Double-spending Prevention for Bitcoin zero-confirmation transactions

Cristina Pérez-Solà, Sergi Delgado-Segura,
Guillermo Navarro-Arribas, Jordi Herrera-Joancomartí

Department of Information Engineering and Communications,
Universitat Autònoma de Barcelona
{cperez, sdelgado, gnavarro, jherrera}@deic.uab.cat

Abstract. Zero-confirmation transactions, i.e., transactions that have been broadcast but are still pending to be included in the blockchain, have gained attention in order to enable fast payments in Bitcoin, shortening the time for performing payments. Fast payments are desirable in certain scenarios, for instance, when buying in vending machines, fast food restaurants, or withdrawing from an ATM. Despite being fast propagated through the network, zero-confirmation transactions are not protected against double-spending attacks, since the double spending protection Bitcoin offers relays on the blockchain and, by definition, such transactions are not yet included in it. In this paper, we propose a double-spending prevention mechanism for Bitcoin zero-confirmation transactions. Our proposal is based on exploiting the flexibility of the Bitcoin scripting language together with a well known vulnerability of the ECDSA signature scheme to discourage attackers from performing such an attack.

1 Introduction

Double spending, or spending a currency token more than once, is the main security problem that digital currencies have to deal with. Unlike physical money, where the physical token is hard to copy and once it has been spent it passes effectively to the recipients' hands, digital currency tokens can be easily copied and double spent if security mechanisms are not properly applied.

Bitcoin deals with this double spending problem by building an append-only ledger, the blockchain, that is replicated in every single Bitcoin full node. The blockchain is made of blocks that are stacked on top of each other. Blocks are made of entries, which contain some source (inputs) and destination (outputs). Entries in the aforementioned blockchain are called transactions, and they are used to transfer bitcoins between different users, typically identified by their Bitcoin addresses. Bitcoin transfer is performed by using an unspent output of a previous transaction (UTXO) as the input of a new one. Therefore, Bitcoin transactions consume previous outputs and generate new ones.

Transactions are broadcast over the Bitcoin P2P network aiming to reach every single node, each one of whom will check the transaction correctness and store the valid ones in their local memory pool of transactions (mempool). Eventually, every valid transaction will be included into a new block by a miner, never before checking, among other things, that the transaction does not claim already spent outputs. Hence, double spending is prevented once a transaction has been included in a block, since it has been proven that no previous transaction spends from the same outputs, and future transactions will be prevented to do so.[1]

However, transactions are not automatically included in blocks. In the meantime between transaction broadcasting and its publication in a block, transactions are known as zero-confirmation transactions, and they are just stored in the mempool of the nodes that have received them. Therefore, during this time window different transactions spending the same outputs can be spread through the Bitcoin P2P network. Having received a transaction spending from an output which has been never used before, the default behavior of a node will be to store the transaction in his local mempool and drop any other incoming transactions trying to spend from the same source. However, different nodes could receive different transaction spending from the same source, and double spending could be attempted. For instance, suppose two transactions (tx_1 and tx_2) that spend the same output from a previous transaction (tx_0) are created by an attacker A. tx_1 is used to pay for some goods to B, while tx_2 is used to return the funds to the attacker. In this scenario, if A can make B believe that tx_1 is the only transaction spending from tx_0's output, but tx_2 finally becomes included into a block, the attack is successful. Figure 1 depicts the aforementioned example.

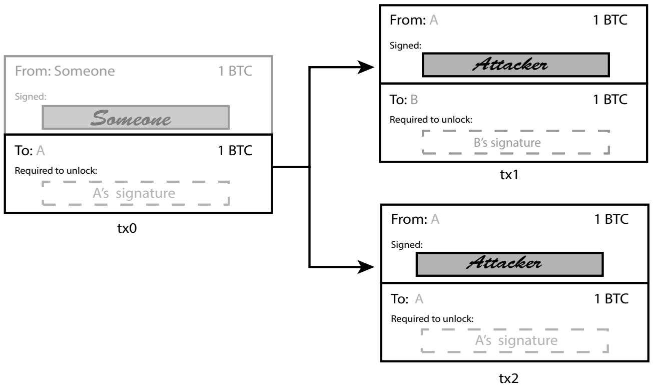

Fig. 1. Double-spending transactions.

Notice that only one of the double-spending transactions will be included in a block due to the double spending protection that Bitcoin achieves by design.

[1] Although it can be argued that Bitcoin transactions are not final since blockchain forks may always occur, throughout this paper, to simplify the discussion, we assume that once a transaction appears in the blockchain it is final.

However, in case tx_2 is included in a block, tx_1 will be discarded and the double-spending attack will succeed.

In this paper we propose a solution to the double spending problem for Bitcoin zero-confirmation transactions. In our proposal, any single observer who identifies a double spending attempt may take part and punish the attacker. Moreover, our solution discourages the attacker to even attempt the double-spending, because doing so makes him risk losing an amount of bitcoins bigger than the double-spent amount. Our solution benefits fast-payment scenarios, like in-shop purchases or trading platforms, where the transfer bitcoin-product/service cannot wait until the transaction is confirmed in the blockchain. Finally, although this paper is focused in Bitcoin transactions for conciseness, a similar construction can be developed for other cryptocurrencies with zero-confirmation transactions and based on ECDSA signatures, for instance Litecoin or Dogecoin.

The paper is structured as follows. First, Section 2 reviews the state of the art in double spending protection for fast payment transactions within Bitcoin. Then, Section 3 introduces the most important concepts about Bitcoin transactions, needed to understand our contribution. Next, Section 4 explains our proposed mechanism for discouraging double spending attempts. After that, Section 5 analyses the benefits each party obtains when applying the proposed protocol. Finally, Section 6 presents the conclusions and provides guidelines for further research. Additionally, Appendix A provides implementation details and Bitcoin scripts to deploy the proposed scheme.

2 State of the art

Double spending attacks on zero-confirmation transactions in Bitcoin were first analyzed by Karame et al. [1, 2]. The authors show that, with some reasonable assumptions and without the need of special computation nor much network overhead, an attacker has a great probability of succeeding with a double spending attack. Moreover, the authors also show how basic countermeasures such as waiting a few seconds before accepting the payment or adding observers that report back to the payee are not enough on their own to avoid these type of attacks. Additionally, they propose another countermeasure, consisting of modifying the protocol rules so that nodes forward double spending transactions instead of dropping them. By doing so all nodes may be notified of the double spending attempts. However, this mechanism facilitates denial of service attacks and, while nodes will indeed be able to see both transactions, they do not have means to distinguish which of the two is the original one.

The idea of monitoring observers has been somehow implemented by companies such as GAP600 [3] in order to provide risk evaluation for accepting zero-confirmation transactions. Nevertheless, the company does not provide details about how is the monitoring or the evaluation performed.

Regarding the mitigation of double spending attacks, Decker et al. [4] proposed some other countermeasures that can reduce the merchant's likelihood of being deceived by an attacker: requiring the merchant to be connected to a large random sample of nodes of the network and not accepting incoming connections. This means that the attacker cannot send transactions directly to the merchant nor is he able to identify the merchant's neighbors.

Other research works have indeed demonstrated that this kind of attacks were possible, and not only was the attacker able to identify the merchant's neighbors but also force them to be a set of nodes controlled by the attacker [5–7]. Moreover, non of the countermeasures forces attackers to stop trying, nor punishes them by not doing so. However, if some penalty could be applied to anyone who tries to perform such an attack, and this penalty could be applied by any node who detects it (instead of just dropping the transactions), attackers may be discouraged to even try it.

As we have seen in other recent Bitcoin proposals, such as the Lightning network [8], losing all the funds of a transaction between two actors is a good discouragement for them not to deceive their counterpart. In such a way, and alternatively to other works [4], we address the double spending problem in fast payments by introducing a penalty for the attacker.

3 Background on Bitcoin transactions

As we have already stated, Bitcoin transactions are the tool for value transfer in the Bitcoin protocol. Bitcoin transfer is performed by using an unspent output of a previous transaction (UTXO) as the input of a new one. Therefore, Bitcoin transactions consume previous outputs and generate new ones.

The input of a Bitcoin transaction contains three fields. The first one indicates which output is being spent by providing the identifier of the transaction where the output is included. The second field includes the index of the output that is going to be spent (since a single transaction may include multiple outputs). Of course, such output must not have been spent before, so it must be an unspent transaction output (UTXO). Finally, the third field, identified as `scriptSig`, provides the conditions needed to be met for the transaction to be valid and the payment be correctly performed. Such conditions were defined in the output that is going to be spent. Both conditions, the specification in the output and the fulfillment in the input, are codified using the Bitcoin scripting language [9], a stack-based language defined in the Bitcoin protocol.

The output of a Bitcoin transaction includes two fields. The first one indicates the amount of bitcoins that will be deposited in such output.[2] The second field, named `scriptPubKey`, defines the conditions under which this output could be spent.

The most general condition for spending an output is to be able to perform a digital signature using the private key associated with a Bitcoin address (and this is the reason the field is called `scriptSig`). However, the Bitcoin scripting

[2] Notice that the input does not provide any amount of bitcoins since all bitcoins of the output that is referenced will be transferred. The difference between those amounts is the transaction fee collected by the miner.

language is flexible enough to allow the definition of many other scripts that encode different conditions under which the outputs may be spent, as we describe in the next section.

3.1 The Bitcoin scripting language

In order to spend an UTXO, the locking conditions specified in its `scriptPubkey` field have to be met. The fulfilment of these conditions is provided by the values included in the `scriptSig` field of the input that spends the UTXO. To evaluate if an input is able to spend a corresponding output, the code included in the `scriptPubKey` is appended to the values included in the `scriptSig`, and the complete set of instructions is executed. Only if the execution returns true, the input is able to spend the UTXO. Notice that this general approach allows not only to spend UTXO based on digital signatures but also to create much richer constructions, the so called smart contracts.

Such smart contracts can indeed specify complex conditions.[3] For instance, besides Pay-to-Public-Key-Hash (P2PKH) or Pay-to-Public-Key (P2PK) outputs where a standard digital signature must be provided to spend the UTXO, there exist multi-signature constructions, where an UTXO is locked under n public keys, and at least m matching signatures must be provided in the corresponding input in order to spend it. Furthermore, even more general constructions can be deployed using Pay-to-Script-Hash (P2SH) outputs. P2SH outputs encode an ad-hoc set of instructions. The resulting hash of the instructions set is included in the `scriptPubKey` field of the UTXO as the locking condition. The input that spends such UTXO must provide the corresponding script whose hash matches the value specified in the `scriptPubKey` field of the UTXO. Moreover, the script must also evaluate to true in order to consider the validation a success.

3.2 Digital signatures on Bitcoin

Digital signatures in the Bitcoin system are performed through the Elliptic Curve Digital Signature Algorithm (ECDSA). ECDSA has a set of system parameters: an elliptic curve field and equation $\mathtt{C}$, a generator G of the elliptic curve $\mathtt{C}$, and a prime q which corresponds to the order of G. The values for these parameters are defined to be `secp256k1` [10] for Bitcoin.

Let $*$ be the operation of multiplying an elliptic curve point by a scalar. Given a specific configuration of the parameters and a private key d, the ECDSA signature algorithm over the message m is defined as follows:

1. Randomly choose an integer k in $[1, q-1]$
2. $(x, y) = k * G$
3. $r = x \mod q$
4. $s = k^{-1}(m + rd) \mod q$

[3] An interested reader could refer to [9] for additional information about Bitcoin smart contracts and script types.

5. If either s or r are 0, go back to step 1.
6. Output: $sig(m) = (r, s)$

The ECDSA signature scheme is therefore probabilistic, that is, there exist many different valid signatures made with the same private key for the same message. The selection of a specific signature from the set of valid ones is determined by the election of the integer k.

There exists a well known ECDSA signature vulnerability (also present in the non-elliptic curve signature scheme of ElGamal and its popular variant, DSA [11, 12]) by which an attacker that observes two signatures of different messages made with the same private key is able to extract the private key if the signer reuses the same k selected on step 1. Therefore, the selection of k is critical to the security of the system.

Indeed, given two ECDSA signatures that have been created using the same k and the same private key, $sig_1 = sig(m_1) = (r, s_1)$ and $sig_2 = sig(m_2) = (r, s_2)$ with $m_1 \neq m_2$, an attacker that obtains m_1, sig_1 and m_2, sig_2 may derive the private key d:

1. Recall that, by the definition of the signature scheme:
$$\begin{aligned} s_1 = k^{-1}(m_1 + rd) \mod q \Rightarrow ks_1 = m_1 + rd \mod q \\ s_2 = k^{-1}(m_2 + rd) \mod q \Rightarrow ks_2 = m_2 + rd \mod q \end{aligned}$$
Note that, since r is deterministically generated from k and the fixed parameters of the scheme, the r values of both signatures will be the same.
2. The attacker learns k by computing $k = \frac{m_2 - m_1}{s_2 - s_1}$
3. The attacker learns the private key d by computing $d = \frac{s_1 k - m_1}{r}$ or $d = \frac{s_2 k - m_2}{r}$

Moreover, the leakage of private key information is not only restricted to the case where the exact same k values are used, but also to situations when similar k values are generated [13, 14].

Some Bitcoin wallets adopted deterministic ECDSA after this vulnerability was found to affect some Bitcoin transactions [15–17].

Taking advantage of such vulnerability to perform a private key disclosure in Bitcoin has been previously used for timestamping in data commitment schemes by Clark and Essex [18].

4 Double-spending protection mechanism

Our proposed scheme discourages transaction signers from performing double spending attacks in zero-confirmation transactions used in fast payment scenarios. Fast payment scenarios are those where the merchant delivers the goods or services when seeing the payment transaction in the Bitcoin network, without waiting for the transaction to be confirmed. Examples of such scenarios are on-site shopping where the buyer cannot wait 10 minutes to leave the shop after purchase or in trading platforms where a timely transaction can save/earn you money.

In our scenario, we assume that the adversary is the buyer that pays for some goods to a merchant, and that may have incentives to try to double-spending the payment in order to finish the interaction with both the goods and the payment amount. We assume that he can perform a double-spending attack by generating multiple transactions that spend from the same output and broadcasting them selectively in the Bitcoin P2P network. Additionally, we also assume all peers of the network have the same capabilities, that is, they are able to generate and broadcast double spend transactions (if they know the private key needed to generate a signature).

In order to discourage double-spending attacks, we propose a mechanism to construct special transaction outputs. Such outputs can be spent with a single signature but have the property that if two different signatures for the same output are disclosed (for instance, in two different transactions spending the same output as a double-spending attack), the private key used to sign the transaction is revealed. This allows any observer to generate a third transaction spending the same output and sending the amount to an address controlled by himself.

To allow such construction, we propose a new Bitcoin script that we call **fixed-r pay-to-pubkey script (FR-P2PK)**. Such script is a variant of the standard pay-to-pubkey (P2PK) script where a signature is required to redeem, but a FR-P2PK script adds an additional condition to be able to spend the output: the signature must be made with a specific r value. Then, due to the ECDSA vulnerability described in Section 3.2, this special condition discourages double-spending of that output. Indeed, if the sender generates another transaction that spends the same output and propagates it through the P2P network, the sender risks losing all the funds deposited in the address, because any peer that captures both transactions will be able to derive the private key.

4.1 Basic protection mechanism

Let Alice be a user that wants to take advantage of the proposed double-spending protection mechanism and let $\{PK_a, SK_a\}$ be an ECDSA key pair belonging to Alice.

The double spending protection mechanism is made of two phases: *initialization*, that is performed before the payment is made, and *fast-payment*, where the payment is executed.

Initialization: The initialization phase is performed beforehand. During this phase, Alice generates a *funding transaction* that transfers some funds from an output in her control to a FR-P2PK output also under her control. In order to do so, Alice chooses a random integer k and a public key PK_a (for which she knows the associated secret key SK_a), constructs the FR-P2PK output, and sends some funds to an output in her control (see Figure 2). Alice broadcasts the funding transaction and waits for the transaction to be confirmed, upon which the initialization phase is considered terminated.

A single funding transaction may include multiple FR-P2PK outputs (with different public keys) in order to allow Alice to use the proposed protection mechanism multiple times. Moreover, Alice may repeat the initialization phase if she runs out of unspent FR-P2PK outputs. Note that this phase is independent of any specific payments and that Alice alone participates in the procedure. Additionally, notice that Alice remains in control of all the funds deposited by the funding transaction, and is able to transfer them back to an standard output whenever she wants.

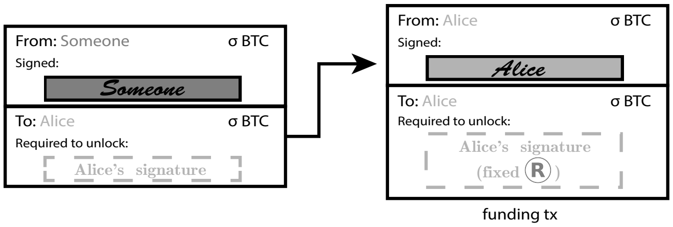

Fig. 2. Creation of the funding transaction.

At some point in the future, Alice wants to send some amount of bitcoins to another user Bob. Alice does not want to wait for the confirmation of the transaction and Bob is not willing to accept the transaction without confirmation. So they decide to use the proposed double spending protection mechanism, executing the fast-payment phase.

Fast payment: Alice creates a *fast-payment transaction* that pays to Bob spending from the FR-P2PK output of the funding transaction. The input script in the fast-payment transaction forces Alice (the redeemer) not only to prove he has the private key SK_a associated to the given public key PK_a by creating a valid signature, but also to deliver a signature that has been made using the specific k value that Alice chose during the initialization phase (see Figure 3). Alice broadcasts the fast-payment transaction to the Bitcoin P2P network.

Then, when Bob sees the fast-payment transaction in his mempool, he can validate that the output script of the funding transaction spent by the fast payment transaction is indeed a FR-P2PK script. If the validation is correct, Bob knows that if Alice tries to double spend the transaction she takes the risk of loosing the Bitcoins of that output.

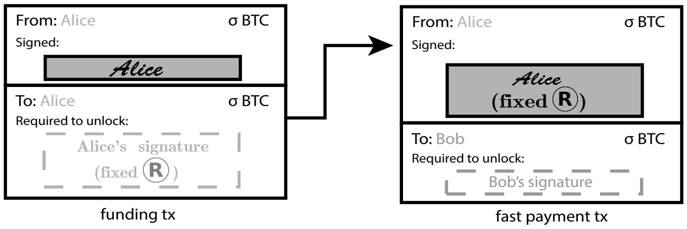

Fig. 3. Fast payment transaction.

If Alice wants to try to double spend the fast-payment transaction (see Figure 4, double-spending attempt), she needs to create a *double spending transaction* that also spends the FR-P2PK output of the funding transaction. This double spending transaction has to be valid, so it needs to include a (second) signature made with SK_a and the k value chosen on the initialization phase. Hence the moment the double spend transaction is created, there exists two different signatures made with the same private key SK_a using the same r. The signatures will be indeed different, since the signed content (i.e. the transactions) will also be different. Recall that, because of the aforementioned ECDSA vulnerability, knowing two different signatures made by the same private key with the same k value is enough to derive the private key used to sign.

Therefore, if Alice broadcasts the double spend transaction, she risks loosing her funds. This happens because any observer that receives both transactions (the fast-payment transaction and the double spend transaction) will be able to derive Alice's secret key SK_a and, as a consequence, create a third transaction, the *penalty transaction* that also spends the FR-P2PK output of the funding transaction but that sends the bitcoins to the observer. Note that this strategy may be performed simultaneously by any observer, ending with multiple penalty transactions, as it is depicted in Figure 4.

4.2 Disincentive-based protection mechanism

The basic protection method described in the previous section has a clear drawback. Suppose that Alice performs a purchase to Bob's shop and Bob accepts a fast transaction from Alice. When such transaction is received, Bob delivers the goods to Alice. However, once Alice has the goods, she may try to perform a double-spending attack. In case an observer sees both the fast-payment transaction and the double spend-transaction, he constructs the penalty transaction, and manages to get it accepted in the blockchain, Alice loses her funds but Bob does not receive the payment. In that case, Bob may have complied with his part of the agreement (e.g. delivered the bought goods) but will not receive the agreed amount of bitcoins in exchange. Alice would have paid the agreed amount

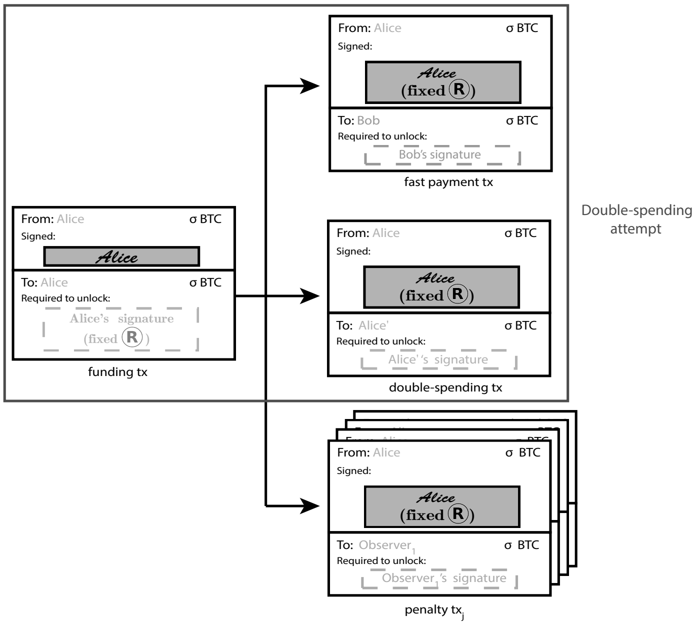

Fig. 4. Transactions involved in the scheme.

of bitcoins to a third party (the observer) instead of paying them to Bob, but she would remain in possession of the goods. The observer would obtain the total amount of the transaction. As a consequence, Alice will not loose anything by trying the double-spend (just the amount she is already willing to pay for it), and thus the proposed method may not be discouraging enough to prevent double spending attempts.

However, a minor modification to the proposed method is enough to discourage Alice from attempting any double-spend: enforcing that the amount deposited to the FR-P2PK output of the funding transaction is higher than the paid amount by a certain factor λ. Recall that a Bitcoin UTXO must be spent in its totality (i.e. it is not possible to spend a part of an UTXO). Therefore, if the FR-P2PK output has an amount bigger than what Alice must pay to Bob, Alice proceeds to create the fast-payment transaction including two different outputs: one that pays to Bob the agreed amount, and the other that pays back to her the change. This has no consequences on the normal operation of the protocol, that is, if Alice well-behaves, Bob ends up with his payment and Alice gets her change back. However, because the entire FR-P2PK output is spent, if Alice tries to double-spend the fast payment she risks loosing the entire amount of the FR-P2PK output, and not only the amount paid to Bob.

As we will discuss in Section 5, by adjusting the λ factor Alice's penalty for double-spending can also be adjusted (and thus Bob's confidence on the fast payment).

Moreover, note that the fast-payment transaction may also have multiple inputs spending different FR-P2PK outputs. This allows Alice to be able to perform payments of different amounts and with different penalty levels without having to freeze a high amount of bitcoins into FR-P2PK outputs.

The role of the observers. The funding transaction is confirmed before starting the fast-payment phase, so any full node of the network is aware of its existence. Moreover, because it has an output with an easily identifiable script, the FR-P2PK script, any observer aware of the specification of our proposed mechanism is able to identify the transaction as a funding transaction belonging to our protocol. Therefore, such an observer will be able to monitor his mempool, looking for transactions that spend the FR-P2PK output. Once a transaction spending from the FR-P2PK output is seen, the observer is able to actively listen the network, searching for any other transaction spending the same output. If the observer is able to catch a double spending transaction, he should be able to construct a penalty transaction, moving the funds to an address controlled by himself. If the observer does not capture a double spending transaction, he may stop this active listening period and return to its normal behaviour when a transaction spending the FR-P2PK output is included in the blockchain.

5 Proposal analysis

In this section we provide an analysis of the possible outcomes of performing a payment with the proposed mechanism. The analysis measures the benefits of each party taking part in the system to show how it discourages double-spending attacks. Table 1 summarizes the notation used in this section.

Symbol	Meaning
τ_f	Fast-payment transaction
τ_d	Double-spending transaction
τ_{p_j}	Penalty transaction j
$Pr[\tau_x \in \mathcal{B}]$	Probability that transaction τ_x is included in the blockchain
σ	Payment amount
$\lambda \cdot \sigma$	Funding transaction output amount
γ	Value of goods

Table 1. Notation summary

Our analysis makes the follow assumptions. First of all, we assume that Alice always generates the fast payment transaction since it is the triggering action for the payment. Once the fast payment has been generated, we assume that Bob sees the payment and, at that time and acting honestly, he delivers the goods to Alice. Furthermore, to focus the analysis in the proposed mechanism, we assume that at least one of the transactions of the system τ_f, τ_d, or τ_{p_j} will be confirmed and that transactions do not include fees. Of course, due to the double-spending protection of Bitcoin for on-chain transactions, at most one of these transactions gets into the blockchain, that is, the events $\tau_f \in \mathcal{B}$, $\tau_d \in \mathcal{B}$, and $\tau_{p_j} \in \mathcal{B}$ are mutually exclusive. Finally, notice that $Pr[\tau_f \in \mathcal{B}] + Pr[\tau_d \in \mathcal{B}] + \sum_j Pr[\tau_{p_j} \in \mathcal{B}] = 1$, since such probabilities depend on the distribution hash rate devoted to the interests of every set of users (Alice, Bob and the rest of the network, acting as observers) and we can assume that such sets will be disjunct.

Taking into account these assumptions, the proposed protocol may end in three different states, as described by Figure 5. If the fast payment transaction τ_f gets confirmed, then Bob receives the payment for the goods, Alice receives the change (the amount deposited to the funding transaction minus the payment) and the goods, and the observer does not intervene. If the double spending transaction τ_d is confirmed, then Alice gets everything (the full amount deposited in the funding transaction and the goods) and therefore both Bob and the observers do not obtain anything. Finally, if one of the penalty transactions τ_{p_j} is confirmed, then Alice obtains the goods but looses the full deposited amount that goes to the observer. Figure 5 also describes the possible paths that end up in each of the states.

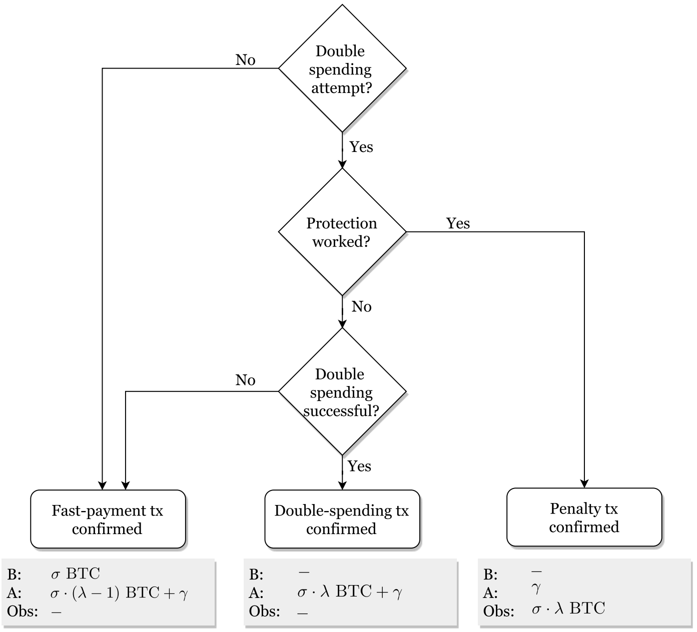
Double spending attempt?
No
Yes
Protection worked?
Yes
No
Double spending successful?
No
Yes
Fast-payment tx confirmed
Double-spending tx confirmed
Penalty tx confirmed
B: σ BTC
A: $\sigma \cdot (\lambda - 1)$ BTC $+ \gamma$
Obs: –
B: –
A: $\sigma \cdot \lambda$ BTC $+ \gamma$
Obs: –
B: –
A: γ
Obs: $\sigma \cdot \lambda$ BTC

Fig. 5. Flow chart showing the protocol's final states and the paths leading to them.

We define the payoff $\mathcal{P}$ of any party participating in the protocol as the gains (or losses) obtained by deviating from the correct operation of the protocol. That is, the payoff of all parties (Alice, Bob, and the observers) will be 0 when no double spending is attempted (leftmost box in Figure 5).[4] In that case, there will be an equilibrium, since Alice pays the specified price for some goods and obtains the goods in exchange; Bob delivers the goods and gets paid for them; and the observers do not intervene. On the contrary, if Alice tries to double spend the payment, the equilibrium may be altered and the payoff will reflect the gains or losses each party assumes.

Then, Bob's payoff function $\mathcal{P}_B$ is given by the following expression:

$$\begin{aligned}\mathcal{P}_B &= Pr[\tau_f \in \mathcal{B}] \cdot (\sigma - \gamma) - Pr[\tau_d \in \mathcal{B}] \cdot \gamma - Pr[\tau_{p_j} \in \mathcal{B}] \cdot \gamma = \\ &= Pr[\tau_f \in \mathcal{B}] \cdot \sigma - \gamma\end{aligned}$$

Note that, for fixed σ and γ, Bob's payoff only depends on $Pr[\tau_f \in \mathcal{B}]$. Recall that our mechanism tries to disincentivize Alice from double-spending the payment transaction, but does not directly benefit the merchant (regardless of the λ value used by the protocol).[5]

In a similar way, Alice's payoff $\mathcal{P}_A$ is given by:

$$\mathcal{P}_A = Pr[\tau_f \in \mathcal{B}] \cdot (\gamma - \sigma) + Pr[\tau_d \in \mathcal{B}] \cdot (\gamma) + Pr[\tau_{p_j} \in \mathcal{B}] \cdot (\gamma - \sigma\lambda)$$

Alice's maximum payoff is, therefore, γ, and is obtained when Alice's successfully double spends the transaction, thus keeping the goods γ without paying anything. However, Alice's minimum payoff (maximum losses) depends on λ, a parameter that can be adjusted in our protocol. Therefore, by adjusting λ, the protocol allows to tune Alice's losses, and so the risks she assumes by trying to perform a double spending attack. The bigger the λ is, the higher the risks Alice's faces on a double spend attempt.

Finally, an observer's j payoff is given by the expression:

$$\mathcal{P}_{O_j} = Pr[\tau_{p_j} \in \mathcal{B}] \cdot (\sigma\lambda)$$

Figure 6 shows the evolution of the parties payoffs as a function of $Pr[\tau_f \in \mathcal{B}]$ and $Pr[\tau_d \in \mathcal{B}]$[6] for the case where $\sigma = \gamma$ (the value of goods is equal to the price it is paid for them), for different values of the parameter λ. The payoff dimension is measured based on the value σ. That means that a payoff of 3 implies a benefit of 3 times the value of σ while a payoff of -3 implies a lost of 3 times the value of σ. The payoff results are thus proportional to σ.

The graphics show that, as expected, when there is no double spending attempt ($Pr[\tau_f \in \mathcal{B}] = 1$) there is an equilibrium in the parties' payoffs, and in

[4] We assume that he price of the goods is equal to the value of the goods.

[5] Note, however, that Bob may also act as an observer himself, being able to create a penalty transaction and trying to gain the observer's payoff.

[6] Since $Pr[\tau_f \in \mathcal{B}] + Pr[\tau_d \in \mathcal{B}] + \sum_j Pr[\tau_{p_j} \in \mathcal{B}] = 1$, fixing the first two probabilities uniquely determines the third operand.

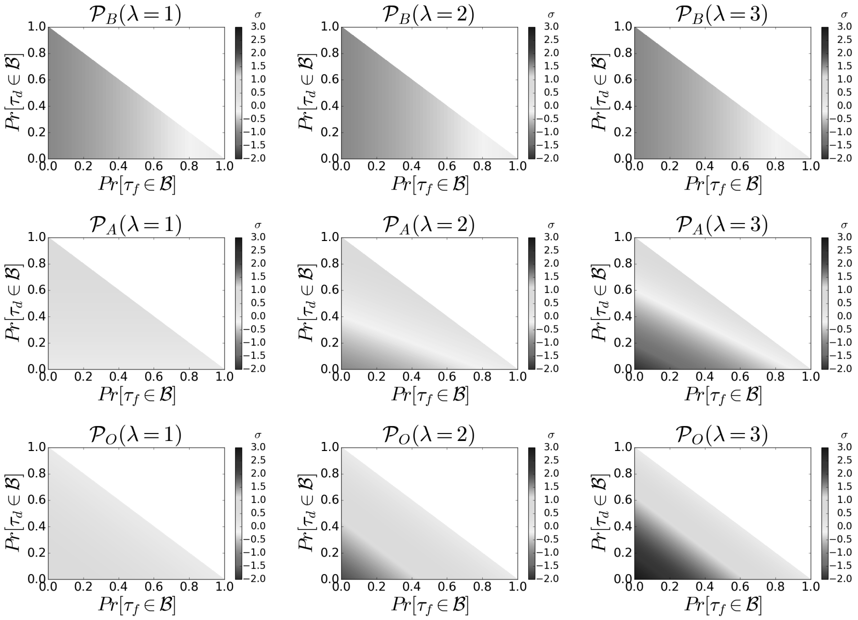

Fig. 6. Parties payoffs for $\sigma = \gamma$.

all graphics we obtain $\mathcal{P}_B = \mathcal{P}_A = \mathcal{P}_{O_j} = 0$ (green zone). Note that, as λ increases, Bob's payoff (first three graphics) remains exactly the same since his payoff is independent of the parameter λ. On the contrary, Alice's payoff (next three graphics) depends on λ. With $\lambda = 1$, Alice's payoff is always positive or zero: Alice does not lose anything by trying to double spend and may even gain something if the attack is successful. That situation is the basic protection mechanism described in Section 4.1. However, by increasing λ the scenario changes radically for Alice: the probabilities range at which Alice gains something from the attack decrease fast and, at the same time, for some probability values she even starts to get a negative payoff (that is, she has to assume losses). Finally, notice that the observer's payoff (last three graphics) is never negative, and his gains increase with λ.

Notice that our analysis does not assume any specific values on the probabilities $Pr[\tau_f \in \mathcal{B}]$, $Pr[\tau_d \in \mathcal{B}]$, and $Pr[\tau_{p_j} \in \mathcal{B}]$. However, as we have already indicated, such probabilities depend on the hash distribution of the Bitcoin network among mining the transactions, τ_f, τ_d, and τ_{p_j}. For that reason, in case the hash rate devoted to τ_d with respect the rest is low, the graphics show that Alice's payoff, for values $\lambda > 1$, is moving in the red zone thus being negative (Alice is loosing money). The greater the λ value the bigger the red zone.

6 Conclusions

The speed at which payments in blockchain based cryptocurrencies can be performed is lower bounded by the block generation interval, which in Bitcoin is fixed to 10 minutes. In order to provide fast payments, one of the alternatives used in these scenarios is to rely on zero-confirmation transactions, that is, transactions that have been seen on the network but have not yet been included in the blockchain. Experimental analysis have shown that, in Bitcoin, most of the transactions propagated through the network reach 75% of the nodes in less than 8 seconds [19], which is two orders of magnitude faster than the block production interval. However, zero-confirmation transactions are not secured by the standard Bitcoin double-spending protection mechanism, since this mechanism is applied to transactions included in the blockchain and zero-confirmation transactions are not yet in blocks by definition.

In this paper, we have presented a mechanism to secure fast payments within Bitcoin by reducing the risk of double-spending attacks in zero-confirmation transactions. The proposed mechanism discourages double spending attempts by creating a special type of outputs that enforce private key disclosure in case of double-spending attempt. Any Bitcoin network user may act as an observer and obtain a reward by detecting double-spending attempts. The reward the observer receives is equal to the price the attacker pays as punishment for having tried to double spend a transaction.

Further research will be focused on experimentally testing the proposed approach in a Bitcoin-like P2P network, in order to quantify the probabilities of each transaction entering the blockchain depending on the exact capabilities of the attacker (both in terms of network connectivity and hash power) and the percentage of nodes of the network that are aware of the existence of the protection mechanism. In turn, this would allow us to better evaluate the risks the merchant is facing with each transaction and to study the overhead of transactions relayed through the network by the usage of our protocol.

References

1. Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 906–917. ACM, 2012.
2. Ghassan O. Karame, Elli Androulaki, Marc Roeschlin, Arthur Gervais, and Srdjan Čapkun. Misbehavior in bitcoin: A study of double-spending and accountability. *ACM Trans. Inf. Syst. Secur.*, 18(1):2:1–2:32, May 2015.
3. GAP600. Gap600 bitcoin transactions guaranteed. http://gap600.com/, 2017.
4. Tobias Bamert, Christian Decker, Lennart Elsen, Roger Wattenhofer, and Samuel Welten. Have a snack, pay with bitcoins. In *Proceedings of the IEEE Internation Conference on Peer-to-Peer Computing (P2P)*, Trento, Italy, 2013., 2013.
5. Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in bitcoin p2p network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 15–29. ACM, 2014.

6. Alex Biryukov and Ivan Pustogarov. Bitcoin over tor isn't a good idea. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 122–134. IEEE, 2015.
7. Joshua A. Kroll, Ian C. Davey, and Edward W. Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *The Twelfth Workshop on the Economics of Information Security (WEIS 2013)*, June 2013.
8. Joseph Poon and Thaddeus Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. Technical report, 2015. https://lightning.network.
9. Andreas M Antonopoulos. Transaction scripts and script language. In *Mastering Bitcoin: unlocking digital cryptocurrencies*, chapter 5. O'Reilly Media, Inc., 2014.
10. Certicom Research. Sec 2: Recommended elliptic curve domain parameters. Technical report, Certicom Corp., January 2010.
11. Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.
12. Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.
13. Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, 2002.
14. Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. Pseudo-random number generation within cryptographic algorithms: The DDS case. In *Annual International Cryptology Conference*, pages 277–291. Springer, 1997.
15. Nils Schneider. Recovering Bitcoin private keys using weak signatures from the blockchain, 2013.
16. Filippo Valsorda. Exploiting ECDSA failures in the Bitcoin blockchain, 2014.
17. Bitcoin.org. Android security vulnerability, 2013.
18. Jeremy Clark and Aleksander Essex. Commitcoin: Carbon dating commitments with bitcoin. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 390–398. Springer Berlin Heidelberg, 2012.
19. Christian Decker. Data propagation: How fast does information move in the network? http://bitcoinstats.com/network/propagation/, 2017.

Appendix A: Implementation details

In this appendix, we describe how to construct the FR-P2PK output of the funding transaction as well as the inputs of the transactions that spend it, taking into account Bitcoin's signature format and scripting language.

First of all, notice that it would be possible to encapsulate the proposed FR-P2PK script into a standard P2SH output. However, doing so makes the funding transaction no longer recognizable as belonging to our protocol by external observers. Therefore, an observer that is aware of the existence of our protocol would be able to detect that the mechanism is being used only after one of the transactions spending the encapsulated FR-P2PK is seen in the network. This transaction will include the FR-P2PK script in the `scriptSig` (input script). The moment the observer processes this script, he can start the active listening period in which he looks for other transactions spending from the same funding transaction output. Because timing is critical in our scenario, we argue that using directly a FR-P2PK output in the funding transaction is the best alternative. However, encapsulating the FR-P2PK script into a P2SH output offers similar protection. On the other hand, we assume that the fast payment transactions in our scheme are not flagged as replaceable with the Bitcoin's Replace by Fee mechanism (RBF), so they cannot be replaced by other newer transactions with higher fees.

In a Bitcoin transaction, signatures are represented by a single hexadecimal value, which corresponds to the DER encoding of the two-element sequence of the r and s integers that make up an ECDSA signature. Figure 7 describes the format of a Bitcoin signature. Each integer r and s (see Section 3.2 regarding the notation) is encoded with three different fields: a 1-byte field with the `0x02` flag denoting the integer type, a 1-byte field with the size l of the integer (in bytes), and an l-byte field with the integer value itself. Then, the signature includes a 1-byte field with a flag denoting a sequence (`0x30`), a 1-byte field with the length of the sequence, the sequence of the two integers, and finally a 1-byte field with the hash type, a flag that indicates the parts of the transaction that are hashed and signed.

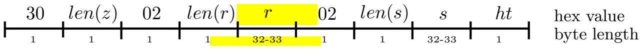

Fig. 7. Bitcoin signature format.

Both r and s are 32 byte integers. However, when the first bit of any of the values is set (that is, the fist byte is $>$ `0x7f`), an additional byte (`0x00`) is added in front of the value, thus making it 33 byte long. The reason is that DER rules interpret this first bit as a sign, and therefore not adding `0x00` would cause the value to be interpreted as negative. Therefore, Bitcoin signatures range from 71 to 73 bytes. For the sake of simplicity, let us assume that we are dealing with 71-byte signatures, that is, signatures where both r and s are 32 byte long (Figure 8a).

Taking into account the format of signatures in Bitcoin, the script of a FR-P2PK output is defined as follows:

```
ScriptPubKey: OP_DUP <pubKey> OP_CHECKSIGVERIFY
              OP_SIZE <0x47> OP_EQUALVERIFY
              <sigmask> OP_AND <r> OP_EQUAL
ScriptSig:    <sig>
```

where

- `<pubKey>` is the public key that will be used to validate the signature,
- `<sigmask>` is a 71-byte array that has ones in the positions where r and ht are specified and zeros in the rest of positions, and
- `<r>` is the 71-byte array that represents the integer r in DER format in the positions where it is found in a signature, `0x01` in the ht field, and zeros in the rest of positions.

Figures 8b-c show the construction of `<sigmask>` and `<r>`, respectively. Regarding the construction of the byte array `<r>`, on the one hand the integer r is derived uniquely from the randomly chosen k value (recall Step 3 of the ECDSA signature generation algorithm in Section 3.2). Note that any value of k may be used by the protocol, what matters is that is is fixed beforehand (that is, before the signature is made). On the other hand, the hash flag tag ht is set to `0x01`, which corresponds to `SIGHASH_ALL`. This flag signals that the signed content corresponds to the entire transaction (except the signature scripts themselves). By enforcing that signatures cover the entire transaction, we ensure that a double spending attempt will include a signature different from the one found in the fast-payment transaction, and thus that observing both transactions indeed allows to derive the private key.

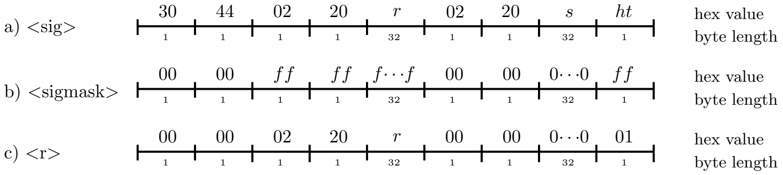
a) <sig>
30 44 02 20 r 02 20 s ht
1 1 1 1 32 1 1 32 1
b) <sigmask>
00 00 ff ff $f\cdots f$ 00 00 $0\cdots0$ ff
1 1 1 1 32 1 1 32 1
c) <r>
00 00 02 20 r 00 00 $0\cdots0$ 01
1 1 1 1 32 1 1 32 1
hex value
byte length

Fig. 8. Values used in the proposed script.

In order to redeem a FR-P2PK output, the `scriptSig` of the input only needs to include a single signature. The signature must be correct when validated with the specified public key and must be performed with a specific r value. Otherwise, the validation will fail.

Let's analyse how does the script perform these validations. First, the script duplicates the signature (`OP_DUP`). This is needed in order to be able to perform different validations over the same signature value. Then, the signature is validated against the specified public key (`<pubKey> OP_CHECKSIGVERIFY`). After that, the length of the signature is checked (`OP_SIZE <0x47> OP_EQUALVERIFY`). Finally, a bitwise AND between the signature and `<sigmask>` is computed (with `<sigmask> OP_AND`), and the result is compared with `<r>` (`<r> OP_EQUAL`). If both values are equal (that is, the signature was made using the specified r and with a hash flag of `0x01`), the script terminates successfully; otherwise, the script terminates with a False value on the stack, making it fail.

Note that the only way to ensure that the script succeeds is by providing a valid signature that matches the specified `<r>`. Therefore, two different transactions spending the same FR-P2PK output would include two different signatures made with the same private key and the same k, and thus by obtaining the two transations one is able to infer the private key that was used to create the signatures.

We have created a funding transaction in the Bitcoin testnet that exemplifies the proposed protection mechanism. Following the transaction naming used in

Figures 2, 3, and 4, the funding transaction[7] contains the output with a FR-P2PK script. The output of the funding transaction is currently unspendable due to the fact that it uses an `OP_AND` opcode that is disabled in the standard Bitcoin software.

[7] http://tbtc.blockr.io/api/v1/tx/info/8e27cae62d1df357b65b634a8482672d85f71804a5c7fc392050517a5bfeb04f