is of type P2PKH, its correpsonding scriptSig is obtained by concatenating the signature and the UTXO owner's public key as follows:

0x4830450221008623da9c40faf27cc4d3c6514f74c9ef65fba189079d2d189ba5
f2400fba9b3502201801ba05f916906ba1483a4109616a75efcc28d6976708b8
d2fdd702440493ed0241043391bcbfdcc5f9a3c9bcb9b1b4f3296f55ca46c620
7d6c3da8eba4e2ec8b1996c445397a1ac33f610669c307e5853980d7883b036f
9865d5cbbdb7564ed25894

The following code creates a SIGHASH_ALL_ANYONECANPAY message for the fourth UTXO and then signs it:

```
tx = BTC_TX("testnet")                      # Create a testnet tx object using
                                            # the BTC_TX class defined earlier
tx.inputs = 1
tx.prev_tx_id = ["4fd95f9ef04a4a6fb21fa5618513d95f905e2c676973fbb41d074ac53f9a7e89"]

tx.prev_out_index = [0]

tx.scriptSig = ["76a914ccb0fced337a301b301e3e5dc73f4a6ae268486488ac"]

tx.nSequence = [4294967295]

tx.outputs = 2
tx.value = [6062628, 19000]
tx.scriptPubKey = ["76a9149f9a7abd600c0caa03983a77c8c3df8e062cb2fa88ac",
                   "76a9142cd2c0ac48f6383c72ca45a66fffa37a26b38d7288ac"]

tx.version = 1
tx.nLockTime = 0

tx_raw = tx.serialize() + '81000000'        # Append SIGHASH_ALL_ANYONECANPAY 4 bytes


#Transform result to base256
tx_raw_256 = tx_raw.decode('hex')

#Run SHA256
'''Note that the final message must be a double SHA256 of the serialized output.
In what follows, tx_raw_sha256 runs a single SH256 instance while the second
iteration is done by the ecdsa_Sign method itself.
'''
tx_raw_sha256 = hashlib.sha256(tx_raw_256).digest()

priv_key_4 = int("ed79ab140cbc162770cec536cc927421e8e5ca143018a8197f664c76b91d7922", 16)

r, s = ecdsa_Sign(priv_key_4, tx_raw_sha256)
```

This resulted in a signature (r,s) with:

- r ≡ 0xea7d40b54efe8bcadb03a9d80f9d2083a8e2514b28d408742775b8b48b888b6d
- s ≡ 0x520a8acc1492c6e227a2eedda054c2ab0da0c84be09b70caa066397986696a7c

The signature is subsequently encoded in DER format and appended with the SIGHASH_ALL_ANYONECANPAY byte. The outcome is :

0x483045022100ea7d40b54efe8bcadb03a9d80f9d2083a8e2514b28d408742775
b8b48b888b6d0220520a8acc1492c6e227a2eedda054c2ab0da0c84be09b70ca
a066397986696a7c81

Since this UTXO is of type P2PKH, its correpsonding scriptSig is obtained by concatenating the signature and the UTXO owner's public key as follows:

0x483045022100ea7d40b54efe8bcadb03a9d80f9d2083a8e2514b28d408742775
b8b48b888b6d0220520a8acc1492c6e227a2eedda054c2ab0da0c84be09b70ca
a066397986696a7c814104ae8d673db4f5d11cf9f39eaa801dce86babff2f9ca
df504260f01e7c017855e3fdf1a5fad9a3413a4bf81a9e4b8ef47fda1a08cc6e
58605779513e295dd5797b

With the various scriptSig fields computed, one gets the testnet Bitcoin transaction with txid:

0xcf06db9c0a2cddafcfbf2b39d27d1e16d45a7b0ec4fedf6df55d205f7f0b5ffc

```
The serialized transaction is:
01000000047bbcb59fd586b3a00f4f21516cc750d11bde1edd1837158bc3fe763fd6253366000000008a47304402201f8dd6
a75fdd21b36c46b9f6281ddd4339862194c8c4d6931e80761987435a8e022049f1c316242ff34feb3e767d00e02cc9ac5545
b1849fd6cebb34d095aefd4ce301410427a5ebe8b8af49e2afd490eb5ce05469b535bb433ea0bd176eee4977252e8a05961a
9f6a58c16dcd2be136c9fe73bc35689673bd8e5e3cb21a68c1e7367fc7e3ffffffff18b749f5f72b82ce138e12575f00bdf5
af408bd6eafa766f62c4846d304d700f000000008a47304402203c62c8893223e9a8abff7a983e2f569806d5f1f1cb954f25
fbeedd204f1c67f302201d0eaecec9e280bca7d5cc395277e4b8be07d1de34980d4860de202f3db59f8103410427a5ebe8b8
af49e2afd490eb5ce05469b535bb433ea0bd176eee4977252e8a05961a9f6a58c16dcd2be136c9fe73bc35689673bd8e5e3c
b21a68c1e7367fc7e3ffffffff17ae6c92c10c80ee0bccee9f6bfc3b68c24b7d15e31d1d272da28aa8cba5b088000000008b
4830450221008623da9c40faf27cc4d3c6514f74c9ef65fba189079d2d189ba5f2400fba9b3502201801ba05f916906ba148
3a4109616a75efcc28d6976708b8d2fdd702440493ed0241043391bcbfdcc5f9a3c9bcb9b1b4f3296f55ca46c6207d6c3da8
eba4e2ec8b1996c445397a1ac33f610669c307e5853980d7883b036f9865d5cbbdb7564ed25894ffffffff897e9a3fc54a07
1db4fb7369672c5e905fd9138561a51fb26f4a4af09e5fd94f000000008b483045022100ea7d40b54efe8bcadb03a9d80f9d
2083a8e2514b28d408742775b8b48b888b6d0220520a8acc1492c6e227a2eedda054c2ab0da0c84be09b70caa06639798669
6a7c814104ae8d673db4f5d11cf9f39eaa801dce86babff2f9cadf504260f01e7c017855e3fdf1a5fad9a3413a4bf81a9e4b
8ef47fda1a08cc6e58605779513e295dd5797bffffffff0224825c00000000001976a9149f9a7abd600c0caa03983a77c8c3
df8e062cb2fa88ac384a00000000000001976a9142cd2c0ac48f6383c72ca45a66fffa37a26b38d7288ac00000000
```

Signature verification : We wrap up with a python code that produces the message associated with a UTXO of type P2PKH or P2SH-MS and verifies the validity of a signature on this message. The code is for education only and is not optimized for efficiency. We will rely on the following class and methods previously introduced:

- Methods **change_Endianness**, **int2bytes**, **get_Serialized_Tx** from section 2.
- Method **parse_Hexstr** and class **BTC_TX** introduced in section 5.
- Methods **ecdsa_Verify**, **decode_DER_Signature**, and **extract_Pubkey** introduced in *"Bitcoin Elliptic Curve Digital signature Algorithm (ECDSA)"*.

First, we start by defining the various sighash bytes:

```
_ALL = 1                                          # Sign all inputs and all outputs
_NONE = 2                                         # Sign all inputs but none of the outputs
_SINGLE = 3                                       # Sign all inputs and only the output with same
                                                  # -- index as curr input
_ANYONECANPAY = 128                               # Sign only curr input (combine with other flags)

_ALL_ANYONECANPAY = _ALL + _ANYONECANPAY          # Sign only curr input, and all outputs
_NONE_ANYONECANPAY = _NONE + _ANYONECANPAY        # Sign only curr input, and none of the outputs
_SINGLE_ANYONECANPAY = _SINGLE + _ANYONECANPAY    # Sign only curr input, and output with same
                                                  # -- index as curr input
```

Next, we define method **modify_Tx (tx, i_ind)** that builds a UTXO's message based on its sighash byte (we only consider P2PKH and P2SH-MS). It takes two arguments:

1. An object **tx** of type BTC_TX which holds the current Bitcoin transaction.
2. Index **i_ind** of the input whose scriptSig's signature needs to be verified.

It returns a five-tuple consisting of:

1. An array of the modified Bitcoin transactions in deserialized form. We choose an array because a UTXO may have many signatures (e.g., P2SH-MS) not required to share the same sighash and hence resulting in different messages.
2. An array of the serialized form of the modified Bitcoin transactions.
3. An array of the SHA256 output when applied to the modified Bitcoin transactions.
4. An array of the relevant signatures associated with the input.
5. An array of the relevant public keys associated with the input.

```
def modify_Tx (tx, i_ind):

    '''1) Limit to P2PKH and P2MS types'''
    assert (i_ind in range(tx.inputs))
    decomp_scriptSig = tx.decomp_ScriptSig()
    assert("Signature" and "Public Key" in                  # A key value of "Other data" indicates that
        decomp_scriptSig[i_ind].keys())                     # -- the transaction is not P2PKH or P2SH-MS
    prev_tx = tx.get_Prev_Tx_Deserialized()


    '''2) Get scriptPubKey/redeemScript'''
    if ("Redeem Script" in                                  # A key value of "Redeem Script" indicates
        decomp_scriptSig[i_ind].keys()):                    # -- a P2SH-MS type. Extract redeemScript
        prev_script = decomp_scriptSig[                     # -- of the relevant previous tx output
            i_ind]["Redeem Script"]

    else:
        prev_script = prev_tx[i_ind].scriptPubKey[          # Extract scriptPubKey of the relevant previous
            tx.prev_out_index[i_ind]]                       # -- tx output


    '''3) Get sig(s), pubkey(s) and sigbyte'''
    sig = decomp_scriptSig[i_ind]["Signature"]              # Array of signatures
    sighash_array = [
        int(sig[i][-2:],16) for i in range(len(sig))]       # Array to account for multisig case where
                                                            # -- each signatures may not have same SIGHASH
    pubkey = decomp_scriptSig[i_ind]["Public Key"]

    '''4) Set inputs' scriptSig to empty string'''
    tx_mod = copy.deepcopy(tx)
    for i in range(tx.inputs):
        tx_mod.set_ScriptSig(i,'')


    '''5) Update scriptSig of relevant input'''
    tx_mod.set_ScriptSig(i_ind, prev_script)

    '''6) Conduct updates based on sighash flag'''
    tx_mod_array = [tx_mod]*len(sig)                            # P2SH-MS inputs have more than 1 sig and
                                                                # -- each one will have its own message.

    for j in range(len(sig)):
        assert(sighash_array[j] in [_ALL, _NONE, _SINGLE,
            _ALL_ANYONECANPAY, _NONE_ANYONECANPAY,
            _SINGLE_ANYONECANPAY])


        if (sighash_array[j] - _NONE in [0,_ANYONECANPAY]): # Sighash in {"NONE", "NONE_ANYONECANPAY"}

            for i in range(tx_mod.inputs):
                if (i != i_ind): tx_mod.nSequence[i] = 0    # nSequence of non-current inputs set to 0

            tx_mod_array[j].reset_Outputs()                 # Set the outputs to the empty vector


        if (sighash_array[j] - _SINGLE in
                [0,_ANYONECANPAY]):                         # Sighash in {"SINGLE","SINGLE_ANYONECANPAY"}

            assert (i_ind in range(tx.outputs))             # Ensure more ouputs than inputs
            for i in range(tx_mod.inputs):
                if (i != i_ind):                            # The nSequence number of each input other
                    tx_mod_array[j].nSequence[i] = 0        # -- than the current one is set to 0

            tx_mod_array[j].reset_Outputs()
            tx_mod_array[j].outputs = i_ind + 1             # Update vout count

            k = 0
            while (k < i_ind):                              # All outpus with index less than that of
                tx_mod_array[j].value.append(int(           # -- the current input are updated with
                    'ffffffffffffffff',16))                 # -- a value of -1 and an empty string
                tx_mod_array[j].scriptPubKey.append('')     # -- locking script
                k += 1

            tx_mod_array[j].value.append(                   # The output whose index matches that of the
                tx.value[i_ind])                            # -- current input maintains its original
            tx_mod_array[j].scriptPubKey.append(            # -- and its original locking script
                tx.scriptPubKey[i_ind])


        if (sighash_array[j] - _ANYONECANPAY in             # Sighash in {SINGLE_ANYONECANPAY",
            [_ALL,_NONE, _SINGLE]):                         # -- "NONE_ANYONECANPAY","ALL_ANYONECANPAY"}

            for i in reversed(range(tx_mod.inputs)):        # Remove all inputs except for current one
                if (i != i_ind):
                    del tx_mod_array[j].prev_tx_id[i]
                    del tx_mod_array[j].prev_out_index[i]
                    del tx_mod_array[j].scriptSig[i]
                    del tx_mod_array[j].nSequence[i]
            tx_mod_array[j].inputs = 1                      # Update vin count to 1
```

```
    ''' 7) Append sighash (4-byte format)'''
    sigbyte_array = [change_Endianness(
        int2bytes(sighash, 4)) for sighash in
        sighash_array]

    tx_mod_raw_array = [tx_mod_array[j].serialize()
        + sigbyte_array[j] for j in range(len(sig))]

    '''8) Transform to base256 and take sha256'''
    tx_mod_raw_256_array = [tx_mod_raw.decode(
        'hex') for tx_mod_raw in tx_mod_raw_array]

    tx_mod_raw_sha256_array = [hashlib.sha256(
        tx_mod_raw_256).digest() for tx_mod_raw_256 in
        tx_mod_raw_256_array]


    return tx_mod_array, tx_mod_raw_array, \
        tx_mod_raw_sha256_array, sig, pubkey
```

Note that we only performed a single instead of a double SHA256 on the message. This is because the ECDSA signature algorithm will later perform an additional SHA256 prior to signing the message.

The last step is to verify the validity of a signature given a message and an appropriate public key. Method **check_Sig(txid, i_ind, net)** does precisely this. It takes three arguments namely, a **txid**, an input index **i_ind** in order to specify which UTXO to consider, and a network type **net** to clarify if the Bitcoin transaction is on mainnet or testnet. Only UTXOs of type P2PKH or P2SH-MS are tolerated. The method outputs a Boolean value indicating if the signature(s) associated with the specified UTXO is (are) valid:

```
def check_Sig(txid, i_ind, net = "mainnet"):

    '''Form the modified message to sign'''
    tx_raw = get_Serialized_Tx(txid, net)                    # Get serialized hex format of current tx
    tx = BTC_TX(net)                                         # Create tx instance to hold current tx
    tx.deserialize(tx_raw)
    tx_mod_raw_sha256_array, sig, pubkey = modify_Tx(
        tx, i_ind)[2:]


    '''Extract signature-relevant data'''
    r_dec, s_dec, h_dec = [], [], []
    for i in range(len(sig)):
        r, s, h = decode_DER_Signature(sig[i])[3:]
        r_dec.append(r)
        s_dec.append(s)
        h_dec.append(h)


    '''Extract public key relevant data'''
    H = []
    for i in range(len(pubkey)):
        H.append(extract_Pubkey(pubkey[i]))


    '''Run the ecdsa verification algorithm'''
    sig_val_array = []
    for j in range(len(sig)):
        for k in range(len(pubkey)):
            sig_val_array.append(ecdsa_Verify(
                r_dec[j], s_dec[j], H[k],
                tx_mod_raw_sha256_array[j]))

        if (True not in sig_val_array):
            return False
        sig_val_array = []

    return True
```

Recall from section 3 and opcode OP_CHECKMULTISIG that the signatures associated with a P2SH-MS UTXO are ordered in the same way as their corresponding public keys

in the scriptSig field. The ECDSA verification algorithm in the method above uses this observation to validate a UTXO's signature(s).

We illustrate below how one can run the method to verify the validity of signatures in Bitcoin transaction with txid:

0x039b145453739a8d58198eb9caa63eb9db9a8bb08cbd0977237e05082561a4a5

This Bitcoin transaction has three inputs with respective UTXOs of type P2SH-MS.

```
txid = "039b145453739a8d58198eb9caa63eb9db9a8bb\
08cbd0977237e05082561a4a5"

net =  "mainnet"

tx_raw = get_Serialized_Tx(txid, net)
tx = BTC_TX(net)
tx.deserialize(tx_raw);
inputs = tx.inputs

for i in range(inputs):

    '''Conduct signature verification'''
    if (check_Sig(txid, i, net) == True):
        print "\nInput v_in #",i, "has a valid signature"

    else:
        print "\nInput v_in #",i, "has an invalid signature"


Input v_in # 0 has a valid signature

Input v_in # 1 has a valid signature

Input v_in # 2 has a valid signature
```

8 Transaction malleability

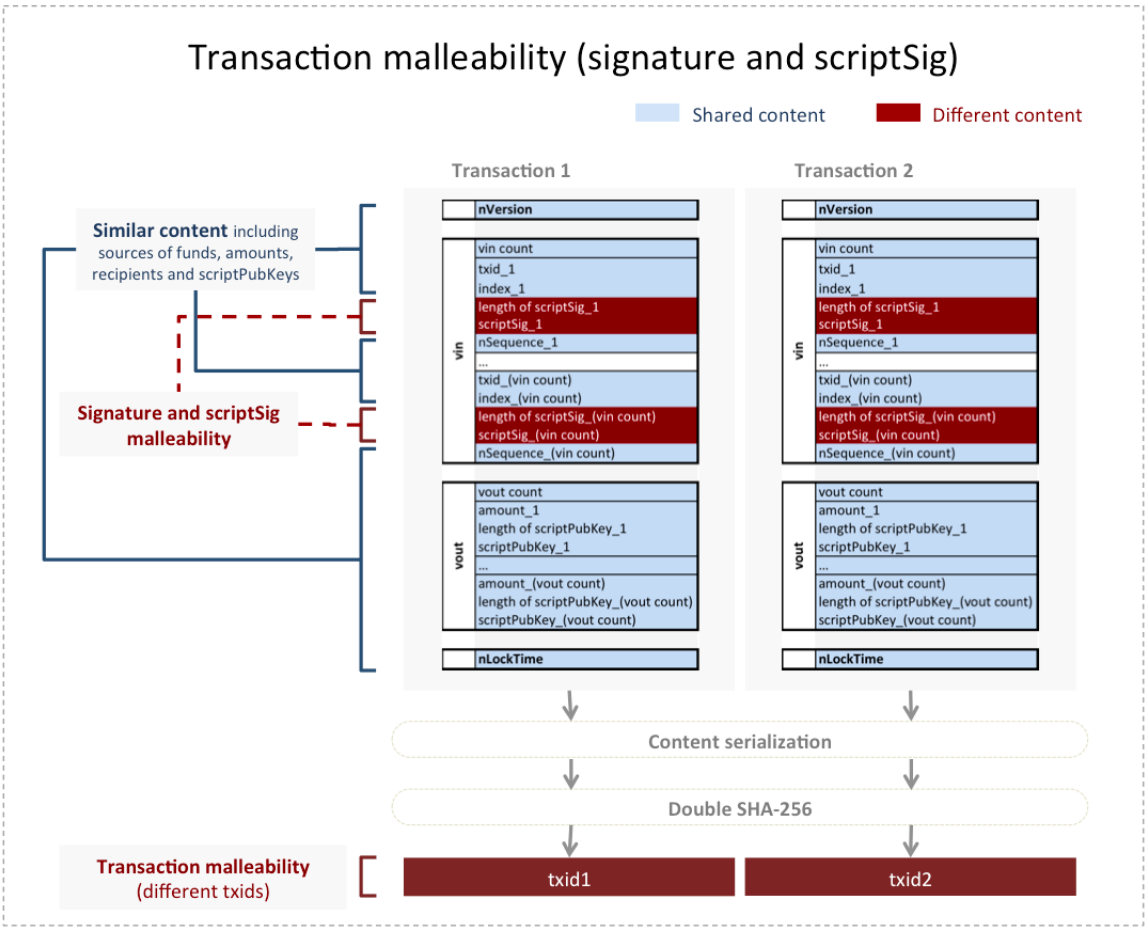
In section 2, we saw that the **txid** of a Bitcoin transaction is obtained from the double SHA256 of its serialized representation. By virtue of being a hashing function, SHA256 exhibits collision resistance. This property, described in the chapter *"Digital Signature and Other Prerequisites"*, ensures with overwhelming probability that any modification to the content of the Bitcoin transaction will result in a different txid.

Functionally equivalent Bitcoin transactions: An important observation is that two Bitcoin transactions with two different txid could still be **functionally identical**. We say that transactions TX_1 and TX_2 are functionally equivalent if and only if the following conditions hold:

- TX_1 and TX_2 share the same set of inputs. This means that each UTXO referenced by TX_1 (respectively TX_2) is also referenced by TX_2 (respectively TX_1). Furthermore, each input in TX_1 (respectively TX_2) and its homologous counterpart in TX_2 (respectively TX_1) share the same nSequence number.
- The output sets of TX_1 and TX_2 are identical. In other words, each output of TX_1 (respectively TX_2) specified by a Satoshi amount and scriptPubKey is also an output of TX_2 (respectively TX_1).
- TX_1 and TX_2 share the same nLockTime value.

Simply put, the functional equivalence of two Bitcoin transactions means that they both use the same sources of funds, intend to send identical amounts to the same recipients, and encumber their outputs with identical locking conditions.

Note that the functional equivalence conditions exclude any constraint on the scriptSig fields of the various inputs. A scriptSig field would usually contain signature(s) applied to specific Bitcoin transaction content (in line with what was described in section 7) and the message to sign can not be expected to contain its signature in advance. It turns out that functionally equivalent Bitcoin transactions can have different valid scriptSigs resulting in different txids. This phenomenon is commonly referred to as **transaction malleability** and is summarized in the graph below:



Signature and scriptSig malleability: In the chapter entitled *"Bitcoin Elliptic Curve Digital Signature Algorithm (ECDSA)"* we discussed different instances of ECDSA signature malleability. In particular we described three instances of malleability caused by:

1. **Non-DER encoded ECDSA signature** (addressed in BIP 66).
2. **ECDSA's inherent signature construct** (addressed in Pull Request #6769).
3. **ECDSA's reliance on the random parameter** k.

However, signatures are not the only source of Bitcoin transaction malleability. The latter could result from any modification to the scriptSig content as long as the

scriptSig evaluation remains valid. One such example consists in pushing additional data at the beginning of a valid scriptSig associated with a given scriptPubKey. Since the additional data push does not get consumed by the scriptPubKey, the resulting scriptSig's evaluation would still be valid. Note that this particular malleability instance can be resolved by imposing a more restrictive consensus rule that invalidates any Bitcoin transaction whose scriptPuKey evaluation results in more than a single non-zero value on the stack. However, while some malleability sources could be addressed by tightening consensus rules, other sources may be more difficult to mitigate.

Sighash malleability: Our definition of functional equivalence is relevant to the case of a Bitcoin transaction signed with the SIGHASH_ALL flag. In such cases, scriptSig and signature malleability are the only sources of Bitcoin transaction malleability. However, if less restrictive sighash flags were used (as outlined in section 7), our definition of functional equivalence can be further relaxed. For example, a Bitcoin transaction with a single input signed using the SIGHASH_NONE flag can be considered functionally equivalent to any other Bitcoin transaction that uses the same input and nLockTime value, independently of its choice of output(s). In such instances, transaction malleability could result from modifying content pertaining to any of the transaction's outputs. However, contrary to most types of scriptSig malleability, the sender has the ability to circumvent sighash malleability by signing her UTXOs using SIGHASH_ALL.

Transaction malleability could lead to malicious behavior : We end this section with two examples where a malicious party could leverage transaction malleability to steal funds or to prevent a legitimate party from accessing her funds. The first example describes a typical malleability attack such as the one that some suspect was used against MtGox leading to the exchange's closure in February 2014. The second example is a hypothetical scenario that demonstrates how transaction malleability could lead to counter-party risk that can jeopardize the proper functioning of unconfirmed transaction dependency chains.

1. **Example #1:** By definition, a malleable signature scheme could lead to the creation of two valid but different signatures applied to the same Bitcoin transaction. Such an event would cause the Bitcoin network to end up with at least two different txids referencing the same content. Such a situation could motivate a specific type of attack known as a **malleability attack**. The gist of it is as follows:

   (a) Suppose Alice issues a BTC payment to Bob. Let $txid_1$ be its transaction id.

   (b) Suppose that Bob alters the signature of Alice's transaction (assuming it is a malleable scheme) right before $txid_1$ gets any confirmation on the blockchain. This alteration results in a new transaction id, namely $txid_2$, on the same content (i.e., the intended recipient is still Bob, the funding UTXOs are still the same, and the amount remains as is).

   (c) If $txid_2$ gets confirmed on the blockchain before $txid_1$, the latter will become orphaned. If Alice does not have the required level of sophistication to track

UTXOs on the blockchain in order to verify that her original UTXOs have been spent, she will rely instead on the confirmation status of $txid_1$. Given that it was orphaned, she will conclude that the funds never reached Bob's address.

(d) Bob could then defraud her by asking her to issue a new payment knowing that he would have already received the intended funds by virtue of $txid_2$ being confirmed. He would then receive twice the intended amount.

The above malleability attack can be interpreted as a double-spending instance, although the malicious party in this case is the receiver and not the sender.

2. **Example #2 (unconfirmed transaction dependencies):** It is possible for an unconfirmed Bitcoin transaction TX_2 (i.e., not yet broadcasted on the network) to have at least one of its inputs reference a UTXO corresponding to an output of yet another unconfirmed Bitcoin transaction TX_1. Such protocols rely on a chain of dependencies between unconfirmed transactions and are particularly vulnerable to transaction malleability attacks. To see why, suppose that TX_1 gets broadcasted and caught by a malicious node who then malleates it yielding a functionally equivalent transaction with a different txid $TX_1^{'}$. If $TX_1^{'}$ gets confirmed on the network before TX_1, the dependency of TX_2 on TX_1 becomes futile, rendering TX_2 irrelevant. In this example we describe a hypothetical scenario showcasing such dependencies between unconfirmed Bitcoin transactions.

   Suppose Alice and Bob got recently married and decided to create a joint savings account. Contrary to Bob, Alice succeeded in amassing a small fortune over the past few years and agreed to initially fund their joint account by transferring BTC 5 to it. Any spending from this account requires both their signatures and to that end, they created a 2-of-2 multisig P2SH address.

   Now suppose Alice broadcasts her BTC 5 funding transaction to the network. For some reason, suppose that the couple's relationship deteriorated rapidly leading to their divorce prior to making any purchase from their joint account. A revengeful Bob could decide to hold Alice's funds captive unless she pays him a certain amount. The possibility of such an unfortunate turn of events pushes Alice to implement better protective measures. In order to do so, she proceeds as follows:

   - Alice prepares her funding transaction such that one of its outputs consists of BTC 5 destined to the multisig address that she controls with Bob. She signs this Bitcoin transaction and obtains a txid TX_1. However she refrains from broadcasting it to the network.
   - Alice then prepares another transaction whose single input references the BTC 5 UTXO output of TX_1 destined to be sent to an address that she fully controls.
   - Alice subsequently asks Bob to provide his signature for the single input of this second transaction (recall that this single input is actually encumbered with a 2-of-2 multisig redeemScript requiring both Alice and Bob's signatures to be unlocked).

- Once Alice receives Bob's signature, she would produce hers on that single input completing as such this second transaction whose txid can now be calculated as TX_2.

With such a scheme, Alice feels safer to then broadcast TX_1. Her rationale is that in case Bob attempts to withhold the funds from her, she can always broadcast TX_2 and regain her original BTC 5. While Alice initially believed that this new scheme offered enough protection, she quickly realized that unconfirmed transaction TX_2 exhibits a dependency on unconfirmed transaction TX_1, and that her scheme can be vulnerable to a malleability attack. Indeed, a malicious Bob could run a node and listen to transactions broadcasted by Alice. If she sends TX_1 to the network, he could catch it soon enough to malleate it and broadcast a functionally equivalent Bitcoin transaction with txid $TX_1^{'} \neq TX_1$. If the network ends up validating $TX_1^{'}$, Alice's TX_2 becomes irrelevant and her protective scheme futile.

What if Alice and Bob's conjugal problems start after a purchase is successfully conducted? Suppose for instance that the couple purchased BTC 2 worth of furniture (leaving a balance of BTC 3 in their savings account) but then their differences grew apart leading to a separation. To protect herself against such a possibility, Alice devises a similar scheme as the one presented earlier:

- Alice prepares her purchasing transaction with a single input referencing the BTC 5 UTXO output of TX_1 and outputs consisting of BTC 2 destined to the furniture shop owner and BTC 3 destined to the multisig address that she controls with Bob.
- Alice subsequently asks Bob to provide his signature for the single input of this transaction (recall that this single input is actually encumbered with a 2-of-2 multisig redeemScript requiring both Alice and Bob's signatures to be unlocked).
- Once Alice receives Bob's signature, she would produce hers on that single input completing as such this first transaction whose txid can now be calculated as TX_3. However she refrains from broadcasting it to the network.
- Alice then prepares another Bitcoin transaction whose single input references the BTC 3 UTXO output of TX_3 destined to be sent to an address that she fully controls.
- Alice subsequently asks Bob to provide his signature for the single input of this second transaction (recall that this unique input is actually encumbered with a 2-of-2 multisig redeemScript requiring both Alice and Bob's signatures to be unlocked).
- Once Alice receives Bob's signature, she would produce hers on that single input completing as such this second transaction whose txid can now be calculated as TX_4.

Alice is tempted to feel safer about broadcasting TX_3 now that she has TX_4 to protect her in case Bob decides to hold the BTC 3 balance captive. However, here too, unconfirmed transaction TX_4 exhibits a dependency on unconfirmed

transaction TX_3, leading to a similar vulnerability. Indeed, a malicious Bob could malleate TX_3 into $TX_3^{'}$ and potentially render TX_4 useless.

In order to address malicious behavior derived from transaction malleability, Bitcoin developers devised a clever solution that removes all instances of non-intentional malleability. By non-intentional we mean cases that exclude sighash malleability as well as malleability derived from different signatures generated on the same UTXO message by its legitimate owner. The solution known as **Segregated Witness** or **Segwit** paved the way to an improved Bitcoin transaction architecture that dissociates the scriptSig fields (i.e., witness fields) from the rest of the Bitcoin transaction. We will dedicate a separate chapter to Segwit transactions.

One of the important achievements of Segwit in so far as the removal of transaction malleability is concerned, is that it allowed for an effective implementation of the **Lightning Network** which inherently relies on unconfirmed transaction dependency chains. The importance of the Lightning Network in addressing the Bitcoin scalability challenges deserves a separate chapter that we will publish in the near future.

References

[1] Script. https://en.bitcoin.it/wiki/Script, March 2019.

[2] P2sh statistics. https://www.p2sh.info/dashboard/db/p2sh-repartition-by-type?orgId=1, Accessed 9 August 2019.

[3] almel. explanation of what an op return transaction looks like. https://bitcoin.stackexchange.com/questions/29554/explanation-of-what-an-op-return-transaction-looks-like), July 2014.

[4] Gavin Andresen. Block v2, height in coinbase. https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki, July 2012.

[5] Gavin Andresen. Pay to script hash. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki, January 2012.

[6] Boaz Barak. Zero knowledge proofs. https://bit.ly/326dX4c.

[7] Massimo Bartoletti and Livio Pompianu. An analysis of bitcoin op return metadata. *Universita degli Studi di Cagliari*, 2017.

[8] Sean Bowe. First zero-knowledge contingent payment (zkcp). https://www.youtube.com/watch?v=ONUsnRgLVB8, February 2016.

[9] Sean Bowe. pay-to-sudoku. https://github.com/zcash-hackworks/pay-to-sudoku, 2016.

[10] BtcDrak, Mark Friedenbach, and Eric Lombrozo. Bip 112 - checksequenceverify. https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki, August 2015.

[11] Mark Friedenbach, BtcDrak, Nicolas Dorier, and kinoshitajona. Bip 68 - relative lock-time using consensus-enforced sequence numbers. https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki, May 2015.

[12] David A. Harding and Peter Todd. Bip 125 - opt-in full replace-by-fee signaling. https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki, December 2015.

[13] Thomas Kerin and Mark Friedenbach. Bip 113 - median time-past as endpoint for lock-time calculations. https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki, August 2015.

[14] Gregory Maxwell. The first successful zero-knowledge contingent payment. https://bitcoin-rpc.github.io/en/2016/02/26/zero-knowledge-contingent-payments-announcement/, February 2016.

[15] Sergi Delgado Segura. Bitcoin tools. https://github.com/sr-gi/bitcoin_tools/blob/master/bitcoin_tools/utils.py, March 2018.

[16] Nick Szabo. Smart contracts. https://bit.ly/2rLG2Nr, 1994.

[17] Nick Szabo. Formalizing and securing relationships on public networks. https://nakamotoinstitute.org/formalizing-securing-relationships/, 1997.

[18] Peter Todd. Bip 65 - checklocktimeverify. https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki, October 2014.

[19] Peter Todd and Amir Taaki. Paypub: Trustless payments for information publishing on bitcoin. https://github.com/unsystem/paypub/blob/master/EXPLANATION, May 2014.