For the most part, our goal is to keep the input/output encryption protocol built for Discovery largely unchanged. The benefits are obvious - that code exists, and it has been audited. There's one relatively simple issue to resolve which is replacing the secp256k1 library, as it's not known to be constant-time.

Initialization

We assume validators start with the following shared keys. Generating these keys is described in this post:

1. (sk_io, pk_io)

--> shared key for deriving input/output keys. pk_io

is available on-chain, and sk_io

is shared across all validators' enclaves.

1. master_state_key

--> a symmetric master key used to derive other state keys.

1. master_iv

--> an IV seed used to generate fresh pseudo-random IVs for both encrypted outputs and state encryption.

Input Encryption/Decryption

As usual, a secret contract execution should enable users to encrypt their inputs, which will only ever be decrypted inside of secure enclaves. Therefore, this protocol is part of a larger user tx that should trigger a secret contract. A sketch of the protocol proceeds as follows (I included some key pieces from the larger computation protocol, but it's not meant to be complete - just to provide context):

User:

1. User generates an ephemeral secp256k1 key-pair. I'd recommend we still use secp256k1 curve since again, that code has been implemented and audited.
2. Derives a symmetric key by combining the validators' shared public key (pk_io

) with the ephemeral secret key (a simple EC multiplication - basically Ephemeral DH, as we have now).

1. Symmetrically encrypts/authenticates the transaction inputs and the secret contract function, using AES-256-GCM (again, already implemented).
2. Creates a compute tx with the payload (contract_address, enc_func, enc_inputs, ephemeral_pubkey) and sends it to the Enigma chain.

Validators:

1. Receive the tx, see that it's a compute call - forward it to the enclave for processing.
2. Derive the same symmetric key (let's call it ephemeral_secret), by applying the same EC multiplication of the shared secret key with ephemeral_pubkey.
3. Decrypt (func, inputs) = decrypt(enc_func, enc_inputs, ephemeral_secret) using the key generated in #2

.

1. Execute contract::func(inputs).

Encrypted outputs

Encrypted outputs (where the output is encrypted with the sender's key), could be handled in largely the same way. We can use the same ephemeral_secret

to encrypt the output (which is stored as part of the contract's state), and only the sender will be able to decrypt it. This is generally how Discovery handles it today.

The challenge here is that AES-256-GCM, like most encryptions, is a randomized algorithm (i.e., executing it twice on the same message and with the same key will lead to different results). Since unlike Discovery, now we have multiple validators

to deal with per computation, this means they will each end up with a different result, and will not be able to reach consensus (consensus is reached on a deterministic output).

To solve this, one easy option is for the validators to generate a deterministic IV, for example by taking the hash of the tx. This is extremely insecure in cases where there's key reuse, but given that the protocol above generates a fresh key for each execution, this is probably okay.

Still, users may fail to create fresh keys for every execution, or this may become undesirable in time. Also, I might be missing something in my analysis, so I'm proposing an alternative protocol that is slightly more complicated:

Assume that the validators also share a random master_iv. Using that and a KDF, they can generate a unique IV for each transaction by taking: iv = KDF(master_iv || tx_hash). I recommend using HKDF which is already implemented in Ring, since we already modified it to fit inside SGX.

The above protocol needs to be validated. We aren't doing much here, but more eyes are needed so if anyone has feedback - please share!

One thing to mention is that by using randomized IVs, I'm assuming that the key won't be reused A LOT. Specifically, with a 96-bit IV, there's meaningful risk of a collision after 2^48 encryptions (Birthday paradox), so we definitely need to stay significantly below that threshold.

Encrypted State

In light of the above, the current assessment is that getting an encrypted state should not be difficult, assuming we avoid writing a complicated serialization/deserialization protocol (e.g., like the encrypted deltas system that we wrote for Discovery). To be fair, that protocol was mostly useful for really large states, but as we've seen, this is an unlikely product requirement and would require a lot of effort. In other words, it's an optimization and if it's needed - we can evaluate it over time.

The simplest solution to get to an encrypted state, and assuming we have a public state which allows for arbitrary values, is to allow selective encryption of specific state dict entries (i.e., the key is public and the value is encrypted). If a developer wants to encrypt the WHOLE state, then they should have a single attribute called 'state' (or whatever) and fully encrypt a state dictionary as the value.

For this purpose, the validators should also share a symmetric master_state_key

as described in the beginning of this post. With that key, they can use HKDF in the same way to get a fresh deterministic state key for each transactions:

state_key := HKDF(master_state_key || tx_hash)