State-based Endorsement

(formerly known as “ownable state”, also known as “key-level validation”)

E. Androulaki, M. Bjoerkqvist, A. De Caro, D. Enyeart, M. Neugschwandtner, G. Singh, A. Sorniotti **(please add your name here if you contributed)**

# Motivation

The current endorsement policies are too coarse grained for certain use cases since changes to each KVS pairs in the namespace of a chaincode must be endorsed according to a single, chaincode-wide endorsement policy. In many cases however, changes to a key-value pair must be endorsed according to a specific endorsement policy, whereas another key-value pair requires a different endorsement policy. This document proposes an approach towards more flexible, per-key (state) endorsement policies.

# Approach

We propose the concept of state-based endorsement. An endorsable state (formerly known as "ownable state") is **a KVS key with an associated validation parameter** that requires specific organizations ("state owners") to endorse changes to the state. The validation parameter of an endorsable state is an endorsement policy with an AND-concatenation of all state owners. This state-specific endorsement policy will be checked every time a transaction modifies the state.

**A prerequisite for state-based endorsement is a mechanism to store per-key metadata** on the ledger that holds the per-key validation parameter.

### Lifecycle of an endorsable state

1. Creation of a state: an endorsable state is created when the CC sets a KVS key’s metadata to an endorsement policy for the first time.
2. Endorsement of a state: when a peer is asked to endorse changes to a certain endorsable state, it signs with its default signing identity by [default](#_yz5xd0w5a3ea). We also introduce approaches for privacy-preserving endorsement with [ephemeral keys](#_bdgkost0cvl2) or [Idemix](#_93zrtqru10q8).
3. Validation of a state: any update of an endorsable state KVS key and endorsement policy is governed by the on-ledger endorsement policy of this key + the endorsement policy of this key in the transaction’s read-write set. See the following section on [validation](#_ymffdfwgqp03) for details.

### Validation with state-based endorsement

Validation rules for keys are always stored on the ledger. In addition, there might be updates to those rules in the block that is currently being validated. When reading the following validation rules, keep in mind that validation in fabric happens in parallel and thus the rules have to be robust enough:

* Regular case without state-based endorsement: If there is no validation parameter stored on the ledger and no validation parameter updated by the current block, just check the chaincode’s endorsement policy. When the validation parameter for a key is set for the first time, this is thus also guarded by the CC’s endorsement policy.
* Regular case with state-based endorsement: If there is a key-level validation parameter stored on the ledger and no validation parameter updated for this key by the current block, check the validation parameter stored on the ledger.
* Validation parameter update cases: If there is a key-level validation parameter stored on the ledger and there is an update for this parameter in the current block
  + proceed with checking against the parameter stored on the ledger if the tx updating the validation parameter is ordered after the current tx.
  + reject the tx unless the tx updating the validation parameter has been marked invalid.

While rejecting transactions that try to modify a key in the same block after a validation parameter has been updated might seem harsh, we expect validation parameter updates to happen less often compared to regular value updates and thus such conflict to occur only rarely.

The alternative would be to validate the conflicting tx based on the new validation parameter set by the previous transaction. However, we can only do this once we know that the updated validation parameter has been committed to the ledger. This includes checks such as MVCC, which are outside the scope of the validation logic and can thus not be handled by the way fabric currently works.

If a transaction touches multiple states, **all** validation parameters need to be fulfilled for the transaction to be valid.

As already outlined above, state-based endorsement introduces validation dependencies. For example if there is a TX5 writing to key “foo”, whose validation parameter is set in TX1 in the same block, this introduces a dependency of the validity of TX5 on the validity of TX1. If TX1 is invalid, TX5 might be valid, however if TX1 is valid, TX5 is certainly not.

We currently see the following options to resolve such dependencies:

* Always assume that the previous transactions are valid. This is currently the case for lifecycle operations such as upgrade, which automatically invalidate all following transactions in the same block that reference the same chaincode.
* Use MVCC: by letting the validation code introduce a read dependency on the key it writes to, it automatically creates a dependency on a version of the validation parameter for this key. The issue is that with such a solution multiple blind writes to the same key in the same block wouldn't work anymore. Only the first one would go through, bump the version and all others fail. On top of that, this would only be the case for keys with state-based endorsement, for "regular" keys blind writes would still pass (as they do now).
* Introduce a dependency graph. By extending the current validation API with additional hooks that allow declaring dependencies first, we can serialize validation of dependent transactions.

# APIs

Chaincode-level APIs:

* SetStateEndorsementPolicy(key, ep) - set the endorsement policy for the given key for regular, public data
* SetPrivateDataEndorsementPolicy(collection, key, ep) - set the endorsement policy for the given key for private data. Notice that the metadata (and thus the endorsement policy) will always stay public.
* GetStateEndorsementPolicy(key string) ([]byte, error) - retrieve the endorsement policy for the given key. If not ep is stored, nil is returned.
* GetPrivateDataEndorsementPolicy(collection, key string) ([]byte, error) - retrieve the endorsement policy for the private data referenced by the given key. If not ep is stored, nil is returned.

Optional convenience layer for ownable state:

* AddOrgs([]string) adds the specified channel orgs to the existing key-level endorsement policy for this KVS key. The first time this function is called, a key-level endorsement policy will be created for this KVS key requiring an endorsement from one peer from each org. Subsequent calls to this function will add the specified orgs to the list of orgs that are required to endorse. Orgs can be listed more than once to require more than one peer org signing.
* DelOrgs([]string) delete the specified channel orgs from the existing key-level endorsement policy for this KVS key. If any org is not present, an error will be returned.
* ListOrgs() []string returns an array of channel orgs that are required to endorse changes to this KVS key

# Variants

Different types of endorsable state exist: Based on the confidentiality requirements of the state, state owners and state data can be either public or private, i.e. known to everyone or only known to the parties involved. Data can be hidden by means of collections. Hiding owners requires making the endorsement policy unlinkable (per-key validation parameters are stored as public metadata) as well as hiding collection membership:

|  |  |  |
| --- | --- | --- |
| State owners | State data | Implementation |
| public | public | State data is stored as regular ledger data. The per-key endorsement policy is an AND-concatenation of all state owners |
| public | private | State data is stored using collections. The implicit or the explicit collection type is used, with the membership allowing read-access of the state. The per-key endorsement policy is an AND-concatenation of all state owners. Note that the owners can be just a smaller subset of the collection's members, s.t. organizations that have an auditing, but not an owning role have access to the data as well. |
| (partly) private | private | State data is stored in the local collection namespace. The clients need to take care of data dissemination to the owners of the state, with the membership information being kept in the local collection as private data as well. The endorsement policy is made unlinkable by employing either ephemeral keys or Idemix credentials. If only partial confidentiality for the owners is desired, only some organizations can use unlinkable credentials. |

Below we list a few possible variants to address privacy requirements wrt. endorsement of the state, i.e. private state owners.

## Standard

The standard version of state-based endorsement is one where the policy is expressed as a logical AND over MSP principals. For example, we could have a transaction with an action that writes K1 and K2 with:

* K1 ep → “AND(OrgA.member, OrgB.member)”
* K2 ep → “AND(OrgA.member, OrgD.member)”

This transaction needs to be endorsed by peers of organizations A, B and D to pass validation. If collections are used for data confidentiality, A, B and D need to part of these collections.

## Privacy preserving (with single-use per-state keys)

The standard variant is not privacy-preserving, in that i) the policy leaks the set of orgs that own that state and ii) peers endorse changes to ownable state using their local MSP’s default signing identity, which by default is a linkable credential. Note that there are other sources of information leakage (for instance, the collection name, if the ownable state is hashed and stored in a collection, or the tx creator field); here we focus on preventing the information leakage generated by ownable state. Other approaches can be used to address other sources of information leakage, for instance, local collections can be used to avoid leaking the collection membership and idemix can be used to ensure unlinkability of tx creators.

A way of ensuring that ownable state does not leak its owners requires peers to endorse changes to ownable state using single-use per-state keys. In other words, every peer that owns ownable state generates a fresh keypair for that ownable state, lists the public key (omitting the org affiliation) in that state’s endorsement policy and uses the private key to create endorsements for every transaction that modifies that ownable state.

### Changes to the policy framework

This approach requires small changes in the policy framework. It is currently impossible to formulate a policy that only contains an expression over public keys, free from any org affiliation. The proposed change would be to introduce a new policy principal which is simply a public key. SatisfiesPrincipal() would return true if the public key at hand is bitwise equal to the one listed in the endorsement policy; signature verification can then proceed as usual.

## Privacy preserving (with idemix)

In this case ensuring that the ownable state does not leak its owners is achieved by using idemix credentials to endorse changes to the ownable state. The advantage of this approach is that the idemix credentials can be chosen s.t. no key-exchange round is needed (and thus no mechanism to distribute keys).

## Privacy preserving (with idemix & application-level endorsement / voting validation)

In this case ensuring that the ownable state does not leak its owners is achieved by using idemix credentials to endorse changes to the ownable state. The per-state endorsement policy would list the idemix pseudonyms of the owners. Alice would first retrieve the pseudonyms of Bob and Charlie using off band communication. Then, when creating a state, Alice sends a proposal to her own peer and signs the resulting rw-set at the application level with her idemix credential. When the transaction is committed, Bob and Charlie recognize that the state’s endorsement policy requires them to endorse. Thus they also create a proposal, run it against their own peers and sign it at the application level with their idemix credential. A custom VSCC picks up the individual transactions, checks the rw-sets for equality and whether all required endorsements are present. If this is the case, VSCC updates the ledger based on the rwset.