Identities - Mutable & Immutable

As as validator set of a particular blockchain or some long running cryptographic identity, we want to sign over some set of seed nodes that have crypto addresses that will change over frequently. It would be nice to have optimistically quick dissemination of updated naming information for this sort of thing. Presumably we can't let anyone just decide they want to be the identity of a particular string.

What we want is for the identity to be able to name a relationship with another identity where they can broadcast mappings to other parties and the parties who receive those identities can use them. As a validator with sentry nodes, we want people to be able to contact us over the internet, we want to keep our validator and identity keys closely held, and we want our public identities to be ephemeral. We want to be able to broadcast that we have named these new cryptographic identities as our sentry nodes and when parties try to contact us they use those updating identities.

Can we encode this in logical identity land. Logical identities need a proof checker. You can use this to name sub identities. You could use this to indicate Alice.Sentry.9. In terms of mutability we can introduce eventually consistent mutable cache so that people occasionally update their cache inquiries and maybe introduce timeouts. The other is we can bake timeouts into the identities. It would be Alice.Sentry.9.untilJuly15.

If we had eventually consistent mutable names in the logical identity system that would satisfy the request. The logical names could be made immutable with respect to any given cache. The idea is that once someone has cached this identity they have a proof to show this identity is valid. This is useful if you don't want them to look up alice.sentry.9 every time you want to contact them.

It seems putting the timeouts in identities is weird because they link to known clocks which are not objective. If we put timeouts to known history then we can use immutable proof checking thing. You get this name and you know explicitly it becomes invalid. The name itself is immutable This begs the question of figuring out if a name is now out of date into the identity engine. Now the engine needs to know things like what blocks are known. If you included timeout info into an identity name you push responsibility onto whoever is using the name.

Every machine should have at least 2 identity members, one maps logical names to cryptographic keys and one maps keys to physical information. To add a new element you need a cryptographic proof that should go with logical name. The logical name should come equipped with a proof checker/predicate.

For example, Alice uses her pubkey to sign something that says mempool means X where x is another public key and we would use that pub key to authenticate a mempool worker for example. You could write proof checkers this way. A proof checker is just signatures then, but they come from somewhere and we check them once when they receive the name mapping and then we forget about everything.

It seems we can collapse the notion of public key and proof checker they are a functions that takes a message and returns if it is authorized. Using other conditions like block height would be interesting because we still get immutable names

Predicates & PubKeys

There is a difference between predicates and public keys. Predicates proofs can be arbitrarily large and might be comparatively involved. We don't want every message to have to include some arbitrarily huge proof. If already mapped to a crypto key somewhat simplified. Also gives keys as uniform address system for P2p routing as well. The last piece is not meant to represent anything embedded in the prover more than a string. Every time someone goes to look this up in identity mapper, somewhat needs to check making downstream effects more expensive. It could be a predicate check if a signature of this package exists.

There are 3 different cases. Lets touch on public key predicate distinction. This doesn't need to be an interface distinction. It could be a distinction we require that certain logical names have constant costs or a verification function whether implemented as succinct zkps or pubkeys, it shouldn't matter. If we consider the predicates to be circuits or verification function and hash it then we get the content addresses we want.

Mutable Names

If we just want mutable names we can just use the resource trick we have the prefix and suffix prefix can be immutable and suffixes can point to mutable things. Prefix mapped to fixed crypto key and mutable suffix things have less guarantees. We dont want to assume distributed consensus on top of a system we want top build distributed consensus. There is no base mempool for example it is someones' name. Maybe we have locally the name Alice. From the systems perspective Alice.mempool is Alice's pubkey or predicate.

TLS Analogy

When I access a site over TLS and I asked for the certificate which is used to authenticate this there are 2 things that happen.

1. There is an actual pubkey used to talk to hackmd, this is 9b6ec89dafwe.

2. There is a certificate chain used to prove correctness. There is a pubkey and a proof.

You could consider the proof bundled with the pubkey to be one big pubkey thing, itself, serving as a way of having the HackMD address sign stuff. Those are similar to having a separate proof and a public key can sign things on behalf of the identifier hackmd.io If you consider those to be one big bundle and you are not interested in modeling the extra work it takes then every-time you send a message around you could bundle a proof with a message and you wouldn't need to carry that mapping around. Similarly you would have to carry those around in the routing infra to map to logical identities. It is doable if you make those mutable then every-time we are assessing those proofs we are using some type of external reads (time or state from state machine) and would need to reassess them every time we do something because they might have expired since we last checked.

DNS is mutable, and certificate authorities invalidate old certs and resign them. Unless we want to read every time which we dont we are going to have the same kind of consistency guarantees. Is this orthogonal? At some level it is orthogonal weather we use a cached proof or not its a question of where that expiry happens. If you dont have a cache, every time you look at an identity its different.

We should not attempt anything stronger than optimistic eventual consistency. Anything stronger and we should be using consensus.

What inputs are allowed? If we want it to be like DNS what we can do is circulate in validators like this certificate authority (CA) is no longer okay because Im revoking it. To do that everyone who does CA validations has a local cache of revocations that they update optimistically. That means there is a very specific cache that all certificate checks have to go through in the browser. We dont check the proof every time bc that requires checking revocations.

- One of the things we may want has another more recent logical identity been circulated for this logical name. Which implies you have a cache of logical names.
- Alternatively if we want timeouts on clock time we can do clock based lookups. If based on other parts of the system like state machines its even more complicated, Call out queries and response with no guarantee of termination

Which use case do we want mutability for?

Even if things are immutable we say once I declare a logical identity I can't change it. The mutability vs. immutability distinction is one whether or not do we cache this message forever or we can't, its like an infinite expiry in effect. No state reads are required and if you want that you can only use wall clock time. We could make this restriction. Putting the expiry conditions in the name idea is to push the expiry conditions out of the identity engine. The identity engine has this identity called alice.mempool until Tuesday, it doesn't know what this means but this is an identity. The Mempool knows when this identity expires then it looks for alice.mempool until Tuesday.

We can assume the system will not be used in cases where people want strong consistency and ordering guarantees. Do we get weaker guarantees if we have immutable mappings and not timeout? We get stronger consistency guarantees if we have immutable mapping as you dont have all the mutation problems. If we want them mutable then what should the identity engine have access to check? There is a distinction b/w revocation and updating. Revocations in order to be effective should propagate quickly, We are not trying to get this property out of this naming system.

However, the reason you want revocations to be cached is that they are stateful weather the key is valid depends on if I have received a revocation. I may have received a revocation but this particular lookup operation has no way of knowing because there is no storage. It just can't do revocations.

But the revocations for logical names will use the same system as cryptographic identities You cant revoke a cryptographic key, but you can announce it has been compromised.

You can't solve the revocation problem in general. You can disseminate logical messages and cache them. It is useful to circulate a revocation update from a logical identity away form a cryptographic key so we might consider explicitly we want to do so. If we are going to do that then having a local cache is pretty necessary. It is still technically true that you could use logical names and their proof checkers with the same type as their cryptographic keys. If you check a signature by a logical name you have to check the local cache for revocation information (time and info about local state)

We havn't identified the limits of what we want the mapping to be able too read. Should it be able to read wall clock time, revocation cache? State on some local state machine? Its still a fundamentally different type from checking a crypto signature.

Cryptographic key/verification predicates and that type is immutable and might or might not have constant size or time properties. That type doesn't have access to any time related information local cache or time stamp and doesn't t include the word mempool. The predicate as we describe it does have access to that, The predicate would have to be this counts it has to include thing like checking weather a revocation cache has revoked the sig being used all baked into these predicates

Logical names have to include the predicate, its the only way of knowing whether a specific cryptographic identity corresponds to a specific logical name. What that means is that the expiration conditions are also in that predicate. What happens if you combine these 2 notions. The signatures that used to be valid for this logical names are no longer valid as there is side information that goes into every signature check and changes weather a sig is valid or not.This Would be similar to Taiga. If we can constrain the predicates to something that is cheap to check like block time then its find if the overhead is minimal. Not only for the logical naming but also crypto.

We are using the word logical name to refer to 2 different things. There is a predicate that takes in subjective local information or it could take an only objective private info like a signatures meaning everyone who runs the predicate with their same input data gets the same answer. Secondly question of name that has meaning like mempool. We want this if there is some change in the system (introducing) or maybe more expansive.

Indeed, we cannot just have strings that should be mapped to more formal identity because there is no rules to what identity can be assigned to the string. Somewhere there has to be rules and if we have rules there is a predicate. We can have a standardized way that Alice can give meaning to mempool. Alice has a name space at any point in time orders her writes to mempool. If we allow immutability she can assign to 0. There is no objective logical name mapping.

This makes some assumptions about some hierarchical namespace. A hierarchical namespace is just a subtype of predicate which is of the form this sub identity is valid if and only if there is a sig from the parent identity associated with this string. When it comes to updating these over time, the side information is probably something like this is the sub identity associated with this string. There is a danger having identity checks where each check checks cache.

Checking weather the string is in revocation database or timestamp is not slow, it just requires a limiteds scope of state identity can access.

It is reasonable to assume we can restrict predicates by properties by restricting the space we instantiate the system with. Predicates must be cheap. We can solve this later without changing the architecture. If we require decomposition we might want to have the less general thing. If we say its just strings and we dont decompose the strings then i dont need to have any corresponding identity to mempool or workers. This is the question contained within the meaning assigned to the string, we can do both.

There is a bit of a conflation b/w syntax and semantics. Assuming we encode everything in these predicates we can establish the idea of a sub identity a standard format in the predicate of the form a sub identity associated in the string of x and y requires a proof that the parent signed. Its a standard certificate chain sub identity format. When we talk about how the mempool works we might image that there are sub-identities.

When we colloquially say alice.mempool.workers.4 do i mean there is a sub identity alice.mempool? We want to do this lazily. We want to have the identity system see anything as an opaque byte string on the predicate level but not on the resolution level. Once alice introduces mempool and workers the resolution would go through the hoops. Once you introduce composition

As long as the proof can check that alice.mem.4 is established correctly I think you can write predicates that support this. We can just use a different symbol if we want to enforce there is a mempool identity.

One feature this brings up when checking proofs, can I make reads of other proofs in cache?If I send over a signature and its got a crypto key and signature from key and proof that its associated with identity 4, If i give it to someone who doesn’t know Alice its not a good enough proof. If I give it to someone who knows then it is good enough. Signature is valid but only when someone learns about it.

The use case for mutable sub-identities?

You might want to do identity revocation. Alice has been calling this cryptographic ID alice.mother and found out its been corrupted by the adversary and wants to revoke it.This is weak revocation.You could also want for convenience purposes to talk about alice.mempool.worker.4 despite different duties in the system might change public key and machine over time. If we have something like the seen workers designate the function and the 4 designates the node and we dont care about the node we can just invalidate or revoke the 4. You could get rid of it and replace it. The mempool has a notion of different workers and these may change based on different identity. Mempool has specific set of workers they must maintain. It may be that Alices previous worker 4 has failed and she is getting a new one.