Bitcoin Transaction Malleability

Sidi Mohammed Boughalem (shinokiz@gmail.com)

10. July 2019

The Bitcoin network is a peer-to-peer system that has participants from all over the Internet. The Bitcoin protocol requires participating nodes to retain and update all transaction records; this ensures that all Bitcoin activities are accessible from a consistent transaction history database. Nevertheless, The Bitcoin global cryptocurrency system has been the subject of several criminal cases that exploited several flaws. We investigate in this paper one of them, known as Transaction Malleability. We then provide the analysis of the famous Mt.Gox incident following Decker and Wattenhofer [1].

1 Transaction malleability

Bitcoin payments are encoded as transactions that eventually become part of the blockchain. Each user can create a number of *addresses* that can be used to send, receive and prove ownership of bitcoins. A bitcoin address is the hash of a public key, i.e., a 256-bits integer that comes from an ECDSA key pair (r, s) (where the public key is calculated from a private key).

Definition 1.1 (Transaction)**.** *a Bitcoin transaction is a signed data structure that consists of one or more* ***inputs*** *(specifies which bitcoins will be transferred) and an ordered list of one or more* ***outputs*** *(specifies the address that should be credited with the bitcoins being transferred).*

Formally, an output is a tuple comprising the value that is to be transferred and a locking script, while an input includes the hash of a previous transaction, an index, and an unlocking script. The hash and index form a reference that uniquely identifies the output to be claimed and the unlocking script proves that the user creating the transaction is indeed the owner of the bitcoins being claimed.

Seminar „Mathematical Literacy: Cryptocurrencies “, SS 2019, Universität Regensburg

Example 1.2. *An example of a bitcoin transaction found in the blockchain ([10]):*

```
Input:
Previous tx: f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b009ca73dd04470b9a6
Index: 0
scriptSig: 304502206e21798a42fae0e854281abd38bacd1aeed3ee3738d9e1446618c4571d
1090db022100e2ac980643b0b82c0e88ffdfec6b64e3e6ba35e7ba5fdd7d5d6cc8d25c6b241501

Output:
Value: 5000000000
scriptPubKey:
DUP HASH160 404371705fa9bd789a2fcd52d2c580b65d35549d EQUALVERIFY CHECKSIG
```

Every transaction is serialized, i.e. it is translated into a format that on the one hand, it can be more easily stored and transmitted across a network and on the other hand, it can be referenced as follow:

Definition 1.3 (Transaction ID)**.** *a transaction identifier (Txid) is a unique 32-byte alphanumerical string of data that is used to reference a bitcoin transaction. The transaction identifier is formed by hashing transaction data through the SHA-256 hash function twice. That is:*

$$Tx\ id = SHA\text{-}256(SHA\text{-}256(\textbf{\textit{Transaction data}}))$$

Here, the transaction data consist of

1. *Transaction Hash: Reference to the transaction containing the UTXO being spent.*
2. *Output Index: The index number of the UTXO being spent.*
3. *Unlocking Script: Script that satisfies conditions of the locking script.*
4. *Unlocking Script Size: Size of the unlocking script in bytes.*
5. *Sequence Number.*

Example 1.4. *Here is an example of a Txid [7]:*

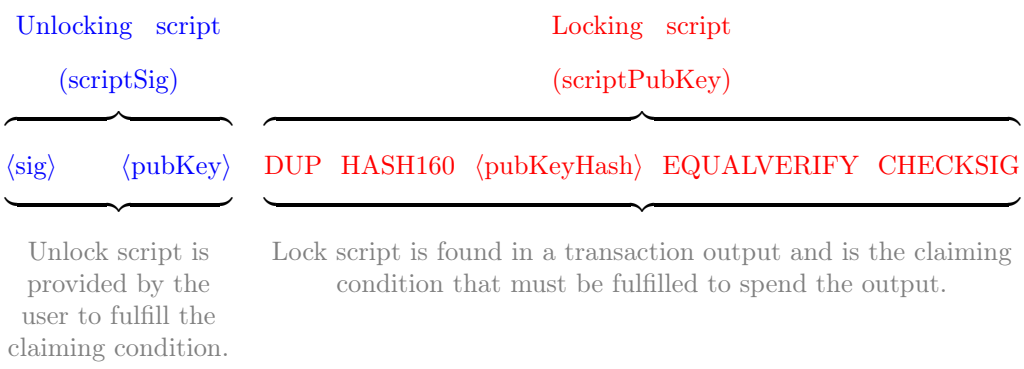
$$adae0270457bad95152c5ae7771b50fae06afa01edeefca4201689e7c99e0b19$$

In other words, one can see the *Txid* as some sort of "convenience" for purely human interaction; indeed, they affect in no way the way the block chain works. If one acquires some goods in some e-commerce website using Bitcoin as payment, then in case of any possible reason the website might lose the transaction (software issues for an example), they can manually check in blockchain whether or not your transaction has been validated, providing your *Txid*. What is relevant however in Definition 1.3, is that the *Txid* depends on the unlocking script (3.).

Definition 1.5 (Bitcoin Script)**.** ***scriptPubKey*** *is a locking script placed on the output of a Bitcoin transaction that requires certain conditions to be met in order for a recipient to spend his/her bitcoins; scriptPubKey is also known as PubKey Script outside of the Bitcoin code. Conversely,* ***scriptSig*** *is the unlocking script that satisfies the conditions placed on the output by the scriptPubKey, and is what allows it to be spent; outside of code, scriptSig is also known as Signature scripts.*

Both scriptPubKey and scriptSig are written in *Script*, the programming language used for constructing bitcoin transactions. Script is a simple and stack-based. It process from left to right and is intentionally, not Turing-complete (in particular, it has noo loops). The lack of functionalities makes Script more secure for Bitcoin transactions as it can only perform a limited number of operations.

The vast majority of transactions use a standard script that set up a claiming condition requiring the claiming script to provide a public key matching the address and a valid signature of the current transaction matching the public key, here is an example of the standard locking script and unlocking script as used by simple transactions transferring bitcoins to an address backed by a single public key: [5]



As one can see, the unlocking script, scriptSig, contains a sig (digital signature) and a pubKey (public key) which must be provided in order for the locking script to be satisfied. Conversely, the locking script, scriptPubKey, contains a pubKeyHash which corresponds as seen before to a Bitcoin address. The process works such that, the scriptSig and scriptPubKey are combined and executed in sequence, with the unlocking script being executed first. For example, Alice sends Bob 1 BTC. When Bob decides to spend the that 1 BTC that he received from Alice, he must first unlock the outputs, which then become locked when the recipient receives his 1 BTC.

Now we take a deeper look at how exactly these script work. Figure 1 shows the script code of scriptSig, and Figure 2 the script code of scriptPubKey.

```
OP_PUSHDATA*
<sig>
OP_PUSHDATA*
<pubKey>
```

Figure 1: scriptSig

```
OP_DUP
OP_HASH160
OP_PUSHDATA*
<pubKeyHash>
OP_EQUALVERIFY
OP_CHECKSIG
```

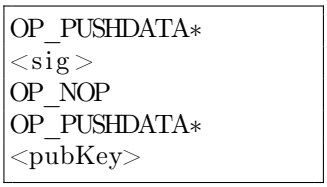
Figure 2: scriptPubKey

Here is how the magic happens: Suppose we have a transaction T_n that has the output of a previous transaction T_{n-1}. scriptSig of T_n pushes the signature and the public key on the stack. The scriptPubKey of T_{n-1} duplicates the public key (OP_DUP), hash it (OP_HASH160) this 20 byte derivative of the public key is also encoded in the address then pushes it on the stack. The two top elements of the stack (i.e. the signature and the hashed public key) are then tested for equality (OP_EQUALVERIFY). If the hash of the public key and the expected hash match, the script continues, otherwise the execution is aborted. Finally, these last two verify that scriptSig signs T_n (OP_CHECKSIG). Figure 3 shows a list of opcodes used in both figure above.

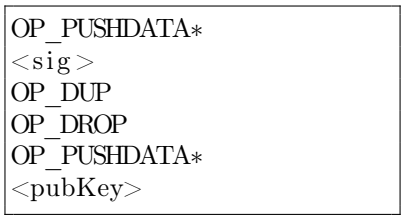
Command	**Input**	**Output**	**Description**
OP_DUP	x	xx	Duplicates the top stack item.
OP_HASH160	in	hash	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_EQUALVERIFY	x_1x_2	$\emptyset$	Returns 1 if the inputs are exactly equal, 0 otherwise, then marks transaction as invalid if top stack value is 0. The top stack value is removed.
OP_CHECKSIG	sig pubKey	True/False	Returns 1 if the signature used is a valid signature for the hash and public key, 0 otherwise.

Figure 3: List of OP codes used in scriptSig and scriptPubKey, [8]

When a transaction is propagating in the network, the nodes that encode it can *mutate* scriptSig without altering the signature: in the following scripts, the encoded signature is valid in both cases but the hash identifying the transaction is different. (One can execute the following scripts in [9], with the help of Figure 4.)



Mutating scriptSig with the operation OP_NOP



Mutating scriptSig with the operations OP_DUP/OP_DROP

Command	**Input**	**Output**	**Description**
OP_NOP	Nothing	Nothing	Does nothing.
OP_DUP	x	xx	Duplicates the top stack item.
OP_DROP	x	Nothing	Removes the top stack item.

Figure 4: List of OP codes used in the mutations of scriptSig, [8]

Thus, the transaction will have the same effect, but the hash identifying the transaction will change. To observe how this is possible, we will closely investigate the OP_PUSHDATA operations (Figure 5). OP_PUSHDATA basically specifies a number of bytes to be pushed as a string on the stack. The simplest one OP_0 encodes the length of the string in a single byte (with value between 0x00 and 0x4b).

Command	**Input**	**Output**	**Description**
OP_0	Nothing	$\emptyset$	An empty array of bytes is pushed onto the stack.
OP_PUSHDATA1	(special)	data	The next byte contains the number of bytes to be pushed onto the stack.
OP_PUSHDATA2	(special)	data	The next two bytes contain the number of bytes to be pushed onto the stack in little endian order.
OP_PUSHDATA4	(special)	data	The next four bytes contain the number of bytes to be pushed onto the stack in little endian order.

Figure 5: List of OP_PUSHDATA, [8]

Now if one replaces the (OP_0) that pushes the public key on the stack with (OP_PUSHDATA2), the claiming script is changed from

```
0x52<sig>86<pubKey>
```

to

```
0x4D5200<sig>4D8600<pubKey>
```

If the unlocking script is changed, the serialized transaction data will be different, therefore, the resulting *Txid* will also be different. These changes in the way pushes are encoded[1], along with the previous examples constitute a *malleability* source, which makes the script vulnerable for external attacks.

Definition 1.6 (Transaction malleability)**.** *Transaction malleability is the process of changing the unique identifier Txid of a transaction by first changing the digital signature used to create it.*

Now let us illustrate how a transaction malleability attack occur. Suppose our dearest *Alice* creates a bitcoin payment transaction, and sends it to her peers. As mentioned before, nothing prevents Alice's peers from mutating her transaction when they propagate it, thus changing its *Txid*. Suppose that *Chuck*, one of Alice's peers, out of malicious intent wants to initiate a malleability attack on her. As the inputs, outputs, and payment amount are all cryptographically signed, Chuck can not steal money or make any semantic changes to the transaction. What Chuck does though, is change the way Alice's *Txid* is computed, then broadcast the transaction with a new *Txid* to the rest of the network. Here it boils down to which transaction gets validated first: the original transaction created by Alice and relayed by her *good peers*, or the modified version created by the malicious Chuck.
Most Bitcoin clients have an option to show users the *Txid* after they send a transaction. As Bitcoin transactions take some time to actually be confirmed as part of the Blockchain, the clients periodically check for the if the transaction has actually made it to the Blockchain or not (by checking if the expected *Txid* has been added to a new block). If a transaction malleability attack occurs, and the *Txid* changes, then the transaction will eventually be added to the Blockchain, but under an unexpected *Txid*. This can confuse client softwares that were looking for a particular *Txid*.

Remark 1.7. *An attacker can only alter the digital signature of the unlocking script prior to the confirmation of a block. After confirmation, the digital signature, and therefore the transaction id, are immutable.*

Let us illustrate all that by the following concrete example:

Example 1.8. *Alice sends* 1 *BTC with Txid* $= A$*. Bob, being a part of Alice's pairs, modifies the transaction so the new Txid is* $= B$*. The transaction makes it to the Blockchain under Txid B and gets successfully added to the Blockchain, which at the same time invalidates the previous transaction under Txid A. In the meantime, Alice's*

[1] It is commonly known as **Push operations in scriptSig of non-standard size type**, see [2].

client software keeps checking for Txid A, but will never find it. Nevertheless, Alice's wallet software will debit 1 *BTC from her account since the modified transaction was confirmed, and still sent* 1 *BTC from her account. At this point, Alice's software has debited* 1 *BTC from her account but cannot confirm the transaction. Alice might eventually give up and think the transaction failed for some reason, and she could retry the transaction. If she does, she will send another* 1 *BTC to the same address. In essence, Bob has tricked Alice into double paying.*

We recall that Bitcoin uses Elliptic Curve Digital Signature Algorithm, or ECDSA for short. Digital Signature Algorithm is a cryptographic algorithm that uses a pair consisting of a public key and a private key. The private key is used to generate a digital signature for a message, and such a signature can be verified by using the signer's corresponding public key. It is based on the finite fields $\mathbb{F}_q$ and $\mathbb{F}_p$ where p and q are large prime numbers, such that $q \mid (p-1)$. ECDSA is a variant of DSA that uses elliptic curves. For Bitcoin, the elliptic curve used is the well known **secp256k1** that has the following associated group $(\mathcal{E}ll(\mathbb{F}_p), \oplus_p)$ where

- $\mathcal{E}ll(\mathbb{F}_p) := \{(x,y) \in \mathbb{F}_p^2 \mid y^2 \equiv x^3 + 7 \mod p\} \cup \{\infty\}$
- p is a prime number of the order $2^{256} - 1$
- $\oplus_p$ denotes the elliptic curve point addition.

Let G be a base point of the elliptic curves, of order n (n is some prime number mod p). The DSA algorithm involves three operations:

(i) Key generation: produces a pair (P_v, P_b) where P_v is a random element in $\mathbb{F}_n^\times$ and

$$P_b = P_v \otimes_p G = G \underbrace{\oplus_p \cdots \oplus_p}_{P_v \text{ times}} G = (x_P, y_P) \quad \text{is a point in the elliptic curve.}$$

(ii) Signing algorithm: Given a message M.

- Calculate $m = hash_{256}(M)$
- Select a random $k \in \mathbb{F}_n^\times$
- Compute $P = (x_P, y_P) = k \otimes_p G$
- Set $r \equiv x_P \mod n$
- Compute $s = \dfrac{m + r \times P_v}{k} \mod n$
- Get the signature $\varphi(m) = (r, s)$

(iii) Verification algorithm: Given a signature φ, a (hashed) message m, and the public key P_b of the signer.

- If P_b does not satisfy the elliptic curve equation, output False.
- If $r \notin \mathbb{F}_n^\times$ or $s \notin \mathbb{F}_n^\times$ output False.
- Compute $u := \dfrac{m}{s} \mod n$
- Compute $v := \dfrac{r}{s} \mod n$
- Compute $Q = (x_Q, y_Q) = (u \otimes_p G) \oplus_p (v \otimes_p G)$
- If $r \equiv x_Q \mod n$ output True, otherwise output False.

There are numerous sources of malleability in the unlocking script (scriptSig). A Bitcoin Improvement Proposal (BIP) by Wuille [2] identifies some of them. We will cite only two :

1. **Non-DER encoded ECDSA signatures**: In Bitcoin, the signature (r, s) is not encoded as a concatenation rs, but rather following a DER[2] in order to make data compatible, regardless of language implementation. Thus it gives only one way to represent (r, s) as an octet string. However, the language used *OpenSSL* allows "padding": Let $r = 0x2c5$ for example. Then the DER allows r to be for example encoded as $0x002c5$. To remedy this, BIP66 provides a strict set of rules that does not allow padding, only if the MSB[3] $= 1$.
   For example : 0x83b1 would be encoded as 0x0083b1 and nothing else. For more details see [3].

2. **Inherent ECDSA signature malleability**: Due to symmetry: if (r, s) is a valid signature, then so is the complementary signature $(r, -s) \mod n$. Given a signature (r, s) it's possible to calculate the complementary signature without knowing the ECDSA private keys, in the following way:
   Let $\varphi(m) = (r, s)$ be an ECDSA signature, of a message M and $\varphi'(m) = (r, n-s)$ its complementary signature. Let $a \equiv (n-s)^{-1} \mod n$. One sees that

$$a \equiv (n-s)^{-1} \mod n \Leftrightarrow an - as \equiv 1 \mod n \Leftrightarrow -as \equiv 1 \mod n$$
$$\Leftrightarrow -s^{-1} \equiv a \equiv (n-s)^{-1} \mod n$$

   Thus, if one looks at the verification algorithm of the signature φ', one sees that

- Compute $u' := \dfrac{m}{-s} \equiv -u \mod n$
- Compute $v' := \dfrac{r}{-s} \equiv -v \mod n$
- Compute $(x_{Q'}, y_{Q'}) = Q' = -Q = (x_Q, -y_Q)$

[2] DER stands for Distinguished Encoding Rules.
[3] MSB stands for Most Significant Bit

Hence, as long as φ, $r \equiv x_Q = x_{Q'} \mod n$ and so φ' is valid as well.
As the complementary signature has a different hash, this will result in a new *Txid*. This was fixed in October 2015 through the BIP062 enforcing a canonical signature representation: both signature values are calculated, but only the signature with the smaller $|s|$ is considered valid. For more details see [2].

Transaction malleability attacks have however been mitigated on the Bitcoin network via the implementation of a soft fork protocol upgrade known as **Segregated Witness**, or SegWit [4]. It was successfully activated on the Bitcoin network on 21st July 2017. With this, the signature data in the unlocking script is moved and omitted (Figure 6) when calculating the *Txid* of the transaction data. Hence, if an attacker modifies the signature data, the *Txid* will remain exactly the same.

The next section will treat a famous example of how the transaction malleability

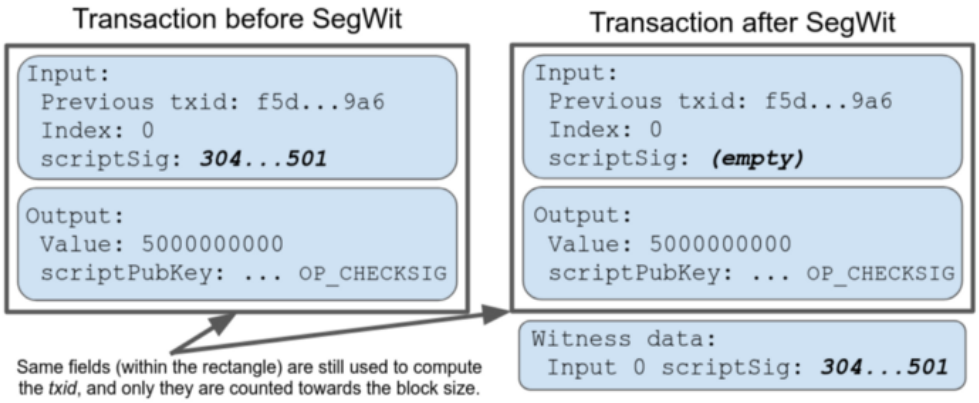


Figure 6: Transaction schema before and after SegWit [6]

exploit was used to steal bitcoins, which eventually resulted in the insolvency of the cryptocurrency exchange *MT.Gox*.

2 The MtGox Incident Timeline

The former cryptocurrency exchange, *Mt.Gox*, was the largest Bitcoin exchange and web wallet service provider in the world. On February 7, 2014, *Mt.Gox* reported technical difficulties due to hacking attacks. On February 10, 2014, *Mt.Gox* issued a press release claiming that it had lost more than 850,000 BTC, of which 750,000 BTC were customer owned bitcoins that were managed by *Mt.Gox*. (At the time of the first press release, bitcoins were trading at 827 US Dollars per bitcoin, resulting in a total value of lost bitcoins of 620 million US Dollars) ([1]). We briefly describe the timeline of this incident by reconstructing it from the press releases by *Mt.Gox*.

– February 7th 2014: *Mt.Gox* announces that it would suspend all Bitcoin withdrawals, due to the fact that the associated transactions could not be confirmed.

Mt.Gox made it still possible for users to continue trading their funds, by transferring them to some sort of *virtual accounts* owned by *Mt.Gox*. [11]

- February 10th 2014: *Mt.Gox* claims in a second press release that the problem with the withdrawal transactions has been identified and names transaction malleability as the "sole" cause. [12]
- February 17th 2014 - February 20th 2014: *Mt.Gox* announces in two press releases that the withdrawals would resume shortly and that a "solution had been found".
- February 23rd 2014: The website of *Mt.Gox* returned only a blank page, without any further explanation, resulting in a trading halt and the complete disappearance of *Mt.Gox*.
- February 28th 2014 : *Mt.Gox* announces during a press conference that it would be filing for bankruptcy in Japan and in the USA. [13]

Let us illustrate how the malleability attack in this case happens. Following example 1.8, Alice deposits 1 BTC into an account on an exchange. Later, Alice tries to withdraw her 1 BTC off the exchange, back to her private wallet. If Alice controls nodes that peer with the exchange, she might be able to change the *Txid* for her withdrawal using transaction malleability. The 1 BTC she withdrew will go into her private wallet under a new *Txid*. If the exchange is naive, Alice might be able to trick it into thinking that it never sent her the said withdrawal and requests to withdraw her 1 BTC again. By doing this repeatedly, Alice could potentially withdraw a large amount of Bitcoin before the exchange realises.

3 Measurement and analysis

Malleability attacks, like every double spending attack, may only be detected while participating in the network: "As soon as one of the two conflicting transactions is considered to be confirmed, the nodes will drop all other conflicting transactions, losing all information about the said attack" ([1]). Decker and Wattenhofer created in January 2013 specialised nodes that would trace and dump all transactions and blocks from the Bitcoin network. The nodes would keep a cache of connection database up to 1,000 connections. On average, the nodes connected to 992 peers, which at that time represented approximately 20% of the reachable nodes.
Since the probability of detecting a double spending attack quickly converges to 1 as the number of sampled peers increases ([14]), one may assume that the transactions collected during the measurements faithfully reflect the double spending attacks in the network during the same period.

Definition 3.1. *Two (or more) transactions are said to* ***conflict*** *if they are associated to the same output.*

Given a set of transactions, one way to detect and group all transaction that have been subject to malleability attack would be to remove (scriptSig) from the transactions.

Definition 3.2. – *a **conflict set** is the set of transactions that do not contain scriptSig anymore and produce the same key.*

– *a **conflict set value** is the amount of bitcoins transferred by any transaction in the conflict set.*

Two transactions in a conflict set have the same output, and thus transfer the same amount of bitcoin. This makes it possible to obtain the amount of all bitcoins involved in malleability attacks by summing up the value of all conflict sets.

Recall that a malleability attack is possible if:

(a) The modified transaction is confirmed and is successfully added to the Blockchain.

(b) The system issuing the transaction rely solely on the *Txid* to track and verify its confirmation.

Since (b) depends on the implementation of the issuing system, one only considers that a malleability attack is successful if condition (a) holds.

Finally, assume that the conflict sets were a direct result of a targeted attack by an attacker against a service. (In fact, there might be other causes for this kind of conflict due to faulty parameters.

3.1 Measurement

All the data provided (numbers, graphics and ratios) are taken from ([1]).

Confirmed attacks: $29,139$ conflict sets contained a transaction that would later be confirmed by a bloc, out of $35,202$ identified conflict set. The remaining $6,063$ transactions were either invalid because they claimed non-existing outputs, had incorrect signatures, or they were part of a further double spending.

Bitcoin value: $302,700$ bitcoins were involved in malleability attacks, obtained by summing the value of the collected conflict sets.

Nature of the attacks: 98% of the attacks ($28,595$ out of the $29,139$) were of type **Push operations in scriptSig of non-standard size** as described in section **??** (replacing the single byte OP_0 with OP_PUSHDATA2). For the remaining 544 conflict sets, all transactions had genuine signatures with the correct OP_ codes and did not encode the same signature. We therefore assume these transactions to be the result of users signing raw transactions multiple times, e.g., for development purposes.

Success rate: 21.36% as out of the 28,595 confirmed attacks using **Push operations in scriptSig of non-standard size** only 5,670 were successful, i.e., 19.46% of modified transactions were later confirmed.

Total profit of the successful attacks: 64,564 bitcoins.

3.2 Analysis

Now assume transaction malleability has been used against *Mt.Gox*. From the reconstruction in Section 2 we identify three major time periods:

- Period 1 (January 2013 — February 7, 2014): Measurement before the closure of withdrawals from *Mt.Gox*.

Over a year of measurements until the closure of withdrawals, a total of 421 conflict sets were identified (Figure 7)

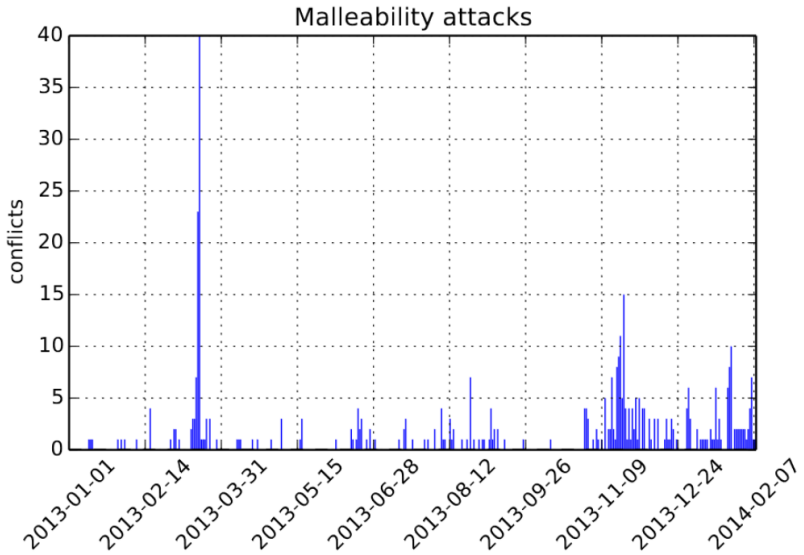


Figure 7: Malleability attacks during period 1

For a total value of 1,811.58 BTC involved in these attacks (Figure 8)

In combination with the above mentioned success rate, the estimate lost value would be of 386 BTC.

- Period 2 (February 8 - February 9, 2014): Withdrawals are suspended but without any known source of the nature of the attacks.

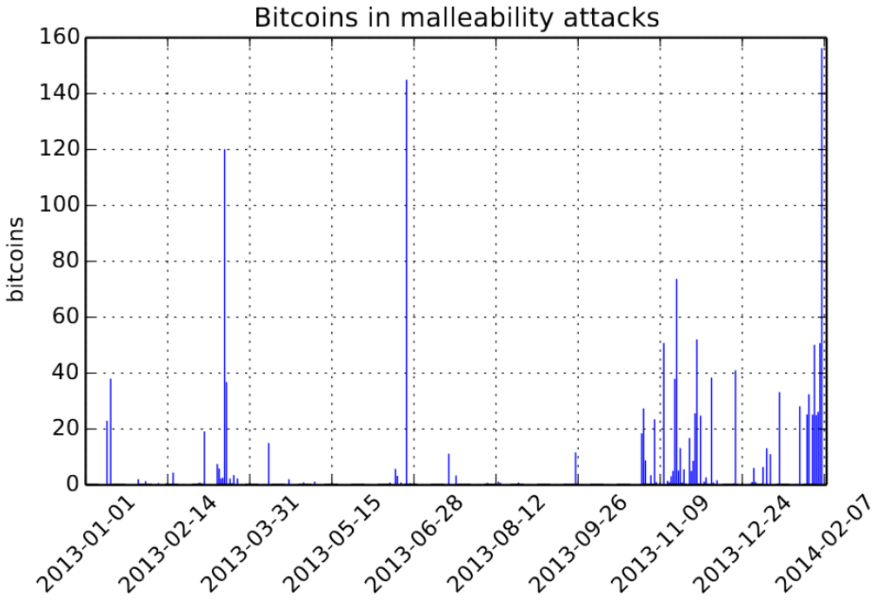


Figure 8: Bitcoins in malleability attacks during period 1

During period 2, a total of 1,062 conflict sets were identified, totalling 5,470 BTC. A noticeable increase of attacks at 17 : 00 UTC on February 9th, from 0.15 attacks per hour to 132 attacks per hour (Figure 9). No exact information about the time the second press release has been published, but one might deduce from the increase in attacks at 17:00 UTC that it should have happened between 14:00 and 17:00 UTC alias 0:00 and 2:00 JST. The sudden increase suggests that immediately after the press release other attackers started imitating the attack in order to exploit the "alleged" weakness.

- Period 3 (February 10 — February 28): Withdrawals are still suspended after revealing that the sole source of the attacks was transaction malleability.

One notices a sudden spike in activity after the second press release. Between February 10 and 11, a total of 25,752 individual attacks were identified, totalling 286,076 BTC, two orders of magnitude larger than all attacks from period 1 combined (Figure 9). A second, smaller wave of attacks starts after February 15, with a total of 9,193 BTC. The attacks have since calmed, returning to levels comparable to those observed in period 1, before the press releases (Figure 9).

As seen in the end of Section 2, by the nature of how malleability attacks work, and assuming *Mt.Gox* had disabled withdrawals like they stated in the first press release, period 2 and 3 (with a total loss estimated at 64,269.76 BTC) could not contribute

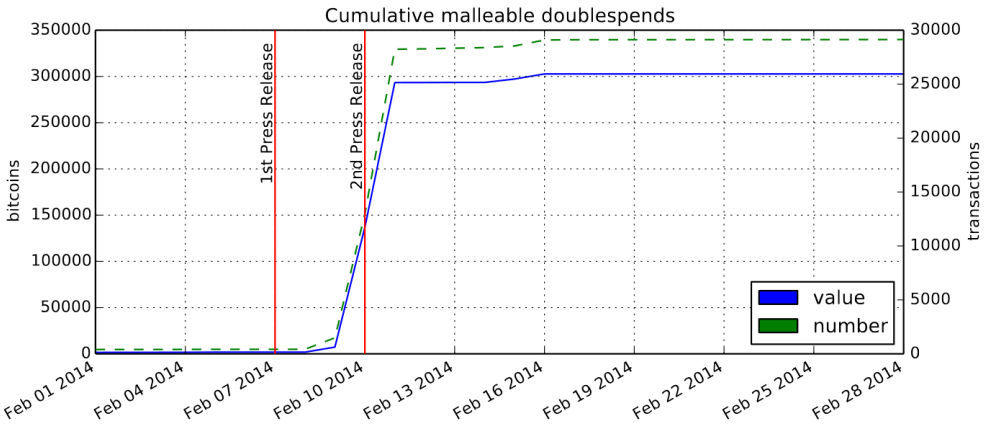


Figure 9: Bitcoins and number of malleability attacks from in period 2 and 3

to the losses declared by *Mt.Gox* since they happened **after** withdrawals have been stopped. Therefore, these attacks cannot have been aimed at *Mt.Gox*.

3.3 Discussion and conclusion

The analysis given above is based on first-hand data that were collected starting January 2013, for over a year preceding the bankruptcy filing by *Mt.Gox*. Thus one can only estimate the number of attacks preceding the measurements, just by reading the Blockchain. Since over 98% out of all attacks use the OP_PUSH code modification, it suffices to inspect all confirmed transactions for scriptSig that do not use minimal push opcodes. A total of 48 transactions were found, involving a total of 33.92 BTC. Assuming that the success rate of 21.34% did not change significantly, a total of less than 160 BTC seems to be involved involved in a few hundreds of attempted malleability attacks in the period 2009 – 2012. This is equivalent to less than 10% of the attacks identified during the measurements.

Different type of attacks might have played a role in *Mt.Gox*'s loss, but malleability attacks could have merely contributed in the declared losses. Indeed merely a total of 302,700 BTC was involved, from which only 1,811 BTC were in attacks before *Mt.Gox* stopped the withdrawels.

To conclude, transaction malleability is a serious issue and should be considered when implementing Bitcoin clients. Different type of attacks might have played a role in *Mt.Gox*'s loss, but malleability attacks could have merely contributed in the declared losses. "However, while *Mt.Gox* claimed to have lost 850,000 BTC due to malleability attacks, merely a total of 302,700 BTC was involved in malleability attacks, of which only 1,811 BTC were in attacks before *Mt.Gox* stopped users from withdrawing bitcoins. Even more, 78.64% of these attacks were ineffective, and thus

barely 387 BTC could have been stolen using malleability attacks, from *Mt.Gox* or even from other businesses."([1]). Different type of attacks might have played a role in *Mt.Gox*'s loss, but malleability attacks could have merely contributed in the declared losses.

References

[1] C. Decker, R. Wattenhofer. *Bitcoin transaction malleability and MtGox*, ESORICS 2014, pp. 313–326, Lecture Notes in Computer Science, 8713, 2014.

[2] P. Wuille. *BIP 0062: Dealing with Malleability*, https://github.com/bitcoin/bips/wiki/Comments:BIP-0062 (Online; accessed March 10, 2014)

[3] P. Wuille. *BIP 0066: Strict DER signatures*, https://github.com/bitcoin/bips/wiki/Comments:BIP-0066 (Online; created January 10, 2015)

[4] E. Lombrozo, J. Lau, P. Wuille. *Segregated Witness* (Consensus layer), https://github.com/bitcoin/bips/wiki/Comments:BIP-0141 (Online; created January 21, 2015)

[5] A.M. Antonopoulos. *Mastering Bitcoin. Programming the Open Blockchain*, Second edition, O'Reilly, 2017.

[6] D. Laptev. *Bitcoin: transactions, malleability, SegWit and scaling*, LightningTo.Me 24-08-2017

[7] E. Klitzke. *Bitcoin Transaction Malleability*, https://eklitzke.org/, 20-07-2017

[8] List of OP codes in Script *Bitcoin Wiki*, https://en.bitcoin.it/wiki/Script

[9] siminchen. *Bitcoin Script IDE*, https://github.com/siminchen/bitcoinIDE

[10] *Wiki Bitcoin*, https://en.bitcoin.it/wiki/Transaction

[11] MtGox. *Mtgox press release announcing the stop of withdrawals*, https://www.mtgox.com/press_release_20140210.html, 2014.

[12] MtGox. *Mtgox press release about transaction malleability*, https://www.mtgox.com/press_release_20140210.html, 2014.

[13] MtGox. *Announcement regarding the applicability of us bankruptcy code chapter 15*, https://www.mtgox.com/img/pdf/20140314-announcement_chapter15.pdf, 2014.

[14] T. Bamert, C. Decker, L. Elsen, S. Welten, R. Wattenhofer. *Have a snack, pay with bitcoin* InIEEE Interna-tion Conference on Peer-to-Peer Computing (P2P), Trento, Italy, 2013.