Bitcoin Private Key Locked Transactions (Short paper)

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

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

Abstract. Bitcoin smart contracts allow the development of new protocols on top of Bitcoin itself. This usually involves the definition of complex scripts, far beyond the requirement of a single signature. In this paper we introduce the concept of private key locked transactions, a novel type of transactions that allows the atomic verification of a given private key (belonging to an asymmetric key pair) during the script execution.

1 Introduction

Bitcoin transactions require the execution of a contract, that defines the terms under which the transaction can actually be redeemed. The contract is defined by the scripts provided by the sender (locking script) and the receiver or redeemer of the transaction (unlocking script).

Hash locked outputs are transaction outputs that require the redeemer to prove he knows the preimage of a given hash in order to redeem the output. So that to create this construction, the transaction output includes the value of the hash h, such that $h = H(m)$ for some m, and a script that specifically asks for the preimage of h (where H is a cryptographic hash function). Then, the output can only be spent by providing a preimage m, whose hash is exactly h.

Hash locks have recently gained popularity in the Bitcoin system because they are used, in combination with time locks, to create bidirectional micropayment channels [4, 9]. This kind of channels allows to securely transfer Bitcoins between parties minimizing the amount of transactions stored in the blockchain.

The specific case of a hash locked transaction where the preimage m is a symmetric key is of special interest as a building block for protocols that operate on top of Bitcoin. For example, a possible idea behind this kind of schemes is to encrypt data using a symmetric key cryptosystem, release the encrypted data, and then create a hash locked output where the unlocking value corresponds to the symmetric key used to encrypt the data. The amount of bitcoins deposited into that output corresponds to the price that is going to be payed for ability to decrypt (and thus obtain) the data. In order to collect the bitcoins deposited into the output, the symmetric key must be disclosed, thus allowing the decryption of the data [5].

Extending this idea, in this paper, we introduce and propose private key locked transactions where the locked value is precisely a private key from an asymmetric key pair. Note that standard scripts requiring a signature are already useful to proof the possession of a private key (because the private key is needed to create the signature). However, the scheme we propose requires not only to proof that the private key is known in order to redeem an output but also to reveal that private key. We believe this will open up the possibility to build new applications on top of Bitcoin.

2 Private key locked transactions

Although current hash locked transactions can be used to encode a symmetric key as the unlocking value, or even private keys from asymmetric key pairs, the fact that a private key is verifiable with its corresponding public key can be used to provide a more robust and secure approach.

Symmetric keys are *arbitrary*, in the sense that they are just a bit string. There is, therefore, no direct way of certifying that a specific symmetric key is valid, apart from ensuring that it correctly decrypts data which was previously encrypted with the same key.

On the contrary, given a private key, one can check whether such private key and a public key belong to the same key pair. Regarding ECDSA (Elliptic Curve Digital Signature Algorithm), this test is trivial: one must check if the integer representing the private key multiplied by the elliptic curve base point (which is a known system parameter) results in the curve point that represents the public key. The current `secp256k1` implementation used by Bitcoin to perform ECDSA operations already includes a function that multiplies the generator by a given scalar (`secp256k1_ecmult_gen`). This function is used, for instance, during the process of computing ECDSA signatures.

If we were able to create new Bitcoin script opcodes, we could implement a new crypto opcode to perform exactly this validation: `OP_CHECKKEYPAIRVERIFY`. The `OP_CHECKKEYPAIRVERIFY` opcode would pop out of the stack two values, `pubKey` and `privKey` (corresponding, respectively, to an ECDSA public key and private key) and check whether the two keys match. The execution will then fail if the keys do not belong to the same keypair. Instead, if the keys match, the opcode would push back the two operands to the stack and the execution will continue.

Note that the execution of `OP_CHECKKEYPAIRVERIFY` would fail if the validation is unsuccessful and would leave the stack as it was before if the validation is successful. This ensures that the new opcode can be implemented as a soft fork by reusing one of the currently unused `OP_NOPx` opcodes, in a similar way that it has been done in the past with `OP_CHECKLOCKTIMEVERIFY` (`OP_NOP2`) and `OP_CHECKSEQUENCEVERIFY` (`OP_NOP3`).

With the usage of this new opcode, a transaction output could be constructed such that, in order to be redeemed, the private key matching the specified public

key has to be revealed. An example of the `scriptPubKey` of one such output together with the `scriptSig` needed to spend it would be:

```
ScriptPubKey: <pubKey> OP_CHECKKEYPAIRVERIFY OP_NIP OP_CHECKSIG
ScriptSig:    <sig> <privKey>
```

The script will first check that the public and private keys belong to the same key pair. Note that, if the validation is successful, the stack values will remain untouched. Therefore, before checking the validity of the signature with `OP_CHECKSIG`, the `privKey` value has to be removed from the stack (since it is not needed for signature validation). The execution of `OP_NIP` removes `privKey` from the stack. Finally, `OP_CHECKSIG` validates the signature with the public key. If the signature is correct, the script terminates successfully.

3 Private key locked tansactions relying in ECDSA

Although being able to use the `OP_CHECKKEYPAIRVERIFY` opcode proposed in the previous section would be useful in building Bitcoin smart contracts, the fact is that this opcode does not currently exist. Therefore, `OP_CHECKKEYPAIRVERIFY` may be proposed as a BIP following the work flow of how new Bitcoin features are proposed. Meanwhile, we propose to use a vulnerability in the ECDSA signature scheme in order to achieve the same goal, that is, in order to build transaction outputs that require the disclosure of a specific private key to be redeemed, ensuring that the private key matches a targeted public key. Note that this vulnerability in our case is considered as a property of ECDSA, which allows us to nicely implement the key verification.

3.1 ECDSA vulnerability

ECDSA (Elliptic Curve Digital Signature Algorithm) is the crytographic algorithm used by Bitcoin to create and validate digital signatures. ECDSA has a set of system parameters: an elliptic curve field and equation `C`, a generator G of the elliptic curve `C`, and a prime q which corresponds to the order of G. The values for these parameters are defined to be `secp256k1` [3] for Bitcoin.

Let use denote by $*$ 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. $s = k^{-1}(m + rd) \mod q$
5. Output[1]: $sig(m) = (r, s)$

[1] A new integer k is chosen and the procedure is repeated if either s or r are 0.

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 [6, 8]) 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, $sig_1(m_1) = (r, s_1)$ and $sig_2(m_2) = (r, s_2)$ 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:
$$s_1 = k^{-1}(m_1 + rd) \mod q \Rightarrow ks_1 = m_1 + rd \mod q$$
$$s_2 = k^{-1}(m_2 + rd) \mod q \Rightarrow ks_2 = m_2 + rd \mod q$$
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 [7, 1].

Some Bitcoin wallets adopted deterministic ECDSA after this vulnerability was found to affect some Bitcoin transactions [10, 11, 2].

3.2 Private key disclose mechanism

Our proposed scheme makes use of the aforementioned ECDSA vulnerability to perform targeted private key disclosure within Bitcoin. The Private key disclosure mechanism we propose allows to construct transaction outputs that need to reveal a private key in order to be redeemed, in such a way that we ensure the revealed private key is the counterpart of a certain public key.

Let $\{PK, SK\}$ be an ECDSA key pair belonging to Bob (with $Addr(PK)$ the Bitcoin address associated to it) and sig_{prev} an existing signature made with SK. Alice (that is interested in acquiring Bob's private key) needs to know the value of the previous signature sig_{prev}, in order to be able to request, afterwards, a second signature made with the same k. The previous signature may appear in the blockchain as the input script of an existing transaction. For instance, if some amount of bitcoins were sent to $Addr(PK)$ with a standard pay-to-pubkey-hash script output and Bob has already transferred those bitcoins to another output

by showing a valid signature made with SK, sig_{prev} will be publicy available in the Bitcoin blockchain. Therefore, any observer will know this value and the signed message m will correspond to a Bitcoin transaction hash.

Once an existing previous signature sig_{prev} is known by Alice, she creates a transaction with an output that requires a second signature sig to be spent. However, instead of using the classical pay-to-pubkey-hash script, she uses a special script that forces Bob (the redeemer) not only to prove he has the private key SK associated to the given address $Addr(PK)$ by creating a valid signature, but also to deliver a signature that has exactly the same k value that was used when creating sig_{prev}. The output may also have a time lock that allows Alice to get back her bitcoins if Bob chooses not to reveal the private key.

Doing so accomplishes two purposes: on the one hand, Bob proves he knows the private key associated to the public key by generating a signature that correctly validates with that public key; on the other hand, Bob is implicitly revealing the private key associated to the same public key. Note that Bob does not directly provide the private key, but provides information from which the private key can be derived.

Moreover, the operation is atomic, in the sense that Bob gets Alice's bitcoins (the amount deposited into the output) only when Alice gets Bob's private key (derived from the two signatures by exploiting the reusage of k ECDSA vulnerability).

Furthermore, unlike when revealing symmetric keys with hash locks, the private key disclosure mechanism allows to validate that the leaked secret key is correct, that is, it matches the specified public key.

Figure 1 shows an scheme of the Bitcoin transactions involved in the construction of a a private key locked output. In this example, the input of transaction Tx_2 contains the signature sig_{prev} made by Bob in the past.

Once the previous signature is known, Alice can construct the transaction Tx_4, that transfers some bitcoins of her property to Bob, only if Bob provides a valid signature that has the same r as the previous signature sig_{prev} that appeared on Tx_2. Moreover, the output has an additional condition with a time lock allowing Alice to get a refund of her bitcoins if Bob decides not to colaborate and does not redeem Tx_4's output.

In the next section, we describe how to construct the output of Tx_4 taking into account Bitcoin's signature format and Bitcoin's scripting language.

3.3 Implementation

Although ECDSA signatures are made of two values, r and s, Bitcoin signatures are just a single hexadecimal value, which corresponds to the DER encoding of the two-element sequence of the two integers. Figure 2 describes the format of a Bitcoin signature, where z denotes the bytes representing r and s (that is, for each value, the `0x02` integer flag, the size, and the value itself) and ht denotes the hash type, a flag that indicates the parts of the transaction that are hashed and signed.

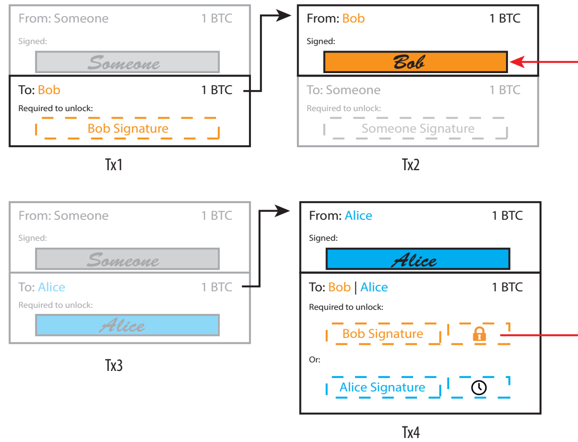

Fig. 1. Transactions involved in the scheme.

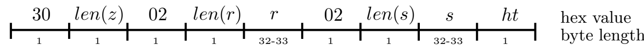

Fig. 2. Bitcoin signature format.

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.

Recall that Alice was in possession of a previous signature sig_{prev} that Bob had made in the past, and that Alice wanted to construct a transaction output that can be redeemed by Bob only if he presents a new signature sig that uses the same k (i.e. has the same r component).

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 3a). Taking into account the format of signatures in Bitcoin, Alice can construct a signature mask sig_{mask}: a byte array that has ones on the positions where r is specified and zeros in the rest of positions. Figure 3b shows the construction of sig_{mask}. Alice can also construct a byte array r_{prev} (Figure 3c) that results from the bitwise AND operation between the previous signature sig_{prev} and the signature mask sig_{mask}.

Finally, Alice can create an output that requires a second signature sig with the same r as the previous signature sig_{prev} by using the values sig_{mask} and r_{prev} she has computed. The `ScriptPubKey` of the output (and its corresponding `ScriptSig`) would then be:

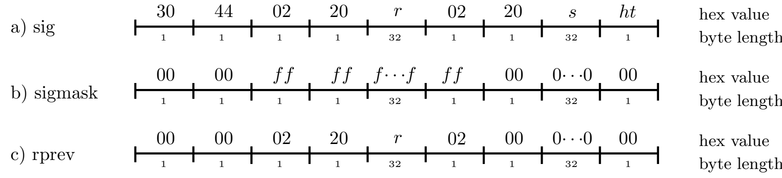

Fig. 3. Values used in the proposed script.

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

First, the script validates the signature against the specified public key. Then, the length of the signature is checked. Finally, a bitwise AND between the new signature and sig_{mask} is computed, and the result is compared with r_{prev}. If both values are equal (that is, both signatures have the same r and thus were made using the same k), 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 has exactly the same r as the previous signature. Therefore, although the redeem `ScriptSig` that spends the output does not include the value of the private key directly, it is implicitly leaking its value by the ECDSA vulnerabilty.

Also note that the `ScriptSig` needed to spend the output only requires one value: the new signature.

We have created a set of transactions in the Bitcoin testnet that exemplify the proposed protocol. Following the transaction naming used in Figure 1, the input[2] of Tx_2 contains a signature made by Bob with his private key SK, together with the public key PK needed to validate it. This signature is public, and thus can be used as the previous signature sig_{prev} needed by our scheme. The private key SK is, therefore, the private key that is going to be disclosed by Bob by providing a second signature with the same k. With this previous signature, Alice can construct Tx_4, whose output[3] contains the special script that requires a second signature from Bob with the same private key and the same k in order to redeem it.

3.4 Discussion

In the implementation performed in the testnet, the output of transaction Tx_4 is currently unspendable due to the fact that it uses an `OP_AND` opcode that

[2] http://tbtc.blockr.io/api/v1/tx/info/ea3f5444aee210f632d00cf208cc69d6f2cb70b42d73a0afa32b6a05794c551d
[3] http://tbtc.blockr.io/api/v1/tx/info/a5a985ee80a68434c46f9f3216b7fe294cd6de9c82f3bcf119ad6ab4c2e13e49

is now disabled. The `OP_AND` opcode was disabled together with many other script opcodes because developers were worried about the security of the existing implementation.

Another detail that has to be taken into account is that, upon finalization of the protocol, a private key corresponding to a specific Bitcoin address is revealed. Therefore, the owner of that private key must ensure that there are no bitcoins left from previous uses of the address. This can be easily accomplished by using a new public key pair for the execution of the protocol or by just transferring any existing funds to other addresses before making the second signature (that effectively discloses the private key).

4 Conclusions

We have presented a private key locked transaction for Bitcoin, where the redeemer is required to disclose a given private key corresponding to a ECDSA asymmetric key pair. This can be achieved by the creation of a new opcode for the Bitcoin Script language, or by disclosing the private key using an existing vulnerability of the ECDSA signature scheme. In the later case the private key can be obtained from two different signatures of the same key re-using a given parameter. Although our current implementation uses a currently disabled opcode (`OP_AND`), we have shown the feasibility of the implementation of our proposal.

We believe that, as shown by the recent popularity of hash locked transactions, a private key locked transaction can be used to implement novel protocols on top of Bitcoin, without introducing more complexity or the need to significantly increase computational capabilities of the Bitcoin Script language.

References

1. 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.
2. Bitcoin.org. Android security vulnerability, 2013.
3. Certicom Research. Sec 2: Recommended elliptic curve domain parameters. Technical report, Certicom Corp., January 2010.
4. Christian Decker and Roger Wattenhofer. A fast and scalable payment network with Bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.
5. Gregory Maxwell. The first successful Zero-Knowledge Contingent Payment, February 2016. https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/.
6. Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.
7. 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.
8. Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

9. Joseph Poon and Thaddeus Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. Technical report, 2015. https://lightning.network.
10. Nils Schneider. Recovering Bitcoin private keys using weak signatures from the blockchain, 2013.
11. Filippo Valsorda. Exploiting ECDSA failures in the Bitcoin blockchain, 2014.