Hey all,

This is the first post of what I expect to eventually become a series of ‘Beyond SGX’ – how to combine novel cryptographic techniques into the base layer that will improve the overall security/privacy guarantees of Secret. The critical takeaway is that any of these improvements should not be seen as a patch on top of SGX, but rather, the combination of two technologies that are are stronger together

. As will be made clear, using cryptography alone or SGX alone each yields lesser guarantees than combining both.

Current design

Currently, Secret relies on a randomly generated consensus seed, which was generated on the network’s launch and is passed securely to each new validator’s enclave when they join the network. This seed is then used to produce all the randomness needed in the network, including all relevant keys. Let’s focus on the network key which validators share and use together with users to derive transaction keys.

First, let’s better define what we’re trying to defend against. From a threat model’s perspective, we can model several attackers in a descending order in terms of power/damage:

1. An adversary that can completely break SGX
2. An adversary that can’t break SGX but can over time leak information selectively (i.e, the attacker can target which data it wants to get, for example - the consensus seed)
3. An adversary that can over time leak data through side channels but indiscriminately
4. No leakage is possible

Secret already protects well against adversaries #3

and #4

, because at best you’d leak some bits here and there but you won’t be able to selectively extract keys that allow you to decrypt critical data en-masse. Similarly, #1

and #2

are currently only theoretical, since any attacks shown to date were done in a lab setting. Still, it’s probably beneficial to find ways to better protect against these attackers as well, especially against #2

which is the more realistic one.

So now let’s focus on attackers #1

and #2

. These attackers can trivially break the enclave, and they would only need to break a single enclave to get the network key and decrypt all historical and future data.

Proposed improved network key design

Instead of having a single network key shared across all validators, we suggest that each validator generate its own share of the network key when it joins the network. In practice, this raises some questions that I won’t go into right now, but the end result should be something like this:

1. Each validator with index i holds nkey_i in their enclave
2. nkey := nkey_1 +...+ nkey_n // In practice, you don't need all shares, just 67% of them
3. pub_nkey := nkey*G // essentially the public key corresponding to the shared private nkey

From this, it should be clear that the network key does not live on a single machine anymore, but rather, it’s split across the different validators. You would now need to break 67% (based on voting power) of the validators’ enclaves

to compromise it.

This looks very nice but we’re not at all done. Currently, users and enclaves use a user’s key-pair combined with an enclave’s key-pair to derive a shared key, but we no longer have the private network key stored in any one place (by design…). So how can we do ECDH?

I suggest to drop ECDH completely in favor of the implementation in this paper (one of the authors is Sergey from Axelar). This paper focuses on solving MEV, and I will touch that later, but for now let’s focus on the main idea, which is not using a regular kind of encryption, but rather an identity-based encryption scheme. Identity based encryption schemes allows you to

generate from a single key (the network key) any number of derived keys based on some public data. The important part is that this can be done independently for the private key and the public key. To give an example, imagine we have (nkey, pub_nkey) as before, then there are two functions pk1 := derive_pub(pubkey, public_data)

and sk1 := derive(privkey, public_data)

, where (sk1, pk1) are a valid key-pair.

This is almost all we need, because with this, we can simply use the block-height as the public data, and use that to create a unique key for each block, and both the users and the enclave can do that independently (users can derive the public key to encrypt their transactions, and validators can do this for the secret key).

BUT WAIT, there's one missing detail. Derive works centrally on a single private key, and we only have shares of that network private key. Well it turns out that this is fine and if you use Shamir Secret Sharing this just naively works - every validator can locally use derive() to derive its share of the new key, and then by combining all shares (or as mentioned before - 67%+ of them), we can reconstruct the full block key.

TL;DR the scheme above gives us a way to have a single private key secret-shared across all validators, such that to compromise it you'd need to break into the majority of them. For every block, we'd have a fresh new block_key that all validators still need to get full access to in their enclaves, but leaking that key from the enclave will at most decrypt transactions from a single block. This scheme is therefore much more secure against attacker #2

, and to an extent, against attacker #1

. It's also a future-proof scheme.

One detail I left out is how do the validators share their shares of the keys in each block. Well, this can be done with the upcoming ABCI++ interface, which allows validators to add some data into their votes while running consensus. Each validator can therefore encrypt their block key share with a key that validators only have inside their enclave and append it to their consensus vote. Validators can then pull those encrypted shares into their enclaves, decrypt, and reconstruct the key.

Performance-wise this is apparently very fast, adding about 150ms to the block time with about 100 txs blocks and 100 validators

. These are numbers from the paper so they need to be validated of course.

I will explain later in a comment how this relates to MEV as well.