INPUT | OUTPUT

IOHK | RESEARCH BLOG

TRANSACTION MALLEABILITY IN CRYPTOCURRENCIES

RESEARCH

SEPTEMBER 14, 2016 DMITRY MESHKOV 3 MIN READ

In this article I'm going to provide a brief review of protection methods against replay attacks, arising from signature malleability of elliptic curve cryptography.

Problem

Most cryptocurrencies are based on public-key cryptography. Each owner transfers coins to the next one digitally signing the transaction `Tx` containing the public key of the next owner.

Thus everyone can verify that the sender wants to send her coins to the recipient, but a problem arises - how to prevent the inclusion of transactin `Tx` in the blockchain twice? Without such a protection an unscrupulous recipient may repeat `Tx` as long as the sender has enough coins at his balance, making it impossible to reuse the same address for more then 1 transaction. In particular the adversary can withdraw some coins from an exchange and repeat this transaction until there are no coins left on exchange (such attacks have already been executed in practice, e.g. for MtGox attack).

The simplest way to keep all the included transactions and compare the new one to them doesn't work because of the elliplic curve signature malleability - it is possible to change a signature but keep it valid at the same time (see here).

Scala code that changes signature like that is very simple:

```
def forgeSignature25519(signature: Array[Byte]): Array[Byte] = {
    val modificator: BigInt = BigInt("7237005577332262213973186563042994240857116359379907606001950938285454250989")
    signature.take(32) ++ (BigInt(signature.takeRight(32).reverse) + modificator).toByteArray.reverse
}
```

Thus, now we have a sequence of transactions `Tx1`, ..., `TxN` with the same fields, but different signatures, and there's a challenge to determine whether they are all generated by the sender or some of them are generated by the adversary.

Solutions

In this section I'll provide examples of how this problem is solved in different cryptocurrencies and will try to describe the merits and drawbacks of each approach.

Canonical signature (Factom, Ripple, Nxt)

As long as there is sequence of valid signature, there may be a rule to select only one of them. The usual way is to select *canonical*

signature, which is lower then group generator ("7237005577332262213973186563042994240857116359379907606001950938285454250989" for curve25519). Unfortunately for some elliplic curves for any given canonical signature an alternative form of that signature can be formed that is also canonical. In such a case it's required to define *fully canonical signature*, being the minimum of all equivalent signatures. The main drawback of this approach is that *fully canonical signature* is not specified in the protocol and default elliplic curve implementations don't usually check if a signature is*canonical* and don't generate *canonical signature*, which makes cross-platform implementation much harder.

Signature independent id

It's also possible to modify transaction uniqueness check, leaving all possible signatures valid as specified in elliptic curve protocol. For example it's possible to use the rule that the transaction data **excluding** yje signature should be valid. That has a drawback of being unable to create 2 transactions with the same fields, while non-deterministic signatures may indicate that sender really wanted to send 2 transaction with the same fields. To fix this it's possible to include transaction id into transaction explicitly, e.g. some cryptocurrencies sign transaction, use this signature as id and then sign this *internal*transaction with signature one more time.

Nonce (Ethereum, Waves)

Another way is to add an additional field to the transactions, increasing for transactions from the same address. Current nonce for every account should be stored in the blokchain state and with each new transaction `Tx` nodes verify that `Tx.nonce = account.nonce + 1`. On top of replay attack protection nonce allows you to broadcast sequence of transactions and be confident that only one of them will be included to blockchain.

For example, if you need your transaction to be included in block as soon as possible, you may rebroadcast your transaction with the same nonce, but increased fee and be sure, that only one of your transactions will pass all checks. Nonce provides additional benefits for transactional layer, but not for free - every transaction should contain nonce, so transaction size become bigger. On the other size nonce should be big enough, because it's not clear how to handle a situation when nonce limit has been reached. To mitigiate this it's possible to reuse a transaction field as nonce, for example use timestamp as a nonce.

In such an approach it's not possible to require to increase nonce by 1, so rule `Tx.timestamp > account.timestamp` should be used. This leads to another attack: if someone broadcasted sequence of transactions `Tx1`, ...,`TxN` with increasing timestamp, "evil" miner may only include `TxN` making transactions`Tx1`, ..., `TxN-1` invalid.

Conclusion

It's not yet clear to me what the best way is to protect against replay attacks, arising from signature malleability of elliptic curves - each approach has it's benefits and drawbacks. This approaches may be combined with each other, e.g. it's possible to require *canonical*signature together with nonce. Feel free to provide more approaches and examples in comments, it would be cool to choose an optimal solution!