Resisting replay attacks efficiently in a permissioned and privacy-preserving Blockchain network

Elli Androulaki, Angelo De Caro, Thorsten Kramp, David W Kravitz,
Alessandro Sorniotti, MarkoVukolic, Binh Nguyen,
Srinivasan Muralidharan, Sheehan Anderson

1. Background

We are considering the following setting:

A Blockchain system consisting of clients (submitting transactions), and validating entities (executing and validating transactions). Transactions submitted by clients is signed by a client certificate, that can be anonymous or carry that client's identity. These certificates are managed by an identity management infrastructure that issues user long term certificates that carry these users' identities (aka enrollment certificates), and privacy-preserving certificates (aka transaction certificates) that are not linked to their owner's identity. Transactions can be signed by the secret key corresponding to either type of certificates to accommodate user transactional privacy.

This method offers (efficient) replay attack protection in this setting. In replay attacks the attacker simply "replays" a message he "eavesdropped" on the network or "saw" on the Blockchain. Replay attacks are a big problem here, as they can incur into the validating entities re-doing a computationally intensive process (contract invocation) and/or affect the state of the corresponding contract, while it requires minimar or no power from the attacker side. To make matters worse, if a transaction was a payment transaction, replays could potentially incur into the payment being performed more than once, without this being the intention of the payer.

2. Related Work

Existing systems resist replay attacks as follows:

A) Record hashes of transactions in the system. This solution would require that validators maintain a log of the hash of each transaction that has ever been announced through the network, and compare a new transaction against their locally stored transaction record. Clearly such approach cannot scale for large networks, and could easily result into validators spending a lot of time to do the check of whether a transaction has been replayed, than executing the actual transaction.

B) Leverage some state that is maintained per user identity (Ethereum). Ethereum keeps some state, e.g., counter (initially set to 1) for each identity/pseudonym in the system. Users also maintain their own counter (initially set to 0) for each identity/pseudonym of theirs. Each time a user sends a transaction using an identity/pseudonym of his, he increases his local counter by one and adds the resulting value to the transaction. The transaction is subsequently signed by that user identity and released to the network. When picking up this transaction, validators check the counter value included within and compare it with the one they have stored locally; if the value is the same, they increase the local value of that identity's counter and accept the transaction. Otherwise, they reject the transaction as invalid or replay. Although this would work well in cases where we have limited number of user identities/pseudonyms (e.g., not too large), it would ultimately not scale in a system where users use a different identifier (transaction certificate) per transaction, and thus have a number of user pseudonyms proportional to the number of transactions.

Other asset management systems, e.g., Bitcoin, though not directly dealing with replay attacks, they resist them. In systems that manage (digital) assets, state is maintained on a per asset basis, i.e., validators only keep a record of who owns what. Resistance to replay attacks come as a direct result from this, as replays of transactions would be immediately be deemed as invalid by the protocol (since can only be shown to be derived from older owners of an asset/coin). While this would be appropriate for asset management systems, this does not fot the needs of a Blockchain systems with more generic use than asset management.

3. Detailed description of our Method

For replay attack protection we suggest a hybrid approach. In general, we require that users add in the transaction a nonce that is generated in a different manner depending on whether the transaction is anonymous (followed and signed by a transaction certificate) or not (followed and signed by a long term enrollment certificate). In particular:

- Users submitting a transaction with their enrollment certificate should include in that transaction a nonce that is a function of the nonce they used in the previous transaction they issued with the same certificate (e.g., a counter function or a hash). The nonce included in the first transaction of each enrollment certificate can be either pre-fixed by the system (e.g., included in the genesis block) or chosen by the user. In the first case, the genesis block would need to include $\mathtt{nonce}_{\mathtt{all}}$, i.e., a fixed number and the nonce used by user with identity `IDA` for his first enrollment certificate signed transaction would be

  $$\mathtt{nonce\_round0}_{\mathtt{IDA}} \leftarrow \mathtt{hash(IDA,\ nonce}_{\mathtt{all}}\mathtt{)},$$

  where `IDA` appears in the enrollment certificate. From that point onward successive transactions of that user with enrollment certificate would include a nonce as follows

  $$\mathtt{nonce\_roundi}_{\mathtt{IDA}} \leftarrow \mathtt{hash(nonce\_round\{i-1\}}_{\mathtt{IDA}}\mathtt{)},$$

  that is the nonce of the i^{th} transaction would be using the hash of the nonce used in the $\{i-1\}^{th}$ transaction of that certificate. Validators here continue to process a transaction they receive parse, as long as it satisfies the condition mentioned above. Upon successful validation of transaction's format, the validators update their database with that nonce.

  Storage overhead:

  i. on the user side: only the most recently used nonce,
  ii. on validator side: O(n), where n is the number of users.

- Users submitting a transaction with a transaction certificate should include in the transaction a random nonce, that would guarantee that two transactions do not result into the same hash. Validators add the hash of this transaction in their local database if the transaction certificate used within it has not expired. To avoid storing large amounts of hashes, we introduce validity periods in the transaction certificates, and require that we maintain an updated record of received transactions' hashes within the current or future validity period.

  Storage overhead (only makes sense for validators here): O(m), where m is the approximate number of transactions within a validity period and corresponding validity period identifier (see below).

Expiration of validity periods: The challenge here derives by the distributed nature of the system, and the need for all the validating entities to share the same view w.r.t. when a particular validity period has expired in relation to the transactions that are to be executed and validated.

For us to guarantee that expiration of validity periods is done in a consistent way across all validators, we first introduce the concept of *validity period identifier,* that is used to uniquely identify a validity period, and acts as a logical clock for the system. It is essential that validity period identifiers are increasing over time, such that there is a total order among validity periods. We then leverage validity period identifiers, and a special type of transactions, *system transactions,* to announce the expiration of a validity period to the Blockchain. System transactions refer to contracts that have been defined in the genesis block, and are part of the infrastructure.

We define one such contract for management of validity periods as follows:

- The contract defines functions for expiring a validity period X, and automatically activating validity period X+1
- The contract's state consists of (among others) the current validity period identifier, and potentially some metadata (e.g., the real-time period this validity period corresponds to)
- The contract specifies the appropriate CA(s) as the only authorized system entities to invoke expiration of validity period
- The contract specifies ways to extend the list of the entities authorized to expire validity periods

The CA responsible for issuing transaction certificates (we call it TCA), has a local counter vid that initializes to vid = 0, at time = t_0 (local time). Assuming that the TCA has set the validity period length to DT, it acts as follows:

i. for each transaction certificate request received between $(t_0, t_1) \leftarrow \{0, DT\}$, it issues transaction certificates with identifier vid = 0, and metadata (t_0, t_1),
ii. when time $t_1 \leftarrow t_0 + DT$ arrives, it issues a transaction to expire validity period "vid = 0", and sets local validity period vid$\leftarrow$vid + 1
iii. for each transaction certificate request received between $(t_i, t_{i+1}) \leftarrow \{i*DT, (i+1)DT\}$, it issues transaction certificates with identifier vid = i
iv. when time $t_{i+1} \leftarrow t_i + DT$ arrives, it issues a transaction to expire validity period "vid = i", and sets local validity period vid$\leftarrow$vid + 1

As this system transaction is added to the total order of transactions in the network, the validators execute the respective transaction, and store the updated state of the validity contract in the cryptographically immutable ledger. The state on the ledger can be accessed with the validity contract ID (chaincodeID), which is placed in a special location for the system contracts to look up their IDs. In this way they can easily access the updated state every time a transaction's certificate / signature needs to be validated against the current validity period. Transactions that follow the system validity transaction containing an expiration of validity period, should be validated assuming the freshly updated validity period identifier. In addition, after the update of a validity period has committed the validators garbage-collect the log of transaction hashes they have collected for replay attack protection and corresponds to the expired validity period. The reason is that replays of older (anonymous) messages would be using expired transaction certificates and need not to be run against the pool of hashes that validators maintain.
Notice that by implementing validity period identifiers as counters, we have an easy way of detecting if a particular transaction certificate has expired or not.

Current status:
Validity period expiration is part of the first release. Replay attack resistance mechanisms will be added next year.