On the Malleability of Bitcoin Transactions

Marcin Andrychowicz, Stefan Dziembowski,
Daniel Malinowski, and Łukasz Mazurek$^{(\boxtimes)}$

University of Warsaw, Warsaw, Poland
{marcin.andrychowicz,stefan.dziembowski,daniel.malinowski,
lukasz.mazurek}@crypto.edu.pl

Abstract. We study the problem of malleability of Bitcoin transactions. Our first two contributions can be summarized as follows:

(i) we perform practical experiments on Bitcoin that show that it is very easy to maul Bitcoin transactions with high probability, and
(ii) we analyze the behavior of the popular Bitcoin wallets in the situation when their transactions are mauled; we conclude that most of them are to some extend not able to handle this situation correctly.

The contributions in points (i) and (ii) are experimental. We also address a more theoretical problem of protecting the Bitcoin distributed contracts against the "malleability" attacks. It is well-known that malleability can pose serious problems in some of those contracts. It concerns mostly the protocols which use a "refund" transaction to withdraw a financial deposit in case the other party interrupts the protocol. Our third contribution is as follows:

(iii) we show a general method for dealing with the transaction malleability in Bitcoin contracts. In short: this is achieved by creating a malleability-resilient "refund" transaction which does not require any modification of the Bitcoin protocol.

1 Introduction

Malleability is a term introduced in cryptography by Dolev et al. [15]. Very informally, a cryptographic primitive is *malleable* if its output C can be transformed ("mauled") to some "related" value C' by someone who does not know the cryptographic secrets that were used to produce C. For example, a symmetric encryption scheme $(\mathrm{Enc}, \mathrm{Dec})$ is malleable if the knowledge of a ciphertext $C = \mathrm{Enc}(K, M)$ suffices to compute C' such that $M' = \mathrm{Dec}(K, C')$ is not equal to M, but is related to it (e.g. M' is equal to M with the first bit set to 0). It is easy to see that the standard cryptographic security definitions (like the semantic security of encryption schemes) in general do not imply non-malleability, and hence the non-malleability is usually viewed as an additional (but often highly

This work was supported by the WELCOME/2010-4/2 grant founded within the framework of the EU Innovative Economy (National Cohesion Strategy) Operational Programme. Moreover, Łukasz Mazurek is a recipient of the Google Europe Fellowship in Security, and this research is supported in part by this Google Fellowship.

DOI: 10.1007/978-3-662-48051-9_1

desirable) feature of the cryptosystems. Since its introduction the concept of non-malleability was studied profoundly, mostly by the theory community, in several different contexts including the encryption and commitment schemes, zero-knowledge [15], multiparty computation protocols [12], hash functions and one-way functions [10], privacy amplification [14], tamper-resilient encoding [16], and many others. Until last year, however, the malleability problem remained largely out of scope of the interests of the security practitioners.

This situation has changed dramatically, when the MtGox Bitcoin exchange suspended its trading in February 2014, announcing that around 850,000 bitcoins belonging to customers were stolen by an attacker exploiting the "malleability of the Bitcoin transactions" [18]. Although there is a good evidence that MtGox used the malleability only as an excuse [13], this announcement definitely raised the awareness of the Bitcoin community of this problem, and in particular, as argued in [13] it massively increased the attempts to exploit this weakness for malicious purposes.

The fact that the Bitcoin transactions are malleable has been known much before the MtGox collapse [21]. Briefly speaking, "malleability" in this case means that, given a transaction T, that transfers x bitcoins from an address A to address B (say), it is possible to construct another transaction T' that is syntactically different from T, but semantically it is identical (i.e. T' also transfers x bitcoins from A to B)[1]. This can be done by anybody, and in particular by an adversary who does not know A's private key. On a high level, the source of the malleability comes from the fact that in the current version of the Bitcoin protocol, each transaction is identified by a hash on its *whole* contents, and hence in some cases such a T' will be considered to be a different transaction than T.

There are actually several ways T' can be produced from T. One can, e.g. exploit the malleability of the signature schemes used in Bitcoin, i.e., the fact that given a signature σ (computed on some message M with a secret key sk) it is easy to compute another valid signature σ' on M (with respect to the same key sk)[2]. Since the standard Bitcoin transactions have a form $T =$ (message M, signature σ on M), thus $T' = (M, \sigma')$ is a valid transaction with the same semantics as T, but syntactically different from T. Another method is based on the fact that Bitcoin permits more complicated transactions than those in the format described above. More precisely, in the so-called "non-standard transactions" the "σ" part is in fact a script in the stack based *Bitcoin scripting language*. Therefore, e.g., adding dummy PUSH and POP instructions to σ produces σ' that is operationally equivalent to σ, yet, from the syntactic point of view it is different. See, e.g., [13,22] for more detailed list of different ways in which the Bitcoin transactions can be mauled.

---

1 For a short description of Bitcoin and the non-standard transactions see Sect. 2.

2 This is because Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) that has the property that for every signature $\sigma = (r, s) \in \{1, \ldots, N-1\}^2$ the value $\sigma' = (r, N - s)$ is also a valid signature on the same message and with respect to the same key as σ.

Recall that in order to place a transaction T on the block chain a user simply broadcasts T over the network. Thus it is easy for an adversary $\mathcal{A}$ to learn T before it is included in the block chain. Hence he can produce a semantically equivalent T' and broadcast T'. If $\mathcal{A}$ is lucky then the miners will include T' into the block chain, instead of T. At the first sight this does not look like a serious problem: since T' is equivalent to T, thus the financial effect of T' will be identical to the effect of T. The reason why malleability may cause problems is that typically in Bitcoin the transactions are identified by their hashes. More precisely (cf., e.g., [8]), an *identifier* (TXID) of every transaction $T = (M, \sigma)$ is defined to be equal to $H(M, \sigma)$, where H is the doubled SHA256 hash function. Hence obviously TXID of T will be different than TXID of T'.

There are essentially three scenarios when this can be a problem. The first one comes from the fact that apparently some software operating on Bitcoin does not take into account the malleability of the transactions. The alleged MtGox bug falls in this category. More concretely, the attack that MtGox claimed to be the victim of looked as follows: (1) a malicious user P deposits x coins on his MtGox account, (2) the client P asks MtGox to transfer his coins back to him, (3) MtGox issues a transaction T transferring x coins to P, (4) the user P launches the malleability attack, obtaining T' that is equivalent to T but has a different TXID (assume that T' gets included into the block chain instead of T), (5) the user complains to MtGox that the transaction was not performed, (6) MtGox checks that there is no transaction with the TXID $H(T)$ and concludes that the user is right, hence MtGox credits the money back to the user's account. Hence effectively P is able to withdraw his coins twice. The whole problem is that, of course, in Step (6) MtGox should have searched not for the transaction with TXID $H(T)$, but for any transaction semantically equivalent to T.

The second scenario is related to the first one in the sense that it should not cause problems if the users are aware that malleability attacks can happen. It is connected to the way in which the procedure of "giving change" is implemented in Bitcoin. Suppose a user A has 3 Ƀ as an unspent output of the transaction T_0 on the block chain, and he wants to transfer 1 Ƀ to some other user B. Typically, what A would do in this case is: create a transaction T_1 that has input T_0 and has two outputs: one that can be claimed by B and has value 1 Ƀ, and the one that can be claimed by himself (i.e. A) and has value 2 Ƀ. He would then post T_1 on the block chain. If he now creates a further transaction T_2 that claims money from T_1 *without* waiting for T_1 to appear on the block chain, then he risks that T_2 will be invalid, in case someone mauls T_1.

The third scenario is much more subtle, as it involves the so-called *Bitcoin contracts* which are protocols to form financial agreements between mutually distrusting Bitcoin users. In this paper we assume readers familiarity with this Bitcoin feature. For an introduction to it see, e.g., [3,4,19] (in this paper we use the notation from [3,4]). Recall that contracts are more complicated than normal money transfers in the standard transactions. To accomplish their goal contracts use the non-standard transactions. One of the common techniques used in constructing such contracts is to let the users sign a transaction T_1 before its input T_0 is included in the block chain. Very informally speaking, the problem is that T_1 is constructed using the TXID $H(T_0)$, which means that an adversary that mauls T_0 into some (equivalent but syntactically different) T_0' can make the transaction T_1 invalid. There are many examples of such situations. One of them is the "Providing a deposit" contract from [19], which we describe in more detail in Sect. 4, where we also explain this attack in more detail. Note that, unlike in the first two scenarios, this problem cannot be mitigated by patching the software.

1.1 Possible Fixes to the Bitcoin Malleability Problem

There are several ways in which one can try to fix the problems caused by the malleability of Bitcoin transactions. For example one can try to modify Bitcoin in order to eliminate malleability from it. Such proposals have indeed been recently put forward. In particular, Pieter Wuille [22] proposed a number of ad-hoc methods to mitigate this problem, by restricting the syntax of Bitcoin transactions. While this interesting proposal may work in practice, it is heuristic and it comes with no formal argument. In particular, it implicitly assumes that the only way to maul the ECDSA signatures is the one described in Footnote 2, and we are not aware of any formal proof that this is indeed the case.

In our previous paper [3] we proposed another modification of Bitcoin which eliminates the malleability problem. The idea of this modification is to identify the transactions by the hashes of their *simplified* versions (excluding the input scripts). With this modification one can of course still modify the input script of the transaction, but the modified transaction would have the same hash. Unlike [22] this solution does not rely on heuristic properties of the signature schemes. On the other hand, the proposal of [22] may be easier to implement in practice, since it requires milder modifications of the Bitcoin specification.

Another solution proposed recently by Peter Todd [1] is to introduce a new instruction `OP_CHECKLOCKTIMEVERIFY` to the Bitcoin scripting language that allows a transaction output to be made unspendable until some point in the future. It does not concern the problem of malleability directly, but using this opcode would allow to easily create Bitcoin contracts resilient to malleability.

Unfortunately, changing the Bitcoin is in general hard, since it is not controlled by any central authority, and hence the modifications done without proper care can result in a catastrophic fork, i.e. a situation where there is a disagreement among the honest parties about the status of transactions[3]. Thus, it is not clear if such modifications will be implemented in the close future. It is therefore natural to ask what can be done, assuming that the Bitcoin system remains malleable.

First of all, fortunately, as described above in many cases malleability is not a problem if the software is written correctly, and therefore the most obvious thing to do is the following.

[3] An example of such a fork was experienced by the Bitcoin community in March 2013, when it was caused by a bug in a popular mining client software update [11]. Fortunately it was resolved manually, but it is still remembered as one of the moments when Bitcoin was close to collapse.

Direction 1: Educate the Bitcoin software developers about this issue. Convince them that it is a real threat and they should always test their software against such attacks.

The only context in which the malleability cannot be dealt with by better programming are the Bitcoin contracts. Hence a natural research objective is as follows.

Direction 2: Develop a technique that helps to deal with the malleability of Bitcoin transactions in the Bitcoin contracts.

The goal of this paper is to contribute to both of these tasks.

1.2 Our Contribution

The technical contents of this paper is divided into two parts corresponding to the research directions described above. We first focus on "Direction 1" (this is done in Sect. 3). Since most of the software practitioners will probably only care about problems where the threat is real, not theoretic, we executed practical experiments that examine the feasibility of the malleability attacks. It turns out that these attacks are quite easy to perform, even by an attacker that has resources comparable to those of an average user. In fact, our experiments show that it is relatively easy to achieve success rates close to 50 %.

We then analyze the behavior of popular Bitcoin clients when they are exposed to such attacks. Our results indicate that all of them show a certain resilience to such attacks. In particular we did not identify weaknesses that would allow users to steal money (as argued in Sect. 1.3 this is in fact something that one would expect from the beginning). On the other hand, we identified a number of smaller weaknesses in most of these clients. In particular, we observed that in many cases the malleability attack results in making the user unable to issue further transactions, unless he "resets" the client. In most cases such a reset can be performed relatively easy, in one case it required an intervention of a technically-educated user (restoring the backup files), and in two cases there seemed to be no way to perform such action.

This shows that some of the Bitcoin developers seem to still ignore the malleability problem, despite of the fact that over 8 months have passed since the infamous MtGox statement.

The second part (contained in Sect. 5) of this paper concerns the "Direction 2". We provide a general technique for transforming a Bitcoin contract that is vulnerable to the malleability attacks (but secure otherwise), into a Bitcoin contract that is secure against such attacks. Our method covers all known to us cases of such contracts, in particular, those listed on the "Contracts" page of the Bitcoin Wiki [19], and the lottery protocol of [5]. It can also be applied to [3], what gives the first fair Two-Party Computation Protocol (with financial consequences) for any functionality whose fairness is guaranteed by the Bitcoin deposits, and which, unlike the original protocol of [3] can be used on the current version of Bitcoin[4].

[4] The protocol of [3] was secure only under the assumption that the Bitcoin is modified to prevent the malleability attacks.

Related Work. Some of the related work was already described in the previous sections. The idea of using Bitcoin to guarantee fairness in the multiparty protocols and to force the users to respect the outcome of the computation was proposed independently in [3,4] and in [6] (and implicitly in [5]), and was also studied in [7]. The protocols of [4] and [5] work only for specific functionalities (i.e. are not generic), and [5] is vulnerable to the malleability attack. The protocols of [3,6] are generic, but are insecure against the malleability attack. Also the protocol of [7] seems to be insecure against such attacks.

1.3 Ethical Issues

We realize that performing the malleability attacks against the Bitcoin can raise questions about the ethical aspects of our work. We would like to stress that we were only attacking transactions that were issued by ourselves (and we never tried to maul transactions coming from third parties). It is also clear that performing such attacks cannot be a threat to stability of the whole Bitcoin system, since, as reported by [13] Bitcoin remained secure even against attacks on a much higher scale ([13] registered 25,752 individual malleability attacks involving 286,076 bitcoins just on two days of February 2014).

Let us also note that even before we started our work we could safely assume that none of the popular Bitcoin clients is vulnerable to the malleability attacks to the extent that would allow malicious users to steal money, as it is practically certain that any such weakness would be immediately exploited by malicious users. In fact, as argued in [13] such malicious attempts were probably behind the large number of malleability attacks immediately after the MtGox collapse. In other words: the experiments that we performed were almost certainly performed by several hackers before us. We believe that therefore making these results public is in the interest of the whole Bitcoin community.

2 Bitcoin Description

We assume reader's familiarity with the basic principles of Bitcoin. For general description of Bitcoin, see e.g. [4,17,20]. For the description of non-standard transaction scripts, see [3,4,19]. Let us only briefly recall that the Bitcoin currency system consists of *addresses* and *transactions* between them. An address is simply a public key pk (technically an address is a *hash* of pk). We will frequently denote key pairs using the capital letters (e.g. A). We will also use the following convention: if $A = (sk, pk)$ then $\mathsf{sig}_A(m)$ denotes a signature on a message m computed with sk and $\mathsf{ver}_A(m, \sigma)$ denotes the result (true or false) of the verification of a signature σ on message m with respect to the public key pk.

Each Bitcoin transaction can have multiple inputs and outputs. Inputs of a transaction T_x are listed as triples $(y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n)$, where each y_i is a hash of some previous transaction T_{y_i}, a_i is an index of the output of T_{y_i} (we say that T_x *redeems the* a_i*-th output of* T_{y_i}) and σ_i is called an *input-script*. The outputs of a transaction are presented as a list of pairs $(v_1, \pi_1), \ldots, (v_m, \pi_m)$, where each v_i specifies some amount of coins (called the *value of the i-th output of* T_x) and π_i is an *output-script*. A transaction can also have a time-lock t, meaning that it is valid only if time t is reached. Hence, altogether transaction's most general form is: $T_x = ((y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$. The *body of* T_x[5] is equal to T_x without the input-scripts, i.e.: $((y_1, a_1), \ldots, (y_n, a_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$, and denoted by $[T_x]$.

One of the most useful properties of Bitcoin is that the users have flexibility in defining the condition on how the transaction T_x can be redeemed. This is achieved by the input- and the output-scripts. One can think of an output-script as a description of a function whose output is Boolean. A transaction T_x defined above is valid if for *every* $i = 1, \ldots, n$ we have that $\pi'_i([T_x], \sigma_i)$[6] evaluates to true, where π'_i is the output-script corresponding to the a_i-th output of T_{y_i}. Another conditions that need to be satisfied are that the time t has already passed, $v_1 + \cdots + v_m \leq v'_1 + \cdots + v'_n$ where each v'_i is the value of the a_i-th output of T_{y_i} and each of these outputs has not been already spent. The scripts are written in the Bitcoin scripting language. Following [4] we will present the transactions as boxes. The redeeming of transactions will be indicated with arrows (cf. e.g. Fig. 3). The transactions where the input script is a signature, and the output script is a verification algorithm are the most common type of transactions and are called *standard transactions*. The address against which the verification is done will be called a *recipient* of this transaction.

We use the security model defined in [4]. In particular, we assume that each party can access the current contents of the block chain, and post messages on it. Let Δ be the is maximal possible delay between broadcasting the transaction and including it in the block chain.

3 Experiments

The Implementation of the Malleability Attack. In order to perform malleability attacks we have implemented a special program called adversary (using the *bitcoinj* [2] library). This program is connected as a peer to the Bitcoin network and listens for transactions sending bitcoins to a particular address *Addr*[7] owned by us. Whenever such a transaction is received by the program, it mauls the transaction by changing (r, s) into $(r, N - s)$ in the ECDSA signature (cf. footnote 2 on Page 2) and broadcasts the modified version of the transaction.

The effectiveness of the attack is measured by the percent of cases in which the mauled transaction becomes confirmed and the original one invalidated. It depends on the fraction of the network (and hence miners), which receives the mauled transaction before the original one. In order to achieve a high effectiveness we need to push the modified transaction to the whole peer-to-peer network

[5] In the original Bitcoin documentation this is called "simplified T_x".

[6] Technically in Bitcoin $[T_x]$ is not directly passed as an argument to π'_i. We adopt this convention to make the exposition clearer.

[7] In our experiments we used addresses `13eb7BFXgHeXfxrdDev1ehrBSGVPG6obu8` and `115g32FHp77hQpuuWpw8j8RYKPvxD1AXyP`.

as fast as possible. Therefore, the adversary connects to many more peers than a typical Bitcoin client, more precisely it maintains on average 1000 connections[8]. Moreover, it connects directly to some nodes, which are known to be maintained by the mining pools[9] and sends the mauled transaction to them in the first place.

Effectiveness Analysis. In order to measure the effectiveness of our attack we set up another machine called victim, which makes hundreds of transactions sending bitcoins to the address *Addr*. More concretely we measured the effectiveness of mauling transactions under 3 different circumstances:

(A) The IP address of a victim is not known to the adversary.
(B) The IP address of a victim is known to the adversary. In this case the adversary can connect directly to the victim, what allows him to discover transactions broadcast by the victim much faster. In our experiments both machines — adversary and victim were located in the same local network, which in some cases can be possible also in real life (a motivated adversary can connect to the local network used by the victim). Our experiments showed that connecting directly to the victim greatly increases the effectiveness of the attack.
(C) The victim is aware of the malleability problem and tries to protect against it by broadcasting her transactions on a higher number of connections. In our experiment the victim was connected to 100 peers on the network and her IP address was not known to the adversary.

The results of our experiments are presented on Fig. 1. The effectiveness depends on the nodes to which the victim is connected, so after each transaction she drops all her connections and establishes new ones. Moreover, the effectiveness depends on the distribution of mined blocks among miners in the testing period, so experiments were performed over longer periods of time (e.g. 24 hours). In order to exclude the influence of the factors like physical proximity of adversary and victim (in experiments A and C) we performed part of them with victim and adversary running on machines far away from each other.

Clients Testing. In order to determine the significance of the malleability problem for individual Bitcoin users, we decided to test how the most popular Bitcoin wallets behave when a transaction is mauled. We have tested 14 Bitcoin wallets listed in [9]. In every test we performed several Bitcoin transfers from the wallet to *Addr*. During the tests the adversary was trying to maul every transaction addressed to this address.

We observed that (1) most of the clients determine whether the transaction is confirmed by looking for a transaction with a matching hash in the block chain, and (2) the clients are likely to receive from the network the modified transaction

[8] Typical Bitcoin client maintains about 8 connections.
[9] More precisely it connects to the nodes maintained by mining pools *GHash.IO* and *Eligius*.

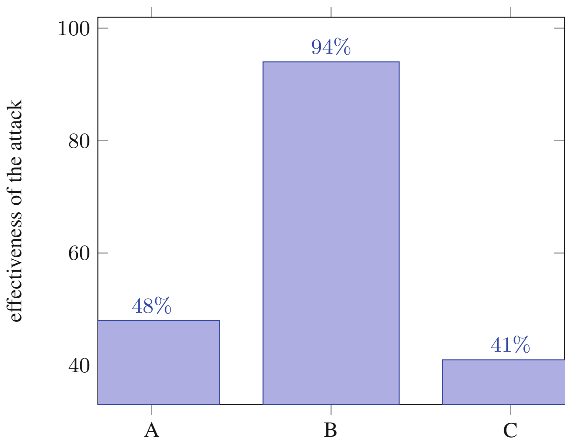


Fig. 1. Effectiveness of the attack under different circumstances.

and therefore list in the transaction history either the original transaction or the modified one or both. This can lead to unpredictable behavior of the wallet of many types, in most cases being hard to precisely describe. In Table 1 we present the results of our tests. In a nutshell, we have observed the following types of behavior of the clients:

(a) the wallet incorrectly computes the balance,
(b) the wallet is unable to make an outgoing transaction because it assumes that some transaction will be confirmed in the future (which in fact will never happen),
(c) the application crashes.

Note that if the client refuses to make an outgoing transaction and it is not possible to remove the troublesome transaction from the history, the user will be unable to spend his bitcoins forever. Moreover, because of the "giving change" procedure, an attack against a single transaction can potentially make all the money in the wallet unspendable (cf. the second scenario on Page 2).

Going more into the details: *Bitcoin Core* and *Xapo* appear to handle the malleability problem correctly. For example *Bitcoin Core* detects the malleability attack and marks it as "double spending" (indicating it with an exclamation mark). *Green Address* and *Armory* display incorrect balances (however, when it comes to allowing a user to make a transaction, they seem to take into account the real balance). *Blockchain.info*, *Coinkite*, *Coinbase*, *Electrum*, *MultiBit*, *Bitcoin Wallet*, and *KnC Wallet* may get stuck waiting for the confirmation of the transaction, which will never be confirmed and hence prevent the user from creating correctly the next transaction. Fortunately, these clients either give the user an option to "synchronize the list of transactions with block chain" or they do it automatically after some time, which makes the problem disappear. In *Hive* we were able the obtain the effect of "resetting" the transaction list only by a manual action (which may be non-trivial for a non-technically educated user).

Table 1. The results of testing 14 Bitcoin wallets listed on [9] against malleability attack. The "X" sign means that the client (a) incorrectly computes the balance, (b) refuses to make an outgoing transaction despite the available funds or (c) crashes. All the tests took place in October 2014 and hence may not correspond to the current software version. Value *never* in the last column means that we could not figure out a way to solve the problem and it did not disappear on its own after a few days.

Wallet name	Type	(a)	(b)	(c)	When the problem disappears
Bitcoin core	Desktop				-
Xapo	Web				-
Armory	Desktop	✘			never
Green address	Destop	✘			never
Blockchain.info	Web	✘	✘		after six blocks without confirmation
Coinkite	Web	✘	✘		after several blocks without confirmation
Coinbase	Web	✘	✘		after several hours
Electrum	Desktop	✘	✘		after application reset
MultiBit	Desktop	✘	✘		after "Reset block chain and transactions" procedure
Bitcoin wallet	Mobile	✘	✘		after "Reset block chain" procedure
KnC wallet	Mobile	✘	✘	✘	after "Wallet reset" procedure
Hive	Desktop	✘	✘	✘	after restoring the wallet from backup
BitGo	Web	✘	✘		never
Mycelium	Mobile	✘	✘		never

The problem is much more severe in case of *BitGo* and *Mycelium*, which also display the troublesome transaction, but there appears to be no way to reset the list. We note that, since BitGo is a web-client, the wrong transaction can be removed by the administrator.[10]

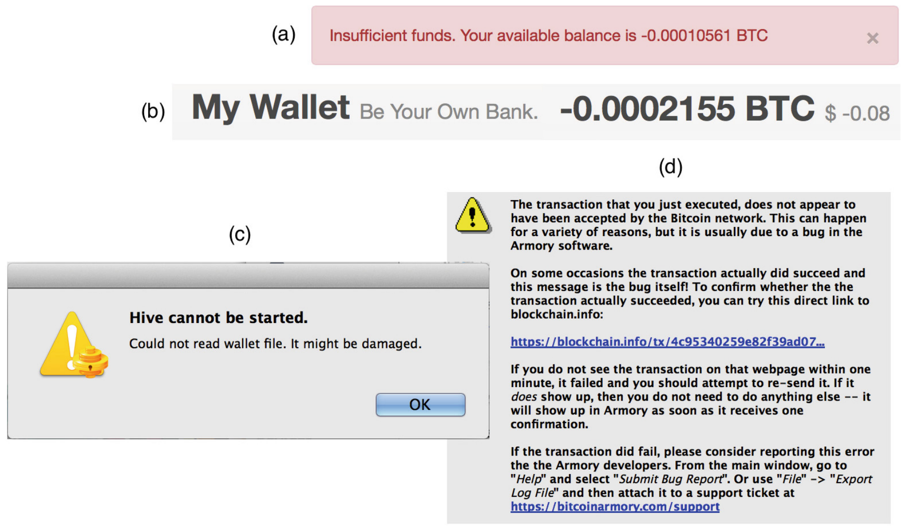
4 Malleability in Bitcoin Contracts

As shown in the earlier sections malleability of Bitcoin transactions can pose a problem to users if Bitcoin clients or services they are using have bugs in their implementation. But when these bugs are fixed then there should be no problems or dangers in typical usage of Bitcoin. Unfortunately malleability is a bigger problem for the security of the Bitcoin contracts. In this section we will describe a (known) malleability attack on a protocol for a deposit. Later we will also identify other protocols that are vulnerable to the malleability attack (Fig. 2).

4.1 The Deposit Protocol

The Deposit protocol [19] is executed between parties A and B. To remain consistent with the rest of this paper we describe it here using the notation from [3,4] (cf. Sect. 2). The idea of this protocol is to allow A to create a financial

[10] In fact, the BitGo administrators reacted to our experiments by contacting us directly.



Insufficient funds. Your available balance is -0.00010561 BTC

My Wallet Be Your Own Bank. -0.0002155 BTC $ -0.08

Hive cannot be started.

Could not read wallet file. It might be damaged.

The transaction that you just executed, does not appear to have been accepted by the Bitcoin network. This can happen for a variety of reasons, but it is usually due to a bug in the Armory software.

On some occasions the transaction actually did succeed and this message is the bug itself! To confirm whether the the transaction actually succeeded, you can try this direct link to blockchain.info:

https://blockchain.info/tx/4c95340259e82f39ad07...

If you do not see the transaction on that webpage within one minute, it failed and you should attempt to re-send it. If it *does* show up, then you do not need to do anything else -- it will show up in Armory as soon as it receives one confirmation.

If the transaction did fail, please consider reporting this error the the Armory developers. From the main window, go to "*Help*" and select "*Submit Bug Report*". Or use "*File*" -> "*Export Log File*" and then attach it to a support ticket at https://bitcoinarmory.com/support

Fig. 2. Different behavior of clients during malleability attacks: (a) BitGo, (b) Blockchain.info., (c) Hive, and (d) Armory.

deposit worth d Ƀ for some period of time. A has to be sure that after time t she will get her money back and B has to be sure that A will not be able to claim her money earlier. One of the possible applications of this protocol is the scenario when B is a server and A is a user that wants to open an account on the server B. In this case B wants to be sure that A is a human and not a bot that creates the accounts automatically. To assure that, B forces A to create a small deposit for some time. For an honest user this should not be a big problem, because the deposit is small and she will get this money back. On the other hand it makes it expensive to create many fake accounts (for some malicious purposes), because the cumulative value of the deposits would grow huge.

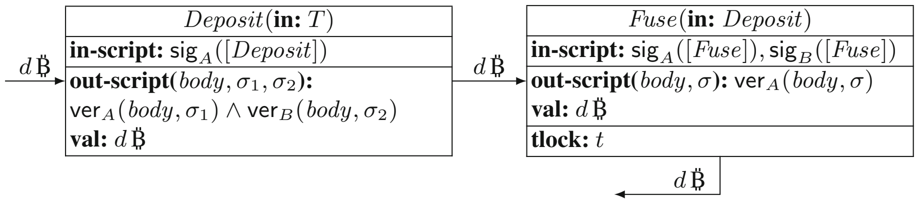
We will now describe the deposit protocol in an informal way. The main idea of this protocol is fairly simple: A "deposits" her money using a transaction *Deposit* that can be spent only using the signatures of both A and B. To be sure that this money will go back to her she creates a transaction *Fuse* that spends *Deposit*. This transaction needs to be signed by B, and hence A asks B to sign it, and only after A receives this signature she posts *Deposit* on the block chain. In order to prevent $\mathcal{A}$ from claiming her money too early *Fuse* contains a timelock t. In more detail the protocol looks as follows:

1. At the beginning A and B exchange their public keys used for signing Bitcoin transactions and they agree on the deposit size d and time t at which the deposit will be freed.
2. Then A creates a transaction *Deposit* of value d Ƀ, but she does not broadcast it. This transaction is constructed in such a way that to spend it someone has to include both signatures of A and B.

3. Afterwards A creates the transaction *Fuse* that has a time lock t, spends the transaction *Deposit* and sends the money back to her. This transaction is also not yet broadcast.
4. Then A sends the transaction *Fuse* to B, he signs it and sends back to A.
5. Only then A sends the transaction *Deposit* to the block chain.
6. After time t she sends the transaction *Fuse* to get the deposit back.

The graph of transactions in the Simple deposit protocol is presented on Fig. 3. The security properties that one would expect from this protocol are as follows:

(a) A is not able to get her deposit back before the time t (assuming that B follows the protocol).
(b) A will not lose her deposit, i.e. she will get d ฿ back before the time $(t + \varDelta)$ (where $\varDelta$ denotes the maximum latency between broadcasting transaction and its confirmation).



Deposit(**in:** T)
in-script: $\mathsf{sig}_A([\mathit{Deposit}])$
out-script$(\mathit{body}, \sigma_1, \sigma_2)$**:**
$\mathsf{ver}_A(\mathit{body}, \sigma_1) \wedge \mathsf{ver}_B(\mathit{body}, \sigma_2)$
val: d ฿
d ฿
d ฿
Fuse(**in:** *Deposit*)
in-script: $\mathsf{sig}_A([\mathit{Fuse}]), \mathsf{sig}_B([\mathit{Fuse}])$
out-script(body, σ)**:** $\mathsf{ver}_A(\mathit{body}, \sigma)$
val: d ฿
tlock: t
d ฿

Fig. 3. The Deposit protocol (vulnerable to malleability) from [19].

It is easy to see that (a) holds, since there is no way *Deposit* can be spent before time t (as it requires B's signature to be spent). One would be tempted to say that also (b) holds, since A can always claim her money back by posting *Fuse* on the block chain. Unfortunately, it turns out that this argument strongly relies on the fact that *Deposit* was posted on the block chain *exactly* in the same version as the one that was used to create the *Fuse* transaction. Hence, if the adversary mauls *Deposit* and posts some *Deposit*$'$ (syntactically different, but semantically equivalent to *Deposit*) then the *Fuse* transaction will not be able to spend it (as it expects its input to have TXID equal to $H(\mathit{Deposit})$, not $H(\mathit{Deposit}'))$.

4.2 Other Protocols Vulnerable to the Malleability Attack

In this section we will list other known Bitcoin contracts that are vulnerable to the malleability attack. The problem with all of them is that they are creating a transaction spending another transaction before the latter is included in the block chain. Each of this protocols can be made resistant to malleability using our technique described in the next section.

– "Example 5: Trading across chains" from [19][11]
– "Example 7: Rapidly-adjusted (micro)payments to a pre-determined party" from [19][12]
– Back and Bentov's lottery protocol [5][13]
– *Simultaneous Bitcoin-based timed commitment scheme* protocol from [3][14]

5 Our Technique

In this section we will show how to fix the Deposit protocol to make it resistant to malleability. This technique can be used also to other protocols, e.g. those listed in Sect. 4.2. Recall that the reason why the protocols from Sects. 4.1 and 4.2 were vulnerable to the malleability attacks was that one party, say A, had to obtain a signature of the other party (B) on a transaction T_1, whose input was a transaction T_0, and this had to happen *before* T_0 was posted on the block chain (in case of the Deposit protocol T_0 and T_1 were the *Deposit* and the *Fuse* transactions, resp.). Our main idea is based on the observation that, using the properties of the Bitcoin scripting language, we can modify this step by making T_0 spendable not by using the B's signature, but by providing a preimage s of some value h under a hash function H (where H can be, e.g., the SHA256 hash function available in the Bitcoin scripting language)[15]. In other words, the T_0's spending condition

$$\textbf{out-script}(\mathit{body}, \ldots, \sigma)\colon\ \cdots \wedge \mathsf{ver}_B(\mathit{body}, \sigma)$$

(cf. the Deposit protocol in Fig. 3) would be replaced by

$$\textbf{out-script}(\mathit{body}, \ldots, x)\colon\ \cdots \wedge H(x) = h,$$

[11] The malleability problem occurs in Step 3 when Party A generates Tx2 (the contract) which spends Tx1, and then asks B to sign it and send it back. This happens before Tx1 is included in the block chain and hence if later the attacker succeeds in posting a mauled version Tx1$'$ of Tx1 on the block chain, then the transaction Tx2 becomes invalid.

[12] The malleability problem is visible in Step 3, where the refund transaction T2 is created. This transaction depends on the transaction T1 that is not included in the block chain at the time when both parties sign it (in Steps 3 and 4).

[13] The problem occurs in Steps 4 and 7, where the "refund_bet" and "refund_reveal" transactions are signed before theirs input transactions "bet" and "reveal" are broadcast.

[14] The problem is visible in Step 2 of the *Commit* phase of this protocol (the $\mathit{Fuse}^{\mathsf{A}}$ and $\mathit{Fuse}^{\mathsf{B}}$ transactions are created before their input transaction *Commit* appears on the block chain).

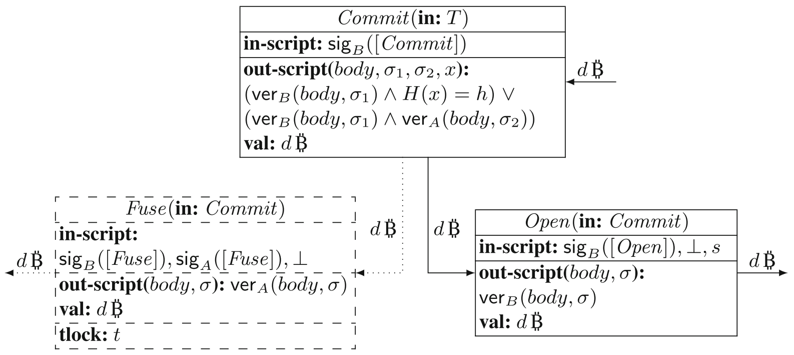
[15] Such transactions are called *hash locked* in the Bitcoin literature. Notice that having an output script, which requires only preimages and no signatures is not secure, because anyone who notices in the network a transaction trying to redeem such output script learns the preimages and can try to redeem this output script on his own. In our case the output script of the transaction T_0 requires also a signature of A, but we omit it (...) to simplify the exposition.

where $h = H(s)$ is communicated by B to A, and s is chosen by B at random. This would allow A to spend T_0 no matter how it is mauled by the adversary, provided that A learns s. At the first sight this solution makes little sense, since there is no way in which B can be forced to send s to A (obviously in every protocol considered above s would need to be sent to A some time *after* T_0 appears on the block chain, as otherwise a malicious A could spend T_0 immediately). Fortunately, it turns out that this problem can be fixed by adding one more element to the protocol. Namely, we can use the *Bitcoin-based timed commitment scheme* from [4] which is a protocol that does exactly what we need here: it allows one party, called the *Committer* (in our case: B) to *commit* to a string s by sending $h = H(s)$ to the *Recipient* (here: A). Later, B can *open* the commitment by sending s to A (before this happens s is secret to $\mathcal{A}$). The special property of this commitment scheme is that the users can specify a time t until which B has to open the commitment. If he does not do it by this time, then he is forced to pay a fine (in bitcoins) to A. As shown in [4], the Bitcoin-based timed commitment scheme is secure even against the malleability attacks. For completeness we present this protocol in more detail in the next section.

5.1 Bitcoin-Based Timed Commitment Scheme

The Bitcoin-based timed commitment scheme protocol (CS) is being executed between the Committer B and the Recipient A. During the commitment phase the Committer commits himself to some string s by revealing its hash $h = H(s)$. Moreover the parties agree on a moment of time t until which the Committer should open the commitment, i.e. reveal the secret value s. The protocol is constructed in such a way that if the Committer does not open the commitment until time t, then the agreed amount of d ฿ is transferred from the Committer to the Recipient. More precisely, at the beginning of the protocol the Committer makes a deposit of d ฿, which is returned to him if he opens the commitment before time t or taken by the Recipient otherwise.

The graph of transactions and the full description of the CS protocol is presented on Fig. 4. The main idea is that if the Committer is honest then only the transactions *Commit* and *Open* will be used (to commit to s and to open s respectively). If, however, the Committer refuses to open his commitment, then the Recipient will post the *Fuse* transaction on the block chain and claim B's deposit. Observe that *Fuse* is time-locked and therefore a malicious recipient cannot claim the money before the time t (and after time t he can do it only if B did not open the commitment). The reader may refer to [4] for more details. Note that even if the transaction *Commit* is maliciously changed before being included in the block chain, the protocol still succeeds because the transaction *Fuse* is created after *Commit* is included in the block chain, so it always contains the correct hash of *Commit*. Therefore, the CS protocol is resistant to the transaction malleability. The properties of the CS protocol are as follows:


$Commit(\textbf{in: } T)$
in-script: $\mathsf{sig}_B([Commit])$
out-script$(body, \sigma_1, \sigma_2, x)$**:**
$(\mathsf{ver}_B(body, \sigma_1) \wedge H(x) = h) \vee$
$(\mathsf{ver}_B(body, \sigma_1) \wedge \mathsf{ver}_A(body, \sigma_2))$
val: d ฿
d ฿
$Fuse(\textbf{in: } Commit)$
in-script:
$\mathsf{sig}_B([Fuse]), \mathsf{sig}_A([Fuse]), \perp$
out-script$(body, \sigma)$**:** $\mathsf{ver}_A(body, \sigma)$
val: d ฿
tlock: t
d ฿
d ฿
d ฿
$Open(\textbf{in: } Commit)$
in-script: $\mathsf{sig}_B([Open]), \perp, s$
out-script$(body, \sigma)$**:**
$\mathsf{ver}_B(body, \sigma)$
val: d ฿
d ฿

Pre-conditions:

1. The protocol is being executed between the Committer B holding the key pair B and the Recipient A holding the key pair A.
2. The Committer knows the secret string s.
3. The block chain contains an unredeemed transaction T with value d ฿, which can be redeemed with the key B.

The $\mathsf{CS.Commit}(\mathsf{B}, \mathsf{A}, d, t, s)$ **phase:**

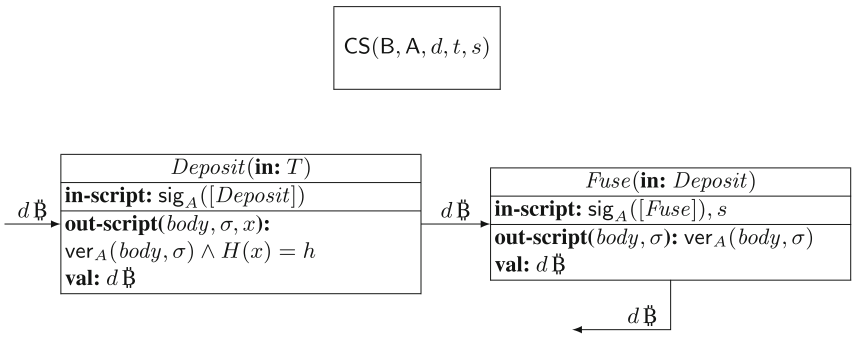
1. The Committer computes $h = H(s)$ and broadcasts the transaction *Commit*. This obviously means that he reveals h, as it is a part of the transaction *Commit*.
2. The Committer waits until the transaction *Commit* is confirmed. Then, he creates a body of the transaction *Fuse*, signs it and sends the signature to the Recipient.
3. If the Recipient does not receive the signature or the signature or transaction *Commit* are incorrect then he quits the protocol.

The $\mathsf{CS.Open}(\mathsf{B}, \mathsf{A}, d, t, s)$ **phase:**

4. Not later than at the time $(t - \varDelta)$ the Committer broadcasts the transaction *Open*, what reveals the secret s.
5. If within time t the transaction *Open* does not appear on the block chain then the Recipient broadcasts the transaction *Fuse* to gain d ฿ (or learn the secret s if the Committer sends the transaction *Open* in the meantime and it is included in the block chain).

Fig. 4. The CS protocol from [4]. The scripts' arguments, which are omitted are denoted by $\perp$.

(a) The Recipient has no information about the secret s before the Committer broadcasts the transaction *Fuse* (this property is called *hiding*).
(b) The Committer cannot open his commitment in a different way than revealing his secret s (this property is called *binding*).
(c) The honest Committer will never lose his deposit, i.e. he will receive it back not later than at the time t.
(d) If the Committer does not reveal his secret s before the time $(t + \varDelta)$ then the Recipient will receive d ฿ of compensation.


$\mathsf{CS}(\mathsf{B}, \mathsf{A}, d, t, s)$
d B
$Deposit$(**in:** T)
in-script: $\mathsf{sig}_A([Deposit])$
out-script($body, \sigma, x$): $\mathsf{ver}_A(body, \sigma) \wedge H(x) = h$
val: d B
d B
$Fuse$(**in:** $Deposit$)
in-script: $\mathsf{sig}_A([Fuse]), s$
out-script($body, \sigma$): $\mathsf{ver}_A(body, \sigma)$
val: d B
d B

Pre-conditions:

1. A holds the key pair A and B holds the key pair B.

The commitment phase:

1. B samples a random secret s.
2. The parties execute the $\mathsf{CS.Commit}(\mathsf{B}, \mathsf{A}, d, t, s)$ protocol.
3. A creates the *Deposit* transaction using the value h from the transaction $\mathsf{CS}(\mathsf{B}, \mathsf{A}, d, t, s).Commit$ and broadcasts it.

The opening phase:

4. B opens the commitment revealing s at the time $(t - \Delta)$.
5. A learns the value of s and broadcasts the *Fuse* transaction.
6. If B has not opened the commitment until time t then A broadcasts the transaction $\mathsf{CS}(\mathsf{B}, \mathsf{A}, d, t, s).Fuse$ to gain d B (or learn the value s if the Committer sends the transaction *Open* in the meantime and it is included in the block chain; in that case A may just broadcast the *Fuse* transaction to get her deposit back).

Fig. 5. The solution of the deposit problem resistant to malleability. $\mathsf{CS}(\mathsf{B}, \mathsf{A}, d, t, r)$ denotes the transactions in the appropriate execution of the CS protocol.

5.2 The Details of Our Method

We now present in more detail our solution of the malleability problem in Bitcoin contracts that was already sketched at the beginning of Sect. 5. It can be used in all of the Bitcoin contracts that are vulnerable to the malleability attacks that we are aware of. In this paper we show how to apply it to the Deposit protocol (described in Sect. 4.1).

The main idea of our solution is to use the CS protocol instead of standard *Fuse* transaction. More precisely at the beginning of the protocol B samples a random secret s and commits himself to it using the *Commit* phase of the CS protocol. Now A knows that B will have to reveal his secret (i.e. the value s s.t. $H(s) = h$) before the time t. So A can create a *Deposit* transaction in such a way that to spend it she has to provide her signature and the value s. That means that after the time t either B will reveal the value s and A will be able to spend the transaction *Deposit* or A will get the deposit of B from the CS protocol. Such a modified protocol is denoted NewDeposit. Its graph of transactions and its full description is presented on Fig. 5.

The properties of the NewDeposit protocol are as follows (all of them hold even against the malleability attacks):

(a) A is not able to get her deposit back before the time $(t - \Delta)$.
(b) The honest A will not lose her deposit, i.e. she will get d B back before the time $(t + 2\Delta)$.
(c) Additionally the honest B will not lose his deposit, i.e. he will get it back before the time t.

The proof of the above properties is straightforward and it is omitted because of the lack of space. We note that this protocol may not be well-suited for the practical applications, as it requires the server to make a deposit. Nevertheless, it is a very good illustration of our technique, that is generic and has several other applications.

References

1. bips/bip-0065.mediawiki. http://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki. Accessed on 10 December 2014
2. bitcoinj library homepage. http://bitcoinj.github.io. Accessed on 20 October 2014
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Fair two-party computations via bitcoin deposits. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) Financial Cryptography and Data Security. Lecture Notes in Computer Science, pp. 105–121. Springer, Berlin Heidelberg (2014)
4. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: 2014 IEEE Symposium on Security and Privacy (SP), May (2014)
5. Back, A., Bentov, I.: Note on fair coin toss via bitcoin (2013). http://www.cs.technion.ac.il/7Eidddo/cointossBitcoin.pdf
6. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014)
7. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. Cryptology ePrint Archive, Report 2014/129 (2014). http://eprint.iacr.org/2014/129. Accepted to ACM CCS 2014
8. Bitcoin.org. Developer reference. http://bitcoin.org/en/developer-reference. Accessed on 20 October 2014
9. Bitcoin.org. List of bitcoin wallets. http://bitcoin.org/en/choose-your-wallet. Accessed on 20 October 2014
10. Boldyreva, A., Cash, D., Fischlin, M., Warinschi, B.: Foundations of non-malleable hash and one-way functions. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 524–541. Springer, Heidelberg (2009)
11. Vitalik, B.: Bitcoin network shaken by blockchain fork, March 2013. Bitcoin Magazine. http://bitcoinmagazine.com/3668/bitcoin-network-shaken-by-blockchain-fork
12. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: 34th Annual ACM Symposium on Theory of Computing, pp. 494–503. ACM Press (2002)

13. Decker, C., Wattenhofer, R.: Bitcoin transaction malleability and mtgox. In: Kutyłowski, M., Vaidya, J. (eds.) ICAIS 2014, Part II. LNCS, vol. 8713, pp. 313–326. Springer, Heidelberg (2014)
14. Dodis, Y., Wichs, D.: Non-malleable extractors and symmetric key cryptography from weak secrets. In: Mitzenmacher, M. (ed.) 41st Annual ACM Symposium on Theory of Computing, pp. 601–610. ACM Press (2009)
15. Dolev, D., Dwork, C., Naor, M.: Nonmalleable cryptography. SIAM J. Comput. **30**(2), 391–437 (2000)
16. Dziembowski, S., Kazana, T., Obremski, M.: Non-malleable codes from two-source extractors. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 239–257. Springer, Heidelberg (2013)
17. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. Consulted **1**, 28 (2008)
18. Weisenthal, J.: Bitcoin just completely crashed as major exchange says withdrawals remain halted, Business Insider (2014). http://www.businessinsider.com/mtgox-statement-on-withdrawals-2014-2
19. Bitcoin Wiki. Contracts. http://en.bitcoin.it/wiki/Contracts. Accessed on 20 October 2014
20. Bitcoin Wiki. Main page. http://en.bitcoin.it/. Accessed on 20 October 2014
21. Bitcoin Wiki. Transaction malleability. http://en.bitcoin.it/wiki/Transaction_Malleability. Accessed on 20 October 2014
22. Wuille, P.: Bitcoin improvement proposal: dealing with malleability. http://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki. Accessed on 20 October 2014